THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Securing the Foundations of Practical Information Flow Control

MAXIMILIAN ALGEHED

CHALMERS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Securing the Foundations of Practical Information Flow Control
MAXIMILIAN ALGEHED

© 2019 MAXIMILIAN ALGEHED

Technical report 203L
ISSN 1652-876X
Department of Computer Science and Engineering
Functional Programming Division

CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Gothenburg, Sweden 2019

Securing the Foundations of Practical Information Flow Control

Maximilian Algehed

Abstract

Language-based information flow control (IFC) promises to secure computer programs against malicious or incompetent programmers by addressing key shortcomings of modern programming languages. In spite of showing great promise, the field remains under-utilised in practise. This thesis makes contributions to the theoretical foundations of IFC aimed at making the techniques practically applicable. The paper addresses two primary topics, IFC as a library and IFC without false alarms. The contributions range from foundational observations about soundness and completeness, to practical considerations of efficiency and expressiveness.

Acknowledgements

First and foremost, I would like to thank my supervisor Mary Sheeran for her patience. Maybe one day we will write a fun little paper together. I also want to thank my co-authors, Alejandro Russo, Thomas Schmitz, Cormac Flanagan, and Jean-Philippe Bernardy for making writing this thesis so much fun. All my friends at the department deserve a big thank you, especially Alexander Sjösten for always looking out for me and making sure I don't get too much work done. On the topic of such stabilising influences, thank you Herr Ingenjör Sundström for so many years of fun, for so many days to remember, and for the evenings I don't quite seem to remember as well as I should. A very important thank you goes to my family, for their unwavering love and support. A big thank you also to Cătălin Hriţcu, for likely being the only person to ever actually take the time to read and understand the contents of this thesis. Aarne Ranta also deserves a thank you for convincing me that there is more to research than numbers. Saving the best for last, I would like to thank Anna-Maria Unterberger, who has given me all the love and support anyone could ever hope to receive, and more. I love you Anna, to the moon and back as many times are there are real numbers.

Contents

Contents **vi**

1 Introduction **1**
- 1.1 Language-Based Information Flow Control 2
- 1.2 Barriers to Adoption for Language Based IFC and the Role of this Thesis 7
- 1.3 Statement of Contributions 14

2 Paper 1 - Encoding DCC in Haskell **21**

3 Paper 2 - A Perspective on the Dependency Core Calculus **47**

4 Paper 3 - Simple Noninterference from Parametricity **57**

5 Paper 4 - Faceted Secure Multi Execution **79**

6 Paper 5 - Optimising Faceted Secure Multi-Execution **113**

Chapter 1

Introduction

Information processing systems are everywhere. They connect us, assist us in our daily lives, and underpin almost all important societal functions. We place trust in computer programs to handle our personal data and to provide valuable functionality. However, such systems are increasingly under attack. Malicious actors make daily attempts to compromise the confidentiality, integrity, and, availability of important systems. In 2017 Americans saw their confidential credit and social security details leak to the internet in the Equifax breach [25]. In the same year, European nations were hit by devastating ransomware attacks that compromised the availability and integrity of their national health services and other crucial infrastructure [22]. According to some statistics, 2017 was the worst year on record for data breaches, totalling almost 1600 recorded incidents [50]. Other sources indicate a staggering growth-rate for cyber crime in the last decade. In 2015 we saw around 150 incidents, a number that grew to over 1000 in 2016 [17].

The year of 2018 also saw a number of high-profile attacks. The New York Times reports that attackers were able to compromise the supply chain of a number of newspapers [45]. The attacks highlight the need to defend against both data leakage, attacks on confidentiality, and data corruption, attacks on integrity. Attacks against a newspaper may leak confidential information, for example about anonymous sources. Perhaps worse, attacks raise concerns about the risk that attackers may influence what is printed on the physical paper or the newspaper's website, a threat to integrity which may erode trust in the news media as a whole.

Large organisations like the New York Times and Equifax are not alone in being under attack. While attacks where a malicious entity gains access to a trustworthy information processing system like that of the New York Times are severe and harmful, incidents of attackers going after individuals are both more common and more worrying. For a fictitious example of such a scenario, consider a smartphone application, we call it Whelp, for suggesting nearby restaurants. For Whelp to be able to provide this service, the application needs access to the user's location. However, while the user may have agreed to share their location with the application developer, normally by granting the Whelp application access to the smartphone's GPS sensor,

they do not necessarily wish to disclose their location to all the third party developers who may have contributed code in the form of software libraries or plugins used by Whelp. However, once the user grants Whelp access to her location, all the code in the Whelp application has access to the location and can use it at its own discretion.

This model, dubbed discretionary access control (DAC), is the de-facto standard model in place in most phone and desktop operating systems. Unfortunately, it is ill suited to ensure that sensitive information is used correctly within an application. Once the application has access to the information, it is up to the programmer to ensure that secrets do not leak. The DAC model places full trust in all the code of the program, including all third party code imported through software libraries, plugins, and in the case of websites running third party code in the user's web-browser, through dynamic web requests. Clearly, DAC is an inadequate security measure as it gives no guarantee about the actual flow of sensitive information inside a system [43]. To remedy this, we instead turn to Language-Based Information Flow Control (IFC).

1.1 Language-Based Information Flow Control

Language-Based Information Flow Control (IFC) [43] is a collection of programming language features and analyses, referred to as enforcement mechanisms, for ensuring that information flow inside a system conforms to a formally stated security policy. In the previous section we saw an example of a security policy: the user's location is allowed to flow to the main developers of Whelp, but not to authors of any third party libraries. The policy is used in the IFC system to either statically or dynamically track information flow in programs and ensure that no secret information can leak from the system.

A website login form provides another simple example of such a security policy. In this context, a typical security policy says that text entered into the *password* field is *secret* and should not flow, through a HTTP-request or any other means, to any URL other than the domain of the website.

Security policies are not limited to specifying confidentiality requirements, like password secrecy, but can also specify integrity constraints. Typically, integrity is phrased in terms of preventing untrusted, public, inputs from affecting trusted, secret, outputs. In this sense, integrity can be thought of as dual to confidentiality. For example, a very common *integrity* security policy says that public, untrusted, input should not be able to flow un-sanitised to an SQL query, as allowing it to do so could cause an SQL injection vulnerability [41].

Formal Specification. In order to begin to mitigate these types of vulnerabilities, it is important to be able to precisely specify what the intended behaviour of a system is, and specifically to be able to specify what counts as secure or insecure system behaviour. A distinction between what behaviour is secure and not is typically referred to as a "security policy". Clearly, it is necessary to have a precise mathematical

```
s := getSecret();
p := 0;
if s > 0 then
  p := 1;
end
writePublic(p);
```

(a)

```
s := getSecret();
p := 0;
writePublic(p);
p := s + 1;
writeSecret(p);
```

(b)

Figure 1.1: An insecure and a secure program

language in which to express policies in order to be able to reliably enforce them. In language-based IFC, the choice of language is that of (partial) orders of security labels [19]. The order, written $\sqsubseteq$ and pronounced "can flow to", stipulates that information from an input channel, like the GPS sensor on a phone, labeled with security label ℓ can flow to an output channel, like a web domain, labeled ℓ' if and only if $\ell \sqsubseteq \ell'$. Because information in a system can come from the combination of multiple inputs, we usually require that the partial order comes equipped with some way of combining security labels ($\sqcup$), making it a security (join semi-) lattice [19]. The canonical example of a security lattice is the two-point lattice with labels Public and Secret where Public $\sqsubseteq$ Secret but not the other way around.

Given this precisely stated security policy, Secret $\not\sqsubseteq$ Public, we consider the two example programs 1.1a and 1.1b. According to our policy, program 1.1a is not secure, as the secret input influences the value written to the public output, whereas program 1.1b is.

Ensuring that information can not flow contrary to the security policy is a key requirement of IFC enforcement mechanisms. This property is normally referred to as soundness. Soundness means that the attacker observable behaviour of a program for an attacker who can see outputs on channel ℓ_a should not depend in any way on input from a channel with a label that cannot flow to ℓ_a. There are many possible definitions of attacker observable behaviour. In the simplest case we only consider the collective outputs of the terminated program attacker observable. We will adopt this "batch-job" model of programs in the rest of this introduction and make only minor modifications to it in order to account for some of the subtleties of program termination. When phrased in terms of the two-point lattice from above, this means that public output cannot depend on secret input, whereas secret output is allowed to depend on both public and secret input. Other definitions include the attacker being able to observe timing and termination behaviour in addition to the output of the program [28].

In this model, specifying the power of an attacker means specifying only what we mean by the observable behaviour of a program. This, in turn, gives rise to a soundness condition. The general form of a soundness condition outlined above, observable behaviour depends only on observable inputs, is referred to as Noninterference

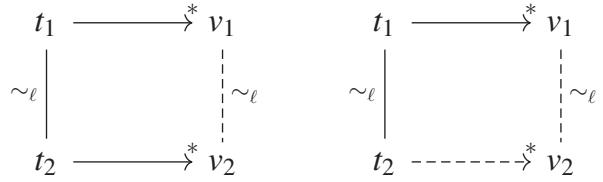
[44]. By varying the definition of observable behaviour, we obtain different variants of the Noninterference condition. For example, when we do not consider termination observably different from non-termination, we get what is called Termination Insensitive Noninterference (TINI). If, on the other hand, we consider termination an observable behaviour, we instead get Termination Sensitive Noninterference (TSNI) [28].

Formal Semantics. To make these notions precise, we need to understand the formal semantics of programming languages. Semantics comes primarily in two flavours, operational and denotational [40]. For now, we will focus on the operational variant, as this is designed to talk directly about how the state of a program evolves under evaluation and therefore is closely related to actual implementations of a programming language, a key to understanding the types of security concerns that arise surrounding real code. We consider the notion of a *configuration*; in imperative languages with side-effects, this is usually a program together with a store and a model of the outside world. In purely functional lambda calculi, like the one we consider in Chapter 5, a program configuration t is simply a term in the programming language. A small-step operational semantics is a binary relation on configurations written $\longrightarrow$. We say that a configuration t "steps to" t' if $t \longrightarrow t'$. We use $t \longrightarrow^* t'$ to say that t takes zero or more steps to get to t'. In other words, $\longrightarrow^*$ is the transitive reflexive closure of $\longrightarrow$ (see [40] for details). Configurations that are in a successfully terminated state and can no longer take a step are called values and are ranged over by the variable v. The reader will find that the usual care has to be taken to not conflate the definitions of a value and a stuck computation.

There are many ways of making the notion of observability precise, based both on operational and denotational semantics [28, 31]. All definitions start with some notion of programs, data, or program traces being ℓ-equivalent, written $t_1 \sim_\ell t_2$. For an example of low equivalence, consider a simple imperative language with configurations (c, σ), where c is a program statement (command) and σ is a store that maps memory locations to values. In this setting, similarity can be defined in terms of labels assigned to the domain of σ, the program variables of c. Given some security policy that assigns labels drawn from the security lattice to the program variables in c, $(c, \sigma_1) \sim_\ell (c, \sigma_2)$ means that all the program variables with labels that can flow to ℓ are identical in σ_1 and σ_2.

In this formalism, we can state common definitions of TINI and TSNI in terms of the diagrams in Figure 1.2. In these diagrams, and similar diagrams to come, we use full and dashed lines to distinguish between the pre-condition of a definition ($t_1 \longrightarrow^* v_1$) and the post-condition ($v_1 \sim_\ell v_2$). In other words, Figure 1.2a says that *given* terms t_1 and t_2 such that $t_1 \sim_\ell t_2$ which reduce to values v_1 and v_2 respectively, *then* it must be the case that $v_1 \sim_\ell v_2$. Likewise, Figure 1.2b says that *if* term t_1 reduces to a value v_1, *then* any t_2 such that $t_1 \sim_\ell t_2$ must follow suit and reduce to a value v_2 such that $v_1 \sim_\ell v_2$.

This definition of TINI can be rephrased as "given two programs which are iden-



(a) TINI (b) TSNI

Figure 1.2: Diagrammatic definitions of soundness conditions.

```
s := getSecret();
while s do
 skip;
end
writePublic(0);
```

(a) A TINI secure program

```
s := getSecret();
if s then
  writeSecret(s + 1);
end
writePublic(0);
```

(b) A TSNI secure program

Figure 1.3: Security with respect to TINI and TSNI

tical from the point of view of ℓ, if both of them terminate then the results are identical from the point of view of ℓ." Similarly, TSNI can be thought of as "if t_1 keeps running forever, then t_2 keeps running forever, and if t_1 terminates, so does t_2 with an ℓ equivalent result." For example, program 1.3a is secure with respect to TINI whereas it is insecure with respect to TSNI. Program 1.3b, meanwhile, is secure with respect to both TINI and TSNI.

There are two primary types of information flows that any sound enforcement mechanism must track, explicit flows and implicit flows [20]. Explicit flow arises when the value of a secret is directly copied from a secret source to a public sink. Implicit flow, on the other hand, arises when the control-flow of the program is modified by a secret in a way which leaks to the public output of the program. This is the type of flow we see in program 1.1a above.

The choice of what types of flows to monitor and prohibit influences the design of the soundness criteria. For example, by only considering the leaks introduced by explicit flows, we obtain variants on the criteria known as "explicit secrecy" [47]. These choices can then be combined with the choice of TINI vs. TSNI (and other, more involved, variants [34, 28, 7]) to form complex soundness conditions.

The choice of soundness criterion depends on how and where an application will be deployed. For example, in a highly concurrent setting like a web-server, a TINI soundness criterion is insufficient to ensure secrets cannot be leaked quickly, and so we require at least TSNI. To see why, consider that while each thread may only leak a single bit via the termination channel, concurrent execution can leak multiple *different* bits in parallel. Inside a smartphone, on the other hand, TINI may

be sufficient to ensure that no sensitive information gets into the wrong hands quickly [7]. In general, the choice of soundness criterion should be heavily influenced by the attacker model and the environment in which the system will operate [12].

The particular soundness criterion that needs to be enforced in turn influences the choice of enforcement mechanisms, as these are usually designed to guarantee only one, or a handful, of such criteria. However, the design and choice of enforcement techniques is not just decided by the soundness criterion. A security engineer needs to consider many factors, like how trusted the programmer writing the code is, and in what language the code is written. For example, if the code is written by a trusted programmer in a type-safe language and the goal is to protect against an attacker who is aiming to perform an SQL injection attack, it may be sufficient to use a weak soundness condition like explicit secrecy [47]. In such a setting, a simple enforcement technique like "taint tracking" may be enough [48]. However, when attempting to protect sensitive information like cryptographic keys from an attacker who is able to call your code multiple times, a stronger security condition like TSNI may be adequate as TINI may leak one bit of sensitive information per run in polynomial time [7]. In some cryptographic applications, it may even be necessary to enforce a *timing-sensitive* [44] version of noninterference, to ensure the absence of information leaks from timing.

Enforcement Mechanisms. Static IFC enforcement mechanisms are usually applied before code is run, either at the time of writing or when inspecting code before running it (like when the browser downloads code from a website), and there are a plethora of techniques based, for example, on type systems, theorem proving, and even abstract interpretation [53, 37, 18, 11, 23]. Static labels on input and output channels are used to approximate an upper bound on "how secret" an expression, variable, or control flow of a program is. These upper bounds are then used to reject potentially insecure program outputs and behaviours. For example, the program 1 a should be rejected because the program variable p, which is written to the public output, depends on secret input through the control flow of the program. Similarly, if we wish to statically enforce TSNI, the program 1.3a should be rejected on the grounds that the termination behaviour depends on the secret input.

Statically deciding whether or not to reject a program whose looping behaviour depends on secret input is difficult. It is well known that program termination is an undecidable problem. As a consequence, any sound static enforcement mechanism for TSNI will reject some secure programs [20].

We say that an enforcement mechanism that rejects or somehow changes the semantics of secure programs is *opaque*. A mechanism that does not suffer from such issues is called *transparent*. Naturally, there are no transparent *entirely static* enforcement mechanisms [20].

Opacity risks hindering the adoption of IFC tools. There are two primary arguments for why this is the case. The first argument is theoretical; many systems include legacy code which was not written with IFC enforcement mechanisms in

mind. Staicu et al. [49] have shown that it is unlikely that such code will, in general, be written in a way which conservative IFC enforcement mechanisms will accept. The second argument is more practical, Alpernas et al. [6] argue that while some modern programming domains do not suffer from opacity, some modern distributed systems require transparent enforcement mechanisms in order to ensure soundness. In conclusion, adding complexity to the software engineering process by introducing limitations on the kind of program the system will accept is likely to hinder widespread adoption of static IFC techniques which lack transparency.

However, one alternative to static enforcement, which aims to reduce false alarms, is dynamic enforcement. A dynamic enforcement mechanism works by modifying the behaviour of a program either by modifying the runtime system of the programming language itself, or the environment in which it is executing. All dynamic systems work by, in some way or another, carrying around information about what inputs have influenced the control flow of the program and the value of the program variables.

Tracking explicit flow is relatively straight-forward to do both statically [33, 35, 53, 55] and dynamically [48]. However, statically tracking implicit flows requires more finesse in order to avoid a large false-positive rate (a high number of secure programs being rejected) [32]. The difference between the difficulty of tracking explicit and implicit flows statically can be put down to explicit flows being a syntactic property computable in polynomial time, and implicit flows being a non-trivial, and therefore undecidable [42], semantic property. Dynamic enforcement mechanisms do not fare much better. In order to be sound, they need to be overly conservative about branches that have not been taken by the program when tracking implicit flows [10], which leads to opacity.

1.2 Barriers to Adoption for Language Based IFC and the Role of this Thesis

While the literature on language based IFC is extensive, including tools, custom programming languages, and software libraries designed to aid application of IFC techniques to real code, the field faces a number of challenges. Despite some lightweight techniques for taint tracking seeing large scale application [48], the field as a whole remains under-utilised in practice. In this thesis, I identify and work to address two major challenges hindering the adoption of IFC techniques to real-world code through both theoretical and practical contributions.

The first barrier to entry is the need for custom languages and tooling. While many IFC systems have been built as extensions to common, established, programming languages like Java or JavaScript, these come in the form of specialised tools like static type checkers or runtime systems [36, 26]. In some cases, the IFC "version" of a language represents a significant departure from the original language [14]. As a consequence, some engineering and design effort still remains in order to integrate these techniques into existing language standards and tools.

One approach to reducing this gap between academic and industrial implementations is the idea of IFC as a library championed by Russo among others [53, 24, 52, 51, 29, 39, 16, 2]. In this approach, IFC systems are embedded wholesale into an existing programming language in the form of a software library. In this thesis, I make extensive use of this technique and every chapter, each based on an published paper, is accompanied by a Haskell implementation of the proposed technique or system. The thesis consists of five different chapters:

1. **Encoding DCC in Haskell** - M. Algehed and A. Russo (PLAS 2017) [4]
2. **A Perspective on the Dependency Core Calculus** - M. Algehed (PLAS 2018) [2]
3. **Simple Noninterference from Parametricity** - M. Algehed and J.P. Bernardy (ICFP 2019) [3]
4. **Faceted Secure Multi Execution** - T. Schmitz, M. Algehed, A. Russo, and C. Flanagan (CCS 2018) [46]
5. **Optimising Faceted Secure Multi-Execution** - M. Algehed, A. Russo, and C. Flanagan (CSF 2019) [5]

IFC as a library. In Chapters 1 and 2, we demonstrate that the "IFC as a library" approach possesses large expressive power even when built on top of *very* small foundations. Using a trusted computing base of only 7 lines of Haskell, we are able to build a library that supports a wide variety of features by using Haskells advanced type system and functional purity. The main idea is to embed the Dependency Core Calculus (DCC) of Abadi et al. [1] and force all sensitive information to be chaperoned by the DCC implementation

The main advantage of this technique is that the difficult part of incorporating effects like state and exceptions into a security library, combining multiple different effectful computations at different security levels in a sound way, is handled by the underlying implementation of DCC. In previous work, this combination of different effects and security levels is made a trusted (although proven correct) part of the implementation [53, 51, 54, 15]. However, when using our approach these primitives become derived operations whose correctness follows from the correctness of the encoding.

While the IFC as a library technique is promising, it leaves a gap between theory and practice. In order to prove that the implementation of the IFC system is sound, authors usually construct a model of the library as a separate program calculus and prove soundness for that calculus, rather than the actual implementation [53]. Given the current state of the art in Haskell verification, this is an unavoidable shortcoming of their work.

However, in Paper 3 we look forward and develop a technique for verifying the implementation of an IFC library directly in the host language. Although this is

```
s := readSecret();
p := 0;
if s then
  p := 1;
end
writePublic(p);
```

(a)

```
s := readSecret();
p := 0;
if s then
  p := 0;
end
writePublic(p);
```

(b)

Figure 1.4: Two similar programs, one is secure and the other is not.

done only for a core IFC library embedded in a language much more expressive than Haskell, it represents a step towards closing the gap between theory and practice. Others have also recognised the power of this approach. For example, Jung et al. [30] present the Iris framework, a similar approach which allows reasoning about libraries using expressive language-specific meta theory.

Transparency. The second issue hindering the widespread adoption of IFC methodologies is the opacity introduced by, for example, imprecise handling of implicit flows. The vast majority of IFC enforcement mechanisms and systems are not transparent, and as such cannot be applied "out of the box" to secure third-party or legacy code not written with the enforcement mechanism in mind. In order to measure the extent of this problem, King et al. [32] employ a variant of the Jif [36] static IFC analysis tool for Java to find information leaks in real Java code. While their analysis revealed a large number of implicit flows (145 out of 162 of detected actual leaks were due to implicit flows) over 90% of the implicit flows reported by the system as possible leaks were false alarms.

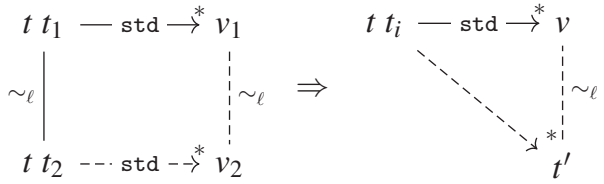
To understand why opacity has the potential to be so devastating, consider programs 1.4a and 1.4b. Program 1.4a is clearly insecure and is rightly rejected by any sound IFC system. Program 1.4b, on the other hand, is not insecure and so should not be rejected. However, typing this program into JSFlow, an IFC aware JavaScript runtime [27], causes it to halt execution before the last line when the secret is equal to 1 in order to prevent information about the secret from "leaking" to the public output. While this program is rejected by JSFlow due to it employing the so-called "No Sensitive Upgrade" (NSU) strategy to deal with implicit flows, the program is accepted by systems based on "Permissive Upgrade" (PU) [8]. Although PU offers some relief, many secure programs are still rejected by this strategy [8, 9].

This behaviour may be deemed acceptable when considering programs like 1.4a and 1.4b that do not compute anything useful. One could argue that no real code looks like this. However, in Paper 5 we show that there are programs that, when written without enforcement in mind, display problematic behaviour similar to that of program 1.4b. For example, when data records of different security levels are kept in the same data structure, like an event queue or a hash table, patterns similar to the

one in 1.4b arise naturally.

The definition of transparency says that "An IFC enforcement technique is *transparent* if it does not alter the semantics of secure programs." There are two parts to this definition, the definition of a secure program and the definition of "alter the semantics". Both these concepts will be made precise in Papers 4 and 5, where we prove that two related enforcement mechanisms are both transparent. For now, we will only cover the technical definitions briefly.

A program is secure with respect to a security policy if for all ℓ, given any two inputs t_1 and t_2 such that $t_1 \sim_\ell t_2$, the program behaves securely with respect to the soundness condition when running *without* the enforcement mechanism. Formally, this means that given a "standard semantics" ($-\texttt{std}\rightarrow$), the TSNI diagram holds with $t\ t_1$ and $t\ t_2$ and $-\texttt{std}\rightarrow$ instead of $\longrightarrow$. The standard semantics is also what provides us with the definition of "alter the semantics", an enforcement mechanism does not alter the semantics of a program if the result of executing under $-\texttt{std}\rightarrow$ and $\longrightarrow$ look the same to an observer at level ℓ, for all ℓ for which the program behaves securely. Using diagrams, the definition can be stated as the following implication holding for all ℓ, t_1, t_2, and t_i:



The examples above make the need for transparent IFC clear and highlight its dual purpose. Transparency both allows IFC to be applied to third party or legacy code, and ensures the security conscious programmer is not forced to program in an awkward or peculiar style. It is clear that providing transparent IFC is an important aspect of practical language-based security.

Multi-Execution. In the literature, however, there are only a handful of enforcement mechanisms for ensuring transparency, collectively known as "multi-execution" techniques. As the name suggest, the idea is to run copies of the same program, or parts of the program, multiple times while altering the input and output behaviour in each run. One early example of this is Secure Multi-Execution (SME) [21]. Under SME, one copy of the program is run for each security level. Security is achieved by giving each copy only the inputs that are visible at its level and using only the outputs to channels labeled with the label of that copy of the program. In the special case of the two-point lattice of public and secret discussed above, SME runs two copies of the program called the public and private runs. The public run is not given the true secret input, rather some default values, and is made to only produce public output. The secret run on the other hand is given both the public and secret input, but is only made to produce secret output. A diagram showing the operation of SME for this lattice with public denoted by the colour blue and secret by red can be found in Figure

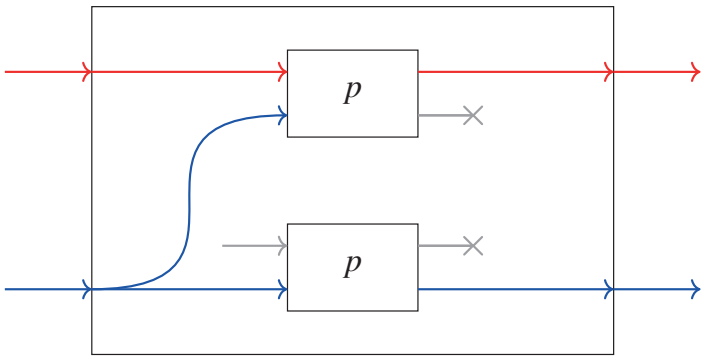


Figure 1.5: Secure Multi-Execution of the program p for the two-point lattice.

```
// Public run                    //Secret run
s := readSecret(); //s = 0       s := readSecret(); //s = 1
p := 0;            //p = 0       p := 0;            //p = 0
if s then                       if s then
  p := 0;                         p := 0;          //p = 0
end                             end
writePublic(p); //output 0      writePublic(p); //no output
```

Figure 1.6

1.5. The simplicity of SME means that it lends itself naturally to a black-box approach, which can, in theory, be applied to any program written in any programming language.

To see how SME works, consider running program 1.4b above with secret input 1. The two runs we get can be seen in Figure 1.6. In the public run, the secret input is given the default value 0 and the public output is 0. In the secret run, however, because we are computing with the actual value of the secret, the public output is blocked. Because the program is secure, the output when running under SME is the same as when running without any enforcement mechanism. If instead of running program 1.4b, we run program 1.4a, an insecure program, the public output would be unaffected by the actual value of the secret, ensuring security.

In contrast to traditional static and dynamic enforcement techniques, SME does not provide any means of detecting attacks. Rather, SME is a form of "program-repair", it does not detect vulnerabilities but automatically creates a secure program from any source program.

While SME is an attractive technique, it suffers a number of drawbacks addressed in follow-up work [9, 38]. SME often requires an excessive number of executions of the same program and fails to work for security lattices with a large or infinite set of labels. For example, several billion labels would be required to provide every user of the social media site Facebook with their own label, making SME a poor fit for this domain.

The limitations of SME prompted Austin and Flanagan to develop Multiple Facet (MF) semantics [9]. Under MF, each program is run with multiple values associated

to each variable, one for each security level which may have influenced the variable. This allows MF to be transparent while being less computationally demanding than full SME. However, it fails to provide the same security guarantees, providing Termination Insensitive security while SME provides Termination Sensitive security [13]. Another drawback of MF compared to SME is that MF is not black-box, as it requires modifying the core semantics of the programming language.

Motivated by the subtly different expressive power, security guarantees, and performance characteristics of SME and MF, in Paper 4 we introduce Multef, a unifying theoretical and practical framework for multi-execution. Multef provides a novel approach to MF and SME, as well as a new execution mode we call Faceted Secure Multi-Execution (FSME) which subsumes both MF and SME and provides a "best of both worlds" approach in terms of performance and security guarantees. Using Multef, we implement a number of case studies and micro benchmarks to show how the performance characteristics of MF, SME, and FSME differ. Multef is implemented using the IFC as a library approach, and therefore benefits greatly from the expressive power of the Haskell language.

This expressive power allows us to build case studies that increase our understanding of the limitations of multi-execution. Motivated by our findings, in Paper 5, we show how naively applying multi-execution incurs significant performance penalties. In the paper, we provide two types of optimisation techniques, dubbed data- and computation-oriented optimisation, which can be employed to reduce overhead introduced by FSME and Faceted databases. In the best case, data-oriented optimisation reduces overheads from exponential to constant space and time without the interaction of the programmer, thereby fulfilling the promise of transparent IFC. Computation-oriented optimisation, on the other hand, relies on the insight that transparent IFC may be too much to ask for in some cases, and instead allows the programmer to insert annotations that help the runtime system reduce overheads. One key insight of this work is that the original formulation of transparent IFC is too rigid to accommodate for our optimisation techniques. In order to remedy this issue we provide a new, stronger, notion of transparency.

Conclusion. In summary, Papers 1, 2, and 3 provide novel insights into the nature of the "security as a library" approach. In particular, Paper 3 shows how to use existing meta-theoretical tools to reason about the implementation of a security library without appealing to an external model of the library. In Papers 4 and 5, meanwhile, we strengthen the foundations of transparent IFC. Specifically, 4 provides a unifying formal framework in which to study the trade-offs between MF, SME, and FSME and 5 uses this framework to highlight and fix key performance issues present in all multi-execution techniques.

Future Work. For security as a library, the goal is to extend the techniques from Paper 3 to full-blown programming languages. I am hopeful that this can be achieved by incorporating ideas like those of the Iris framework [30].

For transparent IFC I am less hopeful. It appears unlikely to me that the techniques from Paper 5 will extend far enough to deal with all performance issues introduced by multi-execution. Specifically, I doubt that it will be possible to find algorithms for deriving optimal multi-execution patterns like those presented for the case studies in the paper automatically. In fact, I consider it important future work to explore the limits of multi-execution and transparent IFC. I will conduct this work with the goal of better understanding when transparency can be achieved efficiently, and when it is necessary to coerce application programmers into using more intrusive approaches like runtime monitors and static analysis.

1.3 Statement of Contributions

Encoding DCC in Haskell It was Alejandro Russo's idea to encode DCC in Haskell. The key idea of using type families was mine. The technical development was performed by me. Both authors contributed to the writing, with Alejandro's primary role being to provide feedback on my writing and structure.

Simple Noninterference from Parametricity The idea behind the paper and the technical development were both mine. Jean-Philippe contributed the key insight of using Sigma types and performed the Agda mechanisation of the core parametricity proof. The mechanisation of the translation of DCC into Agda was done by both authors jointly.

A Perspective on the Dependency Core Calculus This is a single author paper, I did all the work on my own.

Faceted Secure Multi Execution Thomas and I shared the majority of the technical work and the writing equally. I was mainly responsible for the implementation and evaluation of the Multef tool and case studies. The initial prototype implementation was done by Alejandro, while subsequent rewrites of the framework and restructuring to greatly simplify the presentation were performed by me.

Optimising Faceted Secure Multi-Execution Alejandro and I jointly came up with the idea of doing tree-rewriting. The idea of limiting views during computation came from my implementation of a chat server in Multef. The technical development, including proofs, implementation, and type-setting were done by me. The majority of the writing was done by me, Alejandro and Cormac provided useful feedback and input on the entire writing process.

Bibliography

[1] ABADI, M., BANERJEE, A., HEINTZE, N., AND RIECKE, J. G. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), ACM, pp. 147–160.

[2] ALGEHED, M. A perspective on the dependency core calculus. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2018), PLAS '18, ACM, pp. 24–28.

[3] ALGEHED, M., AND BERNARDY, J.-P. Simple noninterference from parametricity. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Functional Programming* (2019), ACM.

[4] ALGEHED, M., AND RUSSO, A. Encoding DCC in Haskell. In *Proc. of the 2017 Workshop on Programming Languages and Analysis for Security* (2017), PLAS '17, ACM.

[5] ALGEHED, M., RUSSO, A., AND FLANAGAN, C. Optimising faceted secure multi-execution. In *Proc. of the 32nd IEEE Computer Security Foundations Workshop* (2019), IEEE Computer Society Press.

[6] ALPERNAS, K., FLANAGAN, C., FOULADI, S., RYZHYK, L., SAGIV, M., SCHMITZ, T., AND WINSTEIN, K. Secure serverless computing using dynamic information flow control. *Proceedings of the ACM on Programming Languages 2*, OOPSLA (2018), 118.

[7] ASKAROV, A., HUNT, S., SABELFELD, A., AND SANDS, D. Termination-insensitive noninterference leaks more than just a bit. In *European symposium on research in computer security* (2008), Springer, pp. 333–348.

[8] AUSTIN, T. H., AND FLANAGAN, C. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (2010), ACM, p. 3.

[9] AUSTIN, T. H., AND FLANAGAN, C. Multiple facets for dynamic information flow. In *ACM Sigplan Notices* (2012), vol. 47, ACM, pp. 165–178.

[10] BALLIU, M., SCHOEPE, D., AND SABELFELD, A. We are family: Relating information-flow trackers. In *European Symposium on Research in Computer Security* (2017), Springer, pp. 124–145.

[11] BARTHE, G., D'ARGENIO, P. R., AND REZK, T. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* (2004), IEEE, pp. 100–114.

[12] BASTYS, I., PIESSENS, F., AND SABELFELD, A. Prudent design principles for information flow control. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security* (2018), ACM, pp. 17–23.

[13] BIELOVA, N., AND REZK, T. Spot the difference: Secure multi-execution and multiple facets. In *European Symposium on Research in Computer Security* (2016), pp. 501–519.

[14] BROBERG, N., VAN DELFT, B., AND SANDS, D. Paragon for practical programming with information-flow control. In *APLAS* (2013), vol. 8301 of *Lecture Notes in Computer Science*, Springer, pp. 217–232.

[15] BUIRAS, P., VYTINIOTIS, D., AND RUSSO, A. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)* (2015), ACM.

[16] CECCHETTI, E., MYERS, A. C., AND ARDEN, O. Nonmalleable information flow control. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017* (2017), pp. 1875–1891.

[17] CENTER, I. T. R. Itrc breach statistics 2005 - 2016. `https://www.idtheftcenter.org/images/breach/Overview2005to2016Finalv2.pdf`, 2017.

[18] DARVAS, Á., HÄHNLE, R., AND SANDS, D. A theorem proving approach to analysis of secure information flow. In *International Conference on Security in Pervasive Computing* (2005), Springer, pp. 193–209.

[19] DENNING, D. E. A lattice model of secure information flow. *Communications of the ACM 19*, 5 (1976), 236–243.

[20] DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Communications of the ACM 20*, 7 (1977), 504–513.

[21] DEVRIESE, D., AND PIESSENS, F. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 109–124.

[22] FIELD, M. Wannacry cyber attack cost the nhs 92m as 19,000 appointments cancelled. `https://www.telegraph.co.uk/technology/2018/10/11/wannacry-cyber-attack-cost-nhs-92m-19000-appointments-cancelled/`, 2018.

[23] GIACOBAZZI, R., AND MASTROENI, I. Abstract non-interference: Parameterizing non-interference by abstract interpretation. *ACM SIGPLAN Notices 39*, 1 (2004), 186–197.

[24] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting data privacy in untrusted web applications. In *OSDI* (2012), pp. 47–60.

[25] HASELTON, T. Credit reporting firm equifax says data breach could potentially affect 143 million us consumers. `https://www.cnbc.com/2017/09/07/credit-reporting-firm-equifax-says-cybersecurity-\incident-could-potentially-affect-143-million-us-consumers.html`, 2017.

[26] HEDIN, D., BIRGISSON, A., BELLO, L., AND SABELFELD, A. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. of the ACM Symposium on Applied Computing (SAC '14)* (Mar. 2014), ACM.

[27] HEDIN, D., BIRGISSON, A., BELLO, L., AND SABELFELD, A. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (2014), ACM, pp. 1663–1671.

[28] HEDIN, D., AND SABELFELD, A. A Perspective on Information-Flow Control. In *Proc. of the 2011 Marktoberdorf Summer School* (2011), IOS Press.

[29] JASKELIOFF, M., AND RUSSO, A. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics* (June 2011), LNCS, Springer-Verlag.

[30] JUNG, R., KREBBERS, R., JOURDAN, J.-H., BIZJAK, A., BIRKEDAL, L., AND DREYER, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming 28* (2018).

[31] KAVVOS, G. Modalities, cohesion, and information flow. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 20.

[32] KING, D., HICKS, B., HICKS, M., AND JAEGER, T. Implicit flows: Cant live with em, cant live without em. In *International Conference on Information Systems Security* (2008), Springer, pp. 56–70.

[33] MARTIN, M., LIVSHITS, B., AND LAM, M. S. Finding application errors and security flaws using pql: a program query language. *ACM SIGPLAN Notices 40*, 10 (2005), 365–383.

[34] MOORE, S., ASKAROV, A., AND CHONG, S. Precise enforcement of progress-sensitive security. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 881–893.

[35] MYERS, A. C., AND MYERS, A. C. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), ACM, pp. 228–241.

[36] MYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java Information Flow. `http://www.cs.cornell.edu/jif`, 2001.

[37] NANEVSKI, A., BANERJEE, A., AND GARG, D. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems (TOPLAS) 35*, 2 (2013), 6.

[38] NGO, M., BIELOVA, N., FLANAGAN, C., REZK, T., RUSSO, A., AND SCHMITZ, T. A better facet of dynamic information flow control. In *WWW'18 Companion: The 2018 Web Conference Companion* (2018), pp. 1–9.

[39] PARKER, J., VAZOU, N., AND HICKS, M. Lweb: information flow security for multi-tier web applications. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 75.

[40] PIERCE, B. C. *Types and programming languages*. MIT press, 2002.

[41] PROJECT, O. W. A. S., AND 73, P. R. P. *OWASP Top 10: The Top 10 Most Critical Web Application Security Threats Enhanced with Text Analytics and Content by PageKicker Robot Phil 73*. CreateSpace Independent Publishing Platform, USA, 2014.

[42] RICE, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society 74*, 2 (1953), 358–366.

[43] SABELFELD, A., AND MYERS, A. C. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.

[44] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE J.Sel. A. Commun. 21*, 1 (Sept. 2006), 5–19.

[45] SANGER, D., AND PERLROTH, N. Cyberattack disrupts printing of major newspapers. `https://www.nytimes.com/2018/12/30/business/media/los-angeles-times-cyberattack.html?rref=collection%2Ftimestopic%2FComputer%20Security%20(Cybersecurity)`, 2018.

[46] SCHMITZ, T., ALGEHED, M., FLANAGAN, C., AND RUSSO, A. Faceted secure multi execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 1617–1634.

[47] SCHOEPE, D., BALLIU, M., PIESSENS, F., AND SABELFELD, A. Let's face it: Faceted values for taint tracking. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I* (2016).

[48] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on* (2010), IEEE, pp. 317–331.

[49] STAICU, C.-A., SCHOEPE, D., BALLIU, M., PRADEL, M., AND SABELFELD, A. An empirical study of information flows in real-world javascript. *arXiv preprint arXiv:1906.11507* (2019).

[50] STATISTA. Annual number of data breaches and exposed records in the united states from 2005 to 2018. `https://tinyurl.com/y387ng4c`, 2019.

[51] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible dynamic information flow control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)* (2011).

[52] TSAI, T. C., RUSSO, A., AND HUGHES, R. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium (CSF '07)* (July 2007).

[53] VASSENA, M., RUSSO, A., BUIRAS, P., AND WAYE, L. Mac a verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming* (2017).

[54] WAYE, L., BUIRAS, P., ARDEN, O., RUSSO, A., AND CHONG, S. Cryptographically secure information flow control on key-value stores. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 1893–1907.

[55] XIE, Y., AND AIKEN, A. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst. 29*, 3 (May 2007).