Thesis for The Degree of Licentiate of Engineering

Securing Software in the Presence of Third-Party Modules

Mohammad M. Ahmadpanah

CHALMERS

Division of Computing Science, Information Security
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2021

Securing Software in the Presence of Third-Party Modules

Mohammad M. Ahmadpanah

ISSN 1652-876X
Department of Computer Science & Engineering
Division of Computing Science, Information Security
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

Abstract

Modular programming is a key concept in software development where the program consists of code modules that are designed and implemented independently. This approach accelerates the development process and enhances scalability of the final product. Modules, however, are often written by third parties, aggravating security concerns such as stealing confidential information, tampering with sensitive data, and executing malicious code.

Trigger-Action Platforms (TAPs) are concrete examples of employing modular programming. Any user can develop TAP applications by connecting trigger and action services, and publish them on public repositories. In the presence of malicious application makers, users cannot trust applications written by third parties, which can threaten users' and platform's security.

We present SandTrap, a novel runtime monitor for JavaScript that can be used to securely integrate third-party applications. SandTrap enforces fine-grained access control policies at the levels of module, API, value, and context. We instantiate SandTrap to IFTTT, Zapier, and Node-RED, three popular JavaScript-driven TAPs, and illustrate how it enforces various policies on a set of benchmarks while incurring a tolerable runtime overhead. We also prove soundness and transparency of the monitoring framework on an essential model of Node-RED.

Furthermore, nontransitive policies have been recently introduced as a natural fit for coarse-grained information-flow control where labels are specified at the level of modules. The flow relation does not need to be transitive, resulting in nonstandard noninterference and enforcement mechanism. We develop a lattice encoding to prove that nontransitive policies can be reduced to classical transitive policies. We also devise a lightweight program transformation that leverages standard flow-sensitive information-flow analyses to enforce nontransitive policies more permissively.

Keywords: Third-Party Modules, Trigger-Action Platforms, JavaScript Runtime Monitor, Nontransitive Noninterference, Information-Flow Control

Acknowledgments

First and foremost, I want to express my appreciation to Andrei, my amazing supervisor, for providing me with this incredible opportunity to work with him, for his persistent guidance, support, kindness, and for constantly offering me new experiences. Thank you for giving me the chance to collaborate with incredible, prestigious researchers whom I've been following for a long time. Our meetings have always been an excellent combination of fun, passion, friendship, and research – I feel blessed to be part of your group!

I am grateful to my co-supervisor, Daniel, for being a huge help and teaching me how to be a cordial collaborator. Thanks for always being encouraging and compassionate!

Big thanks go to Musard and Aslan, the wonderful people who I've been feeling lucky to have as my co-authors. I've learned from you how to mentor a junior and be a true friend with him. I should also thank Eric for being a sharp collaborator in the first months I started my studies. I appreciate the incredible chats with Dave and Wolfgang as well. And special thanks to Storm, our dear guest researcher in the NTNI paper!

None of the invaluable experiences could ever have happened without the full support from my mentor and former supervisor since the first days of my bachelor studies – Thank you, Mehran!

Ivan, Iulia, Benjamin, Alexander, Boel, Irene, Nachi, Elisabet, Andreas, Max, Sandro, Pablo, Fabian, Oskar, Jeff, and other friends in the Computing Science division: Thank you all for the fantastic environment you've made!

Firooz, Amir, Mehrzad, Shirin, Hannah, Fazeleh, and Siavash: having you in the department makes it more pleasant.

My Gothenburg family, Arman, Hamid, and Mohammad: Thank you for helping my adjustment to this new environment go more smoothly and for making me feel here more homey. I particularly appreciate Hamid for every single moment we spent together in the Covid days "discussing" stuff!

Ehsan, Mahmoud, Mina, Mohammad, and Sina: Thank you for always checking in on me; you already know that you are more than friends to me!

Rey: I can't thank you enough for always being there for me, all the time, through thick and thin; I'm so lucky to have you as my best friend. Thank you so much, Rafigh!

Last, but not least, my deepest gratitude goes to my family: Abutorab, Nayereh, Fatemeh, and Hossein – Thank you for everything!

Mohammad M. Ahmadpanah
Gothenburg, Sweden
1 Sep 2021

Contents

Introduction

In software development process, designers always aim at breaking the complexity of the program into smaller building blocks to be able to provide and track the progress of individual features specified in the requirements. This enables developers to implement each block separately and leverage the existing ones if possible, which accelerates the development process, lowers the costs of maintenance, and improves scalability of the program.

Modular programming [19] focuses on logically splitting the functionality of a program into independent yet reusable modules interacting via well-defined interfaces. An interface, or API, is a gate to communicate with the module offering a specific functionality that can be used in parts of a program. Developers can easily load modules and invoke their APIs with desired values in the program. They can also interchange the modules if needed through the development process. Consider a module including a collection of functions to provide network capabilities. Due to the complex nature of network functionalities, the module should be independent of the rest of the program, enabling usage of the module for other applications as well.

As a principle, the complexity of any module should be hidden in the sense that the APIs, along with their documentation, are supposed to be sufficient to realize how to interact with the code. It means that the client program only needs to feed input arguments and handle outputs, without knowing the implementation details.

Modules can be also written in a different language at the core and wrapped to be operable in the developer's language; e.g., `sqlite3` [13] for interaction with sqlite database, `bcrypt` [2] for hashing passwords, and `microtime` [8] for getting the time in microseconds, are Node.js modules

mainly written in C++.

Third-party modules

A *third-party module* is any piece of code, as a unit, written by a third party (i.e., not the program developer) that can be included in the program to add new functionality. Third-party modules typically are distributed and available in application-level package managers like npm [28], Yarn [14], pip [9], and Maven [6], which are public repositories for downloading program dependencies.

Even though modular programming has significant benefits in software development, it poses challenges for security when the code for modules and their APIs is unavailable, written in a different language, or difficult to understand. Naturally, program's security depends on security of the employed modules and how they are applied in the program. Malicious modules (or APIs) and misusing sensitive APIs are the main root causes of vulnerabilities in the programs [36]. Since modules are mostly written by third parties, the concerns get exacerbated more severely.

The security concerns can be divided into three main categories: confidentiality, integrity, and availability. A malicious module can exfiltrate private data from unsuspecting users (an example of threats to confidentiality), alter a sensitive message (an example of integrity violation), and make some undesired delays for the user's application (an example of threats to availability).

Language-based security

Given that the main focus is on application-level policies and mechanisms, we have chosen *language-based approach to security (LBS)* [25, 31, 33]. LBS is a set of techniques for enhancing application security using programming language properties, detecting and preventing vulnerabilities at the language level, not in lower levels such as operating system. An LBS technique consists of (1) a system model (i.e., programming language semantics), (2) a threat model, (3) an enforcement mechanism (e.g., type system and runtime monitor), and (4) the proof for establishing soundness guarantees of the proposed enforcement mechanism.

The assumed system in this thesis is programs using third-party modules and the threat model is malicious module (or application) makers attacking confidentiality and integrity of user's data. In this model, an attacker can develop a third-party module and publish it on public software repositories such as package managers. Program developers and users who require the

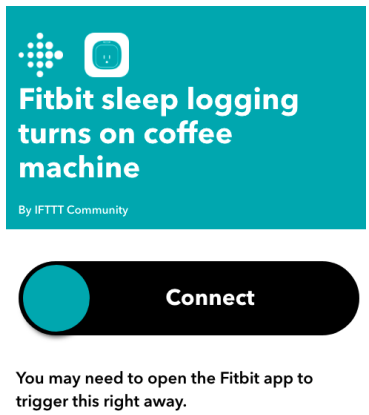
functionality described in the module specification might deploy the malicious module, and therefore, the malicious code can be triggered in program executions, violating policies defined in the system. In another scenario, the attacker may be a legitimate user in the same system running a malicious application that calls a malicious API to modify a shared context, in concurrent with an unsuspecting user's benign program. Lack of proper isolation between applications can be exploited by the attacker and take the innocent user into trouble.

Trigger-action platforms

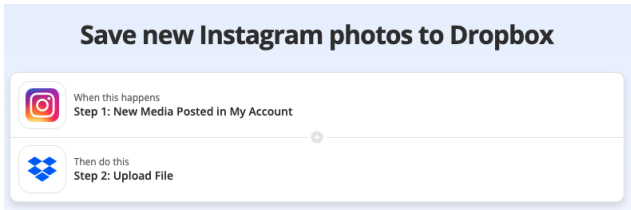
A concrete example of employing modular programming is Trigger-Action Platform (TAP) applications. A TAP links a wide range of otherwise unconnected services and devices, such as IoT and smart devices, social network, healthcare, and cloud services. IFTTT [4], Zapier [35], Node-RED [27], Microsoft Power Automate [7], SmartThings [12], Integromat [5], and Automate.io [1] are in the list of popular TAPs. In a TAP, users can select any trigger and action services they would like to connect, where they only need to know how to set up and relate them to each other using simple interfaces. For instance, as shown in Figure 1, one could deploy and run a new application like getting a hot coffee when you wake up [3], saving new Instagram photos to your Dropbox [10], or monitoring your baby [11], with a few user interactions.

While TAPs enable novel applications across a variety of services, they raise critical security and privacy concerns. A TAP is practically a "person-in-the-middle" between trigger and action services because TAPs often have extensive privileges to act on behalf of the users, for reading, modifying, and deleting a broad range of user's information such as email messages, locations, images, and documents. Attacks to a TAP thus imply compromising the associated trigger and action services. Recently, untrusted TAP that can misuse the rights causing security violations has been studied [20]. Open-source TAPs like Node-RED, with the ability to run on the user's hardware, are alternatives for the users who would like to inspect executions occurring in the TAP. Third-party applications, however, remain a threat to the users', the platform's and the host system's security. The fact that most TAP applications are developed by third-party application makers [18] intensifies the security risks, even if the platform and applications are open-source.

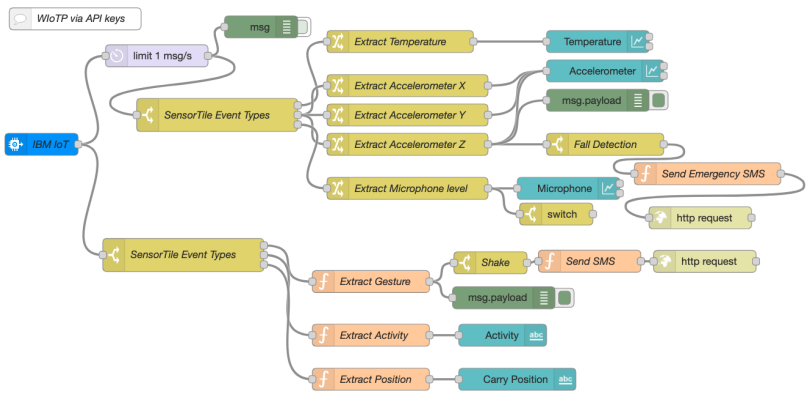
Our threat model indeed fits in TAP ecosystems. Many of TAP applications deployed by users are third-party due to ease of use, especially when the knowledge of programming is not presumed for the end users of TAPs. Except for a small portion of official applications developed by the platform



(a)



(b)



(c)

Figure 1: An example of applications in: (a) IFTTT [3]; (b) Zapier [10]; (c) Node-RED [11].

itself (which can still be considered as third-party), applications written by third parties can execute any code in the user's context. In some of TAPs, multiple users are running applications concurrently at the same time and

if the platform's isolation has some breakouts, the attacker might break into other user's context.

Security policies

In order to secure programs in the presence of third-party modules and applications, this thesis aims at the most prevalent policies for confidentiality (and integrity), i.e., Access Control (AC) [29, 32] and Information-Flow Control (IFC) [24, 31]. An access control policy specifies the list of permitted access requests from an authenticated user, and information-flow control tracks the information propagation during executions of a program. These two policies are complementary since access control cannot prohibit revealing sensitive data after being granted.

AC and IFC are vital when it comes to TAP's security. An application in TAPs like IFTTT and Zapier usually includes one trigger service and possibly more than one action service. The code between these two parties (named *filter code* in IFTTT and *Zap code* in Zapier) may divulge a secret, alter a sensitive message, or make unintended delays for the application, *via API calls*. Monitoring each API call through the execution seems to be a promising approach to enforce access control policies.

In Node-RED, users can deploy applications or single modules in their applications, represented by a graphical user interface (see Figure 1c). Thus, the wiring between modules (or *nodes*) can be considered as the user intended policy. The security of each node still remains a problem; monitoring the local execution of each of them would be sufficient for enforcing the access control policies. Therefore, by isolating nodes and verifying API calls according to nodes' policies, security of the platform and users is guaranteed.

IFC in JavaScript-driven TAPs like IFTTT and Node-RED has been studied in the literature [18, 34], resulting in tools based on infomation-flow trackers like JSFlow [23]. While these techniques detect and prevent information flow violations, lack of proper isolation between applications and modules leads to major security issues. Violations such as leakage of sensitive Dropbox URL links [18], retrieving the recent quakes from a fake website instead of the official one [17], and taking over the entire system [17] are instances of the attacks to improper isolation mechanism. Note that in some TAPs like IFTTT, users share the execution environment with a few others, which also introduces security breaches to the other users.

With the goal to strike the balance between security and functionality for JavaScript-driven TAPs, we devise a sound and transparent monitoring framework to enforce access control policies at the levels of module, API, value, and context. The proposed security mechanism is practical in the sense

that the runtime overhead is tolerable. The first two papers introduce a principled framework to sandbox JavaScript-driven TAPs that enforces access control policies in the presence of third-party modules.

Nontransitive policies

In another research track on programs with third-party modules, nontransitive policies have been recently introduced [26] as a natural fit for coarse-grained information-flow control where labels are specified at the level of modules. In a transitive IFC system [21, 22, 30], which is the classical model, security levels constitute a partially ordered set and information may flow to all higher security levels, i.e., elements of the transitive closure of the flow relation defined by the policy. In this setting, expressing coarse-grained security requirements, especially between untrusted modules, seems to be difficult.

For example, module A may trust only module B, but the transitive relation indirectly propagates the trust relation further to the modules that the module B also trusts – which is undesired for the module A. Mutual and circular information flows are also ruled out; modules A and B cannot send information to each other unless both are at the same security level. They must share the same flow relations to any other modules, which might not always be the case.

Untrusted code typically is executed in the same process, together with sensitive trusted system code. Instead of trusting third-party-provided security policies, application developers must specify security policies to protect sensitive system modules from untrusted code. The need for more flexible IFC policy language motivates designing flow relations that are not necessarily transitive, which is in contrast to the classical notion of security [21].

To support module-level coarse-grained information-flow policies, Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) [26] have been suggested as a new security condition and enforcement. The third paper demonstrates how a nonstandard information flow policy, suitable to reasoning at the level of modules of a program, can be reduced to a standard information policy, leveraging the standard type system to enforce the IFC policy. The immediate result of this reduction is that nontransitive policies, which are expressive enough to specify IFC policies for systems with third-party code, can be enforced by the existing mechanisms for classical transitive noninterference.

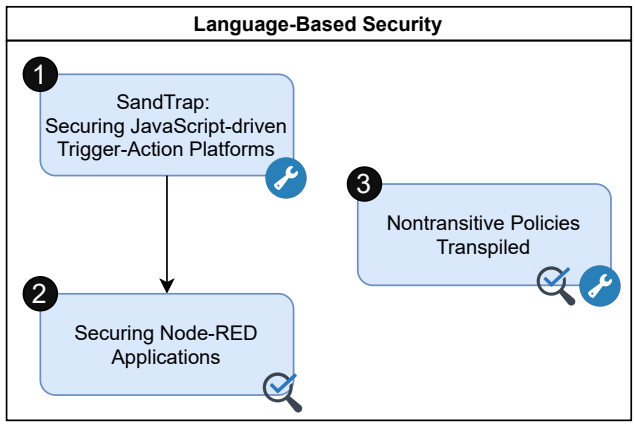


Figure 2: The relationship between papers included in the thesis; the badges indicate that the paper introduces a tool, provides formal guarantees, or both.

Thesis structure

Figure 2 schematically demonstrates the relationship between the papers included in the thesis. As depicted, this thesis contributes to both theoretical and practical aspects of language-based security. Paper 1 introduces a tool for monitoring JavaScript programs and Paper 2 takes a step towards formalizing the monitor. Paper 3 goes from theory to practice, leading up to the transpiler tool.

Paper 1: SandTrap: Securing JavaScript-driven Trigger-Action Platforms [17]

This paper presents a security analysis of JavaScript-driven Trigger-Action Platforms (TAPs), with our findings on identifying exploitable vulnerabilities in the popular platforms, i.e., IFTTT, Zapier, and Node-RED. We demonstrate critical exploitable vulnerabilities in the platforms and discuss their impacts, e.g., by performing an empirical study on the Node-RED ecosystem. To tackle the root of the security problems, we propose SandTrap, a secure yet flexible monitor for JavaScript in the presence of Node.js modules. It supports fine-grained module-, API-, value-, and context-level policies and facilitates their generation. SandTrap advances the state of the art in JavaScript sandboxing by a novel approach that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. We show how SandTrap can secure IFTTT, Zapier, and Node-RED applications with tolerable performance overhead, as evidence

for the utility of the monitor.

Statement of contributions This paper was in collaboration with Daniel Hedin, Musard Balliu, Eric Olsson, and Andrei Sabelfeld. I found some of the vulnerabilities in Node-RED and helped my co-authors to identify them, where we came up with the idea of sandboxing nodes. I was also in charge of instantiating SandTrap to IFTTT, Zapier, and Node-RED. I designed and implemented the case studies, where I reported the evaluation results for secure and insecure applications.

Appeared in: Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)

Paper 2: Securing Node-RED Applications [16]

This paper expands on the recently-discovered critical exploitable vulnerabilities by misusing sensitive APIs within nodes in Node-RED, where the impact ranges from massive exfiltration of data from unsuspecting users to taking over the entire platform. Motivated by the need for an access control mechanism for Node-RED, we propose an essential model of the platform, suitable to reason about nodes and flows. This paper presents a principled framework to enforce fine-grained API control for untrusted Node-RED applications by local access checks that support module-, API-, value-, and context-level policies. We prove soundness and transparency of the monitor.

Statement of contributions This paper was in collaboration with Musard Balliu, Daniel Hedin, Eric Olsson, and Andrei Sabelfeld. As a step towards proving correctness guarantees for SandTrap [17], with the help of my co-authors, I presented the formal models of the monitor and Node-RED. I proved that the enforcement mechanism is sound and transparent for an essential model of Node-RED.

Appeared in: Protocols, Logic, and Strands: Festschrift in honor of Joshua Guttman'21

Paper 3: Nontransitive Policies Transpiled [15]

This paper demonstrates that despite the different aims and intuitions of nontransitive policies compared to classical transitive policies, nontransitive noninterference (NTNI) can in fact be reduced to classical transitive noninterference (TNI). On the security characterization side, we show that NTNI

corresponds to classical noninterference on a lattice that records source-to-sink relations derived from nontransitive policies. On the enforcement side, we devise a lightweight program transformation that enables us to leverage standard flow-sensitive information-flow analyses to enforce nontransitive policies. Further, we improve the permissiveness over the nonstandard nontransitive type enforcement while retaining the soundness. An immediate practical benefit of our work is the implication that we can leverage state-of-the-art flow-sensitive information-flow tools, which we demonstrate by utilizing JOANA to enforce nontransitive policies for Java programs.

Statement of contributions This paper was in collaboration with Aslan Askarov and Andrei Sabelfeld. I was responsible for formalizing and proving the idea of transpiling NTNI to classical TNI, for programs with or without I/O. I also implemented a prototype of the transpiler for Java programs and performed the case studies.

Appeared in: Proceedings of the 6th IEEE European Symposium on Security and Privacy (EuroS&P'21)

Bibliography

[1] automate.io. https://automate.io/, 2021.

[2] bcrypt. https://www.npmjs.com/package/bcrypt, 2021.

[3] Fitbit sleep logging turns on coffee machine. https://ifttt.com/applets/236859p-fitbit-sleep-logging-turns-on-coffee-machine, 2021.

[4] IFTTT: If This Then That. https://ifttt.com/, 2021.

[5] Integromat. https://www.integromat.com/, 2021.

[6] Maven. https://maven.apache.org/, 2021.

[7] Microsoft Power Automate. https://powerautomate.microsoft.com/, 2021.

[8] microtime. https://www.npmjs.com/package/microtime, 2021.

[9] pip. https://pypi.org/project/pip/, 2021.

[10] Save new instagram photos to dropbox. https://zapier.com/apps/dropbox/integrations/instagram/197/save-new-instagram-photos-to-dropbox, 2021.

[11] Smart button baby monitor. https://www.hackster.io/Fan/smart-button-baby-monitor-a03a90, 2021.

[12] SmartThings. https://www.smartthings.com/, 2021.

[13] sqlite3. https://www.npmjs.com/package/sqlite3, 2021.

[14] Yarn. https://yarnpkg.com/, 2021.

[15] M. M. Ahmadpanah, A. Askarov, and A. Sabelfeld. Nontransitive policies transpiled. In *EuroS&P*, 2021.

[16] M. M. Ahmadpanah, M. Balliu, D. Hedin, L. E. Olsson, and A. Sabelfeld. Securing Node-RED applications. In *Protocols, Logic, and Strands: Festschrift in honor of Joshua Guttman*, 2021.

[17] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven trigger-action platforms. In *USENIX Security*, 2021.

[18] I. Bastys, M. Balliu, and A. Sabelfeld. If this then what? controlling flows in IoT apps. In *CCS*, 2018.

[19] K. L. Busbee and D. Braunschweig. *Programming Fundamentals: A Modular Structured Approach*. 2018.

[20] Y. Chen, A. R. Chowdhury, R. Wang, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Data privacy in trigger-action IoT systems. In *S&P*, 2021.

[21] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.

[22] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE S&P*, 1982.

[23] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*, 2014.

[24] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*. 2012.

[25] D. Kozen. Language-based security. In *MFCS*, 1999.

[26] Y. Lu and C. Zhang. Nontransitive security types for coarse-grained information flow control. In *CSF*, 2020.

[27] Node-RED. `https://nodered.org/`, 2021.

[28] npm. Node Package Manager. `https://www.npmjs.com/`, 2021.

[29] J. Qiu, Z. Tian, C. Du, Q. Zuo, S. Su, and B. Fang. A survey on access control in the age of Internet of Things. *IEEE Internet Things J.*, 2020.

[30] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory Menlo Park, 1992.

[31] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 2003.

[32] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD*, 2000.

[33] F. B. Schneider, J. G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics*, 2001.

[34] D. Schreckling, J. D. Parra, C. Doukas, and J. Posegga. Data-centric security for the IoT. In *IoT 360 (2)*, 2015.

[35] Zapier. `https://zapier.com/`, 2021.

[36] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security*, 2019.

1

SandTrap: Securing JavaScript-driven Trigger-Action Platforms

Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld

USENIX Security'21

Abstract. Trigger-Action Platforms (TAPs) seamlessly connect a wide variety of otherwise unconnected devices and services, ranging from IoT devices to cloud services and social networks. TAPs raise critical security and privacy concerns because a TAP is effectively a "person-in-the-middle" between trigger and action services. Third-party code, routinely deployed as "apps" on TAPs, further exacerbates these concerns. This paper focuses on JavaScript-driven TAPs. We show that the popular IFTTT and Zapier platforms and an open-source alternative Node-RED are susceptible to attacks ranging from exfiltrating data from unsuspecting users to taking over the entire platform. We report on the changes by the platforms in response to our findings and present an empirical study to assess the implications for Node-RED. Motivated by the need for a secure yet flexible way to integrate third-party JavaScript apps, we propose SandTrap, a novel JavaScript monitor that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. To aid developers, SandTrap includes a policy generation mechanism. We instantiate SandTrap to IFTTT, Zapier, and Node-RED and illustrate on a set of benchmarks how SandTrap enforces a variety of policies while incurring a tolerable runtime overhead.

1 Introduction

Trigger-Action Platforms (TAPs) seamlessly connect a wide variety of otherwise unconnected devices and services, ranging from IoT devices to cloud services and social networks. TAPs like IFTTT [30], Zapier [74], and Node-RED [48], allow users to run trigger-action *apps* (or *flows*). Upon a *trigger*, the app performs an *action*, such as "Get an email when your EZVIZ camera senses motion", "Save new Instagram photos to Dropbox", and control "a thermostat which can switch a heater on or off depending on temperature". IFTTT's 18 million users run more than a billion apps a month connected to more than 650 partner services [38].

JavaScript is a popular language for both apps and their integration in TAPs. IFTTT enables app makers to write so-called *filter* code, JavaScript to customize the trigger and action ingredients, while Zapier offers so-called *code steps* in JavaScript. For IFTTT's camera-to-email app, the filter code might, for example, skip the action during certain hours. Both IFTTT and Zapier utilize serverless computing to run the JavaScript apps with Node.js on AWS Lambda [4]. Node-RED is also built on top of Node.js, allowing JavaScript packages from third parties. For third-party code, Zapier and Node-RED adopt a single-user integration (Figure 1(a)), with a separate Node.js instance for each user. In contrast, IFTTT utilizes a multi-user integration (Figure 1(b)) where a Node.js instance is reused to process filter code from multiple users. Instance reuse implies reducing the need for an expensive *cold start*, when a function is provisioned with a new container. IFTTT's choice of reusing instances thus implies reducing costs under AWS' economic model [4]. As we will see, the security implications of this choice require great care.

TAP security and privacy challenges TAPs enable novel applications across a variety of services. Yet TAPs raise critical security and privacy concerns because a TAP is effectively a "person-in-the-middle" between trigger and action services. TAPs often rely on OAuth-based access delegation tokens that give them extensive privileges to act on behalf of the users [22]. Compromising a TAP thus implies compromising the associated trigger and

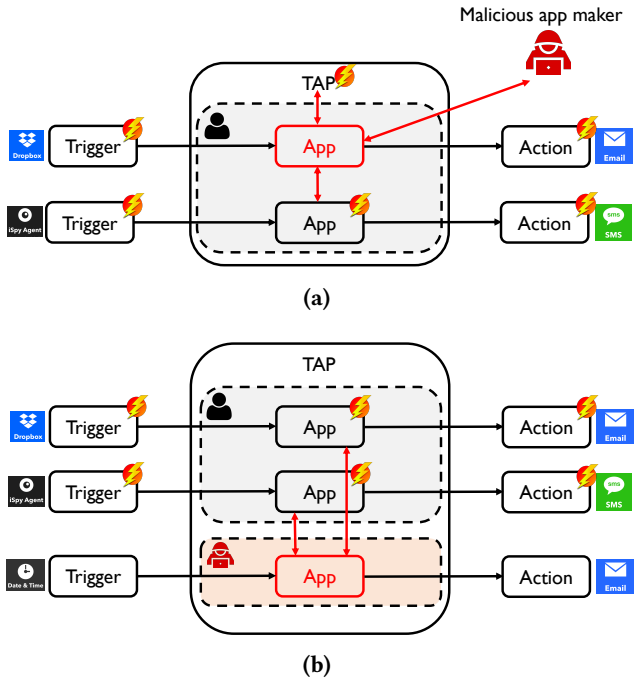


Figure 1: Threat model of a malicious app maker: (a) Victim with a malicious app; (b) Victim with only benign apps.

action services.

TAPs thrive on the model of end-user programming [68]. The fact that most TAP apps are by third-party app makers [8] exacerbates security risks. Wary of these concerns, Gmail recently removed their IFTTT triggers [27]. On the other hand, running the Node-RED platform, on one's own hardware with inspectable open-source code, makes trust to an external platform unnecessary. Third-party apps, however, remain a threat not only to the users' data accessible to these apps but to the entire system's security.

Threat model Figure 1 illustrates our threat model: a malicious app (in red) attacking the confidentiality and integrity of user data. While we touch upon some forms of availability (e.g., when the integrity of action data ensures the associated device is enabled), availability is not the main focus of this work. Indeed, effective approaches to mitigating typical denial-of-service attacks are already in use, such as timing out on filter code execution and request-rate limiting [29].

Under the first attack scenario (Figure 1(a)), the user is tricked into installing a malicious app. This scenario applies to both single- and multi-user

architectures, including all of IFTTT, Zapier, and Node-RED. In IFTTT, the filter code is not inspectable to ordinary users, making it impossible for the users to determine whether the app is malicious. Further, IFTTT does not notify the users when apps are updated. The app might thus be benign upon installation and subsequently updated with malicious content. In this scenario, the attacker aims at compromising the confidentiality of the trigger data or the integrity of the action data. For example, a popular third-party app like "Automatically back up your new iOS photos to Google Drive" can become malicious and leak the photos to the attacker unnoticeably to the user. Further, the attacker targets compromising the confidentiality of the trigger data or the integrity of the action data of *other apps* installed by the user. Finally, the attacker may also target compromising the TAP itself, for example, gaining access to the file system.

Under the second attack scenario (Figure 1(b)), the user has only benign apps installed. This scenario applies to the multi-user architecture, as in IFTTT. The attacker compromises the isolation boundary between apps and violates the confidentiality of the trigger data or the integrity of the action data of other apps installed by *other users*. This is a dangerous scenario because any app user on the platform is a victim.

This leads to our first set of research questions: *Are the popular TAPs secure with respect to integrating third-party JavaScript apps? If not, what are the implications?*

TAP vulnerabilities To answer these questions, we show that the popular IFTTT and Zapier platforms, as well as an open-source alternative Node-RED, are susceptible to a variety of attacks. We demonstrate how an attacker can exfiltrate data from unsuspecting IFTTT users. We show how different apps of the same Zapier user can steal information from each other and how malicious Node-RED apps can compromise other components and take over the entire platform. We report on the changes made by IFTTT and Zapier in response to our findings. Both are proprietary closed platforms, restricting possibilities of empirical studies with the app code they host. On the other hand, Node-RED is an open-source platform, enabling us to present an empirical study of the security implications for the published apps.

The versatility and impact of these exploitable vulnerabilities indicate that these vulnerabilities are not merely implementation issues but instances of a fundamental problem of securing JavaScript-driven TAPs.

SandTrap This motivates the need for a secure yet flexible way to integrate third-party apps. A *secure* way means restricting the code. How do we limit third-party code to the *least privileges* [61] it should have as a component of an app? A *flexible* way means that some apps need to be fully isolated

at the module level, while others need to interact with some modules but only through selected APIs. Some interaction through APIs can be value-sensitive, for example, when allowing an app to make HTTPS requests to specific trusted domains. Finally, TAPs like Node-RED make use of both message passing and the shared context [51] to exchange information between app components, and both types of exchange need to be secured. While flexibility is essential, it must not come at the price of overwhelming the developers with policy annotations. This leads us to our second set of research questions: *How to represent and enforce fine-grained policies on third-party apps in TAPs? How to aid developers in generating these policies?*

Addressing these questions, we present SandTrap, a novel JavaScript monitor that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes [66, 67] to enforce fine-grained access control policies. To aid developers in designing the policies, SandTrap offers a simple policy generation mechanism enabling both (i) *baseline* policies that require no involvement from app developers or users (once and for all apps per platform) and (ii) *advanced* policies customized by developers or users to express fine-grained app-specific security goals. We instantiate SandTrap to IFTTT, Zapier, and Node-RED and illustrate on a set of benchmarks how to enforce a variety of policies while incurring a tolerable runtime overhead.

Contributions In summary, the paper offers the following contributions:

- We demonstrate that the popular TAPs IFTTT and Zapier are susceptible to attacks by malicious JavaScript apps to exfiltrate data of unsuspecting users. We report on the changes by the platforms (Section 3).
- We present vulnerabilities on Node-RED along with an empirical study that estimates their impact (Section 4).
- We present SandTrap, a novel structural JavaScript monitor that enforces fine-grained access control policies (Section 5).
- We evaluate the security and performance of SandTrap for IFTTT, Zapier, and Node-RED (Section 6).

2 Background

We give a brief background on IFTTT, Zapier, and Node-RED, consolidated in Table 1. IFTTT and Zapier are commercial platforms with cloud-based app stores, while Node-RED is an open-source platform, suitable for both local and cloud installations, intended for a single user per installation. Node-RED has a web-based app store for apps (flows) and their components (packages).

IFTTT and Node-RED allow direct app publishing, with no review. While Zapier and Node-RED allow the full power of JavaScript and Node.js APIs and

modules, IFTTT is more restrictive. IFTTT's third-party apps can be written in TypeScript [40], a syntactical superset of JavaScript. The filter code of the apps must be free of direct accesses to the global object, APIs (other than those to access the trigger and action ingredients), I/O, or modules. Some of these checks, like restricting access to APIs and allowing no modules, are enforced statically at the time of installation. Other checks are enforced at runtime. Some of these checks, like the runtime check of allowing no code to be dynamically generated from strings, were introduced after our reports from Section 3.

Both IFTTT and Zapier utilize AWS Lambda [4] for running the JavaScript code of the apps. Once an event is triggered to fire an app, AWS Lambda's function handler in Node.js evaluates the JavaScript code of the app in the context of the parameters associated with the trigger and action services. Lambda functions are computed by Node.js instances, where each instance is a process in a container running Amazon's version of the Linux operating system. Node.js code inside AWS Lambdas may generally use APIs for file and network access. By default, file access is read-only, with the exception of writes to the temporary directory.

When a victim is tricked into installing a malicious app (Figure 1(a)), the malicious app targets the data that the app has access to, which applies to all platforms. The other threats occur even if the victim only has benign apps (Figure 1(b)). Because IFTTT's architecture is multi-user, a malicious app may compromise the data of all other users and apps. Zapier's architecture is single-user with container-based isolation provided by AWS Lambda. This reduces the attack targets to the other apps of the same user. Although Node-RED's architecture is single-user, its local installation opens up for attacking both the other apps of the same user and the entire platform.

The differences in these TAPs motivate the need for a versatile security policy framework, which we design and evaluate in Sections 5 and 6, respectively.

3 IFTTT and Zapier vulnerabilities

This section presents vulnerabilities in IFTTT and Zapier and the reaction of the vendors to address them.

3.1 IFTTT sandbox breakout

IFTTT apps use filter code to customize the app's ingredients (e.g., adjust lights as it gets darker outside) or to skip an action upon a condition (e.g.,

Platform	Distribution	Language	Threats by malicious app maker		Policy		
					Platform provider	App provider	User
IFTTT	Proprietary Cloud installation App store and own apps	TypeScript No dynamic code evaluation, No modules, No APIs or I/O, No direct access to the global object	Compromise data of the installed app	Compromise data of other users and apps	Baseline policy for platform to handle actions and triggers	Value-based parameterized policies for actions and triggers	Instantiation of combined parameterized policies
Zapier		JavaScript Node.js APIs Node.js modules		Compromise data of other apps of the same user	Baseline policy for platform, node-fetch, StoreClient and common modules	Value-based parameterized policies for modules	
Node-RED	Open-source Local and cloud installation App store and own apps			Compromise data of other apps of the same user and the entire platform	Baseline policy for platform, built-in nodes and common modules	Value-based parameterized policies for modules including other nodes	

Table 1: TAPs in comparison.

logging location status only during working hours). Filter code has access to the sensitive data of the associated trigger and action services. For example, the filter code of an app with the trigger "New Dropbox file" has access to the file via the `Dropbox.newFileInFolder.FileUrl` API.

According to IFTTT's documentation, "filter code is run in an isolated environment with a short timeout. There are no methods available that do any I/O (blocking or otherwise)..." [29]. To achieve this isolation, IFTTT runs a combination of static and dynamic security checks mentioned in Section 2, restricting filter code to only accessing the APIs that pertain to the triggers and actions of a given app. For example, an app with an email action can set the body of an email by `Email.sendMeEmail.setBody()` but may not use I/O or global methods like `setTimeout()`.

Unfortunately, it is possible to break out of the sandbox. We create a series of proof-of-concepts (PoCs) that break out of the increasingly hardened sandboxes.

PoC v1 The PoC follows the steps outlined below:

- Make a private app and activate it on IFTTT. The trigger and action services are unimportant as long as it is easy for the attacker to trigger the app. For example, a `Webhook` trigger is fired on a GET request to IFTTT's webhook URL.
- Evade the static security check in IFTTT's web interface for filter code by using `eval`.
- As the filter code is dynamically evaluated by the Lambda function, utilize the filter code to import the AWS Lambda runtime module and poison [36, 37] the prototype of one of the runtime classes: `rapid.prototype.nextInvocation` located in `/var/runtime/RAPIDClient.js`. The poisoning relies on the module caching of `require`, ensuring that the imported runtime is the same instance as the one used by AWS Lambda.
- The poisoning allows collecting data between invocations of filter code. What makes this vulnerability critical is that Node.js instances are kept alive for up to 30 minutes in order to process filter code from arbitrary apps/users. This means that the attacker can collect all future requests and responses for unsuspecting users and apps on the same Node.js instance for up to 30 minutes and then simply re-trigger the malicious app for continuous exfiltration.
- Send the collected data to a server under the attacker's control using `https.request`. We confirm successful exfiltration of mock data on a test clone of IFTTT's Lambda function deployed in AWS Lambda.
- While poisoning the prototype of `rapid.prototype.nextInvocation`, our PoC preserves its functionality, making the exfiltration of information invisible

to the users.

Impact The impact is substantial because it affects all IFTTT apps with filter code, while the attacker does not need any user interaction in order to leak private data. Filter code is a popular feature enabling "flexibility and power" [29]. While there are active forum discussions on filter code [59], IFTTT is a closed platform with no information about the extent to which filter code is used. Furthermore, it is invisible to ordinary users if the apps they have installed contain filter code. Thus, any app with access to sensitive data may be vulnerable. Bastys et al. [8] estimate 35% of IFTTT's apps have access to private data via sensitive triggers, accessing such data as images, videos, SMSes, emails, contact numbers, voice commands, and GPS locations.

Note that this vulnerability can also be exploited to compromise the integrity and availability of action data. While these attacks are generally harder to hide, sensitive actions are prevalent. Bastys et al. [8] estimate 98% of IFTTT's apps to use sensitive actions.

PoC v2 IFTTT promptly acknowledged a "critical" vulnerability and deployed a patch in a matter of days. The patch hardened the check on filter code, disallowing `eval` and `Function`, ensuring that `require` was not available as a function in the TypeScript type system and locking down network access for the Lambda function.

This leads us to a more complex PoC to achieve exfiltration with the same attacker capabilities. The challenge is to get hold of `require` in the face of TypeScript's type system and disabled `eval`. We create an app with functionality to notify of a new Dropbox file by email. Our filter code implements the additional attack steps as follows:

```
declare var require : any;
var payload = `try { ...
  let rapid = require("/var/runtime/RAPIDClient.js");
  // prototype poisoning of rapid.prototype.nextInvocation
  ... }` ;
var f = (() => {}).constructor.call(null,'require', 'Dropbox', 'Meta',
    payload);
var result = f(require, Dropbox, Meta);
Email.sendMeEmail.setBody(result);
```

The essential idea is to (i) bypass TypeScript's type system and reintroduce `require` via a declaration, since it is present in the JavaScript runtime, (ii) use the function constructor while bypassing the `Function` filter passing in `require`, since functions created this way live in the global context where `require` is not available, and (iii) use network capabilities of the malicious app to do the exfiltration, rather than the network capabilities of the lambda function itself. We can thus package exfiltration messages with the sensitive

information of IFTTT users in the body of the email to the attacker by setting `Email.sendMeEmail.setBody(result)`.

PoC v3 In line with our recommendations to introduce JavaScript-level sandboxing, IFTTT introduced basic sandboxing on filter code. Filter code is now run inside of `vm2` [63] sandbox. However, as we will see throughout the paper, as soon as there is some interaction between the host and the sandbox, there is potential for vulnerabilities. This leads us to our final PoC. Our starting point is the observation that filter code is allowed to use Moment Timezone [44] APIs for displaying user and app triggering time in different timezones [29]. To make these APIs accessible, `Meta.currentUserTime` and `Meta.triggerTime` objects, created outside the sandbox, are passed to the filter code inside the sandbox. Our PoC v3 poisons the prototype of the `tz` method of the `moment` prototype. This allows the attacker to arbitrarily modify `Meta.currentUserTime` and `Meta.triggerTime` for other apps, which is critical for apps whose filter code is conditional on time [28]. Thus, the attacker gains control over whether to run or skip actions in other users' apps.

As a short-term patch, `vm2`'s `freeze` [63] method patches the problem by making `moment` prototype read-only. However, while this patch prevents prototype poisoning of the `moment` objects, it does not scale to attacks at other levels of abstraction. For example, *URL attacks* by Bastys et al. [8] on a user who installs a malicious app (Figure 1(a)) allow the attacker exfiltrating secrets by manipulating URLs. An IFTTT app that backs up a Dropbox file on Google Drive may thus leak the file to the attacker by setting the Google Drive upload URL to `"https://attacker.com/log?"+ encodeURIComponent(Dropbox.newFileInFolder.FileUrl)` instead of `Dropbox.newFileInFolder.FileUrl`.

We learn two key lessons from these vulnerabilities. First, the problem of secure JavaScript integration on TAPs is not merely a technical issue but a larger fundamental problem. Already on IFTTT, it is hard to get it right and we will see further complexity for Zapier and Node-RED. Second, these attacks motivate the need for enforcing (i) a *baseline* security policy for all apps on the platform and (ii) *advanced* app-specific policies. In particular, there is need for fine-grained access control at *module-level* (to restrict access to Node.js modules, for all apps), *API-level* (to only allow access to trigger and action APIs and only read access to `Meta.currentUserTime` and `Meta.triggerTime`, for all apps) and *value-level* (to prevent attacks like URL manipulation, for specific apps).

Coordinated disclosure We had continuous interactions with IFTTT's security team through the course of discovering, reporting, and fixing the vulnerabilities. Our first report already suggested proxy-based sandboxing as

a countermeasure, which is what IFTTT ultimately settled for. After each patch, IFTTT's security team reached back to us asking to verify it. We received bounties acknowledging our contributions to IFTTT's security.

3.2 Zapier sandbox breakout

In the interest of space, we keep this section brief and focus on the differences between Zapier and IFTTT. One difference is that it is currently not possible to publish *zaps* (Zappier apps) with code steps for other users. However, scenarios when a user copies malicious JavaScript from forums are realistic [24]. In contrast to IFTTT, Zapier allows fully-fledged JavaScript in zaps with file system (`fs`) and network communication (`http`) modules enabled by default. Another difference is in the use of AWS Lambda runtimes. Zapier's lambda functions are not shared across users. However, we discover that the same Lambda function sometimes runs code steps of *different zaps* of the same user (Figure 1(a)).

PoC We demonstrate the vulnerability by the following PoC. One zap is benign: it sends an email notification whenever there is a new Dropbox file and uses a code step to include the size of the file in the email body. The other zap is malicious: it has no access to Dropbox and yet it exfiltrates the data (including the content of any new Dropbox files) to the attacker. We demonstrate the attack on our own test account, involving no other users.

Impact Because Lambda functions are not shared among users, the impact is somewhat reduced. Nevertheless, these attacks can become more impactful if Zapier decides to allow users sharing zaps with JavaScript. Zapier confirmed that they reuse execution sandboxes per user per language and acknowledged that our PoC exposed unintended behavior. This led to identifying a bug in the way they handle caching in their Node.js integration.

This vulnerability further motivates the need for *fine-grained access control at module-, API-, and value-levels*. Compared to IFTTT, module- and API-level policies are particularly interesting here because of the more liberal choices of what code to allow in Zapier's code steps. Similar to IFTTT, it is natural to divide the desired policies into a *baseline* policy for all zaps that protects the platform's sandbox and *advanced* zap-specific policies that protect zap-specific data.

Coordinated disclosure Zapier was also quick in our interactions. We received a bounty acknowledging our contributions to Zapier's security.

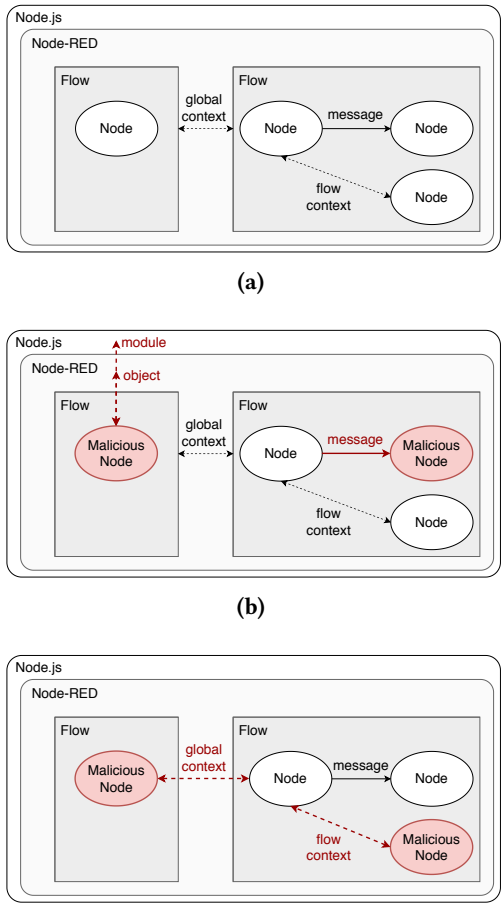


Figure 2: (a) Node-RED architecture; (b) Isolation vulnerabilities; (c) Context vulnerabilities.

4 Node-RED vulnerabilities

Node-RED is "a programming tool for wiring together hardware devices, APIs and online services" [48]. We overview the key components of Node-RED (Section 4.1) and identify two types of vulnerabilities that malicious app makers can exploit: platform-level isolation vulnerabilities (Section 4.2) and application-level context vulnerabilities (Section 4.3). We perform empirical evaluations on a dataset of official and third-party Node-RED packages to study the implications of exploiting these vulnerabilities. We characterize the impact of malicious apps by studying code dependencies and by a security labeling of sources and sinks of Node-RED nodes. We also

study the prevalence of vulnerable apps that expose sensitive information to other Node-RED components via the shared context. We find that more than 70% of Node-RED apps are capable of privacy attacks and more than 76% of integrity attacks. We also identify several concerning vulnerabilities that can be exploited via the shared context.

4.1 Node-RED platform

Figure 2a depicts the Node-RED architecture consisting of a collection of apps, called *flows*, connecting components called *nodes*. The Node-RED runtime (built on Node.js) can run multiple flows enabling not only the direct exchange of messages within a flow, but also indirect inter-flow and inter-node communication via the *global* and the *flow* context [51].

Nodes are reactive Node.js applications that may perform side-effectful computations upon receiving messages on at most one input port (dubbed *source*) and send the results potentially on multiple output ports (dubbed *sinks*). The three main types of Node-RED nodes are *input* (containing no sources), *output* (containing no sinks), and *intermediary* (containing both sources and sinks). Moreover, Node-RED uses *configuration* nodes (containing neither sources nor sinks) to share configuration data, such as login credentials, between multiple nodes.

Flows are JSON files wiring node sinks to node sources in a graph of nodes. End users can either configure and deploy their own flows on the platform's environment or use existing flows provided by the official Node-RED catalog [47] and by third-parties [52]. Figure 3 shows a flow that retrieves earthquake data for logging and notifying the user whenever the magnitude exceeds a threshold. To facilitate end-user programming [68], flows can be shown visually via a graphical user interface and deployed in a push-button fashion.

Contexts provide a way to store information shared between different nodes without using the explicit messages that pass through a flow [51]. For example, a sensor node may regularly publish new values in one flow, while another flow may return the most recent value via HTTP. By storing the

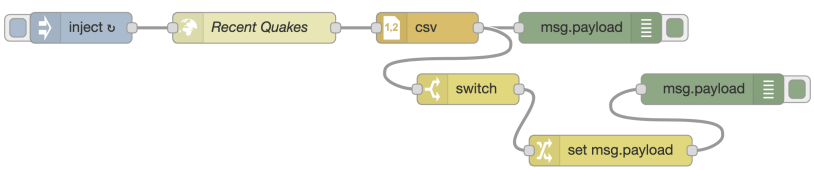


Figure 3: Earthquake notification and logging.

sensor reading in the shared context, it makes the data available for the HTTP flow to return. Node-RED restricts access to the context at three levels: (i) *Node*, only visible to the node that sets the value, (ii) *Flow*, visible to all nodes on the same flow, and (iii) *Global*, visible to all nodes on any flow.

Node-RED security relies on deployment on a trusted network ensuring that the users' sensitive data is processed in a user-controlled environment, and on authentication mechanisms to control access to nodes and wires [49]. Further, the official node `Function` runs the code provided by the user in a `vm` sandbox [54]. However, `Function` nodes are not suitable for running untrusted code because `vm`'s sandbox "is not a security mechanism" [54], and, unsurprisingly, there are straightforward breakouts [32].

We present Node-RED attacks and vulnerabilities that motivate a *baseline* policy to protect the platform and *advanced* flow- and node-specific policies at different granularity levels.

4.2 Platform-level isolation vulnerabilities

Unfortunately, Node-RED is susceptible to attacks by malicious node makers due to insufficient restrictions on nodes. Attackers may develop and publish nodes with full access to the APIs provided by the underlying runtimes, Node-RED and Node.js, as well as the incoming messages within a flow. Figure 2b illustrates the different attack scenarios for malicious nodes. At the Node.js level, an attacker can create a malicious Node-RED node including powerful Node.js libraries like `child_process`, allowing the attacker to execute arbitrary commands and take full control of the user's system [56]. Restricting library access is challenging in Node-RED because attackers can exploit trust propagation due to transitive dependencies in Node.js [58, 75], while at the same time access to a sensitive library like `child_process` is necessary for the functionality of Node-RED.

At the platform level, `RED` [50], the main object in the Node-RED structure, is also vulnerable. A malicious node can manipulate the `RED` object to abort the server (e.g., `RED.server._events = null`) or introduce a covert channel shared between multiple instances of a node in different flows (e.g., by adding new properties to the `RED` object like `RED.dummy`). These attacks motivate the need for a platform-level baseline policy of *access control at the level of modules and shared objects*.

Moreover, application-specific attacks call for advanced security goals and thus advanced policies. If a malicious node is used within a sensitive flow, it may read and modify sensitive data by manipulating incoming messages. For example, a malicious email node can forward a copy of the email text to an attacker's address in addition to the original recipient. The benign

code sets the sending options `sendopts.to` to contain only the address of the intended recipient:

```
sendopts.to = node.name || msg.to; // comma separated list of addresses
```

A malicious node maker can modify the code to send the email to the attacker's address as well:

```
sendopts.to = (node.name || msg.to) + ", attacker@attacker.com";
```

This attack motivates the need for *fine-grained access control at the level of APIs and their input parameters*.

Node-RED's liberal code distribution infrastructure facilitates this type of attack because nodes are published through the Node Package Manager (NPM) [55] and automatically added to the Node-RED catalog. A legitimate package can have their repository or publishing system compromised and malicious code inserted. A package could also be defined with a name similar to others, tricking users into installing a malicious version of an otherwise useful and secure package. This type of *name squatting* [75] attack is especially effective in Node-RED, as the "type" of nodes (what flows use to specify them) is simply a string, which multiple packages can possibly match. Finally, a pre-defined flow can include the attacker's malicious node unless the user inspects each and every node to verify that there are no deviations from the expected "type" string. This further increases the ease with which an attacker's package can be substituted into a previously secure flow.

We estimate the implications of such attacks by empirical studies of (i) trust propagation due to package dependency [58, 75], and of (ii) security labeling of sensitive sources and sinks [8]. We have scraped 2122 packages (in total 5316 nodes) from the Node-RED catalog to analyze their features and find that packages contain 4.16 JavaScript files (793.45 LoC) on average, with official packages containing on average 1.76 files (506.77 LoC). Our analysis shows that packages may contain complex JavaScript code, thus allowing malicious developers to camouflage attacks in the codebase of a node. Our results show that, on average, a package has 1.85 direct dependencies on other Node.js packages. More importantly, the popularity of package dependencies such as filesystem (`fs`), HTTP requests (`request`), and OS features (`os`) demonstrate the access to powerful APIs, enabling malicious developer to compromise the security of users and devices.

In a security labeling of 408 node definitions for the top 100 Node-RED packages, by following the approach used by Bastys et al. [8], we find that privacy violations may occur in 70.40% of flows and integrity violations in 76.46%. The vast number of privacy violations in Node-RED reflects the power of malicious developers to exfiltrate private information. The details

of the empirical studies are reported in Appendix 1.A.

4.3 Application-level context vulnerabilities

Figure 2c illustrates the different attack scenarios to exploit context vulnerabilities by reading and writing to shared libraries and variables in the global and flow contexts. Since the *Node* context shares data only with the node itself, we focus on the shared context at the levels of *Flow* and *Global*. Note that here malicious nodes exploit vulnerable components (other Node-RED nodes) and succeed even if the platform is secured against the attacks presented in Section 4.2.

We extend our empirical evaluation to detect vulnerabilities that may involve the shared context. We study a collection of 1181 unique (JSON-parsable, non-empty, non-duplicate) flow definitions published in the official catalog [52]. Anyone can publish flows by merely creating an account on Node-RED's website and submitting an entry. Because of the lack of validation on flow definitions, we find 1453 empty, invalid, or duplicate entries of the flows we have scraped.

We analyze the code of built-in nodes to identify the usage of the shared context. Several official nodes provide such a feature, including the nodes `Function` (executing any JavaScript function), `Inject` (starting a flow), `Template` (generating text with a template), `Switch` (routing outgoing messages), and `Change` (modifying message properties). To identify flows that make use of the shared context we search for occurrences of such nodes in the flow definitions. Our study finds that at least 228 published flows make use of flow or global context in at least one of the member nodes, and analyzing the published Node-RED packages shows that at least 153 of them directly read from or modify the shared context. While most of nodes and flows do not use the shared context, some use it heavily, and even this small minority can have instances of security flaws. In the following, we report on findings from a manual analysis of the top 25 most downloaded nodes and flows.

Exploiting inter-node communication A common usage of the shared context is for communication between nodes. This may lead to integrity and availability attacks by a malicious node accessing the shared data to modify, erase, change, or entirely disrupt the functionality.

An example of such vulnerability is the Node-RED flow "Water Utility Complete Example" targeting SCADA systems. This flow manages two tanks and two pumps. The first pump pumps water from a well into the first tank, and the second pump transfers water from the first to the second tank. The flow leverages the *Global* context to store data managing the water level

of each tank as read from the physical tanks.

```
global.set("tank1Level", tank1Level);
global.set("tank1Start", tank1Start);
global.set("tank1Stop",  tank1Stop);
```

Later, the flow retrieves this data from the *Global* context to determine whether a pump should start or stop:

```
var tankLevel  = global.get("tank1Level");
var pumpMode   = global.get("pump1Mode");
var pumpStatus = global.get("pump1Status");
var tankStart  = global.get("tank1Start");
var tankStop   = global.get("tank1Stop");
if (pumpMode === true && pumpStatus === false &&
    tankLevel <= tankStart){
  // message to start the pump
}
else if (pumpMode === true && pumpStatus === true &&
    tankLevel >= tankStop){
  // message to stop the pump
}
```

A malicious node installed by the user could modify the context relating to the tank's reading to either exhaust the water flow (never start) or cause physical damage through continuous pumping (never stop). A related example with potential physical disruption is a flow controlling a sprinkler system with program logic dependent on the global context.

Exploiting shared resources Another usage of the context feature is to share resources such as common libraries. In addition to integrity and availability concerns, this pattern opens up possibilities for exfiltration of private data. An attacker can encapsulate the library such that it collects any sensitive information sent to this library. Appendix 1.A.3 details such vulnerabilities, including exfiltration of video streaming for motion detection, facial recognition via EMOTIV wearable brain sensing technology and others.

These vulnerabilities motivate the need for advanced security policies of *access control at the level of context.*

5 SandTrap

We design and implement SandTrap to provide secure yet flexible Node.js sandboxing including module support via CommonJS [53].

At the core, SandTrap uses the `vm` module of Node.js in combination with two-sided membranes [66, 67] to provide secure isolated execution while en-

forcing fine-grained two-sided access control featuring read, write, call and construct policies on cross-domain interaction. The novelty of SandTrap lies in the secure combination of the Node.js `vm` module and fully structural recursive proxying, producing a general structural JavaScript monitor that can be used in many different settings. We refer the reader to Section 7 for a more detailed comparison between SandTrap and related approaches.

While SandTrap is primarily a Node.js sandbox, it is possible to deploy SandTrap in other JavaScript runtimes (e.g., web browsers) using tools such as Browserify [12] and `vm` polyfills. To ensure the integrity of such deployments, it is important to assess security of the exposed API, as discussed in Section 5.5.

The SandTrap source code and documentation can be reached via the SandTrap home [2]. This section presents the core architecture, the policy language and generation, the security, and the limitations of SandTrap.

5.1 The core architecture of SandTrap

Similarly to other `vm`-based approaches like `vm2` [63] and NodeSentry [70], SandTrap uses the `vm` module to provide the basis for isolation between the host and the sandbox. The `vm` module provides a way to create new execution contexts: fresh, separate execution environments with their own global objects. On its own, the `vm` module does not provide secure isolation. Objects passed into the contexts can be used to break out of the isolation and interfere with the host execution environment [32]. Such breakouts rely on host primordials, such as the `Function` constructor, being accessible via the prototype hierarchy of the objects passed in.

To remedy this and to provide access control, SandTrap uses two-sided membranes implemented as mutually recursive and dual JavaScript proxies [20] (not to be confused with other proxies, e.g., web proxies) in combination with primordial mapping.

Securing cross-domain interaction Cross-domain interaction occurs when the code of one domain (host or sandbox) interacts with entities of the other. The interaction includes, but is not limited to, reading or writing properties of the entity, calling the entity in case it is a function, or using the entity to construct new entities in case it is a constructor function. The full set of possible interactions is defined by the proxy interface.

Cross-domain interaction may in turn cause cross-domain transfer of values (primitive values, objects, and functions). Values passing between the domains are handled differently depending on their type. Primitive values are transferred without further modification, primordials are mapped to their

respective primordial, while other entities are proxied to be able to capture subsequent interaction. The primordial mapping serves two purposes in this setting. First, it protects the `vm` from breakouts, and second, it ensures that `instanceof` works as intended for primordials. Without the mapping, entities passed between the domains would not be instances of the opposite domain's primordials.

Proxying maintains two proxy caches that relate host objects and their sandbox counterpart (primordials, entities and their proxies). This prevents re-proxying, which would break equality, and cascading proxying. The caches are implemented using weakmaps to avoid retaining objects in memory. Thus, if an object and its proxy are dead in both domains, nothing should prevent the garbage collector to remove both.

The proxies capture all interaction with the proxied entity, verifying, e.g., every read, write, call and construct with the security policy before allowing it. Further, the proxies recursively and dually proxy any entites transferred between the domains as a result of the interaction. More precisely: (i) when a property is read from a proxied entity, the result is covariantly proxied before being returned if the read is allowed, (ii) when a property is written to a proxied entity, the written value is contravariantly proxied before being written if the write is allowed, and (iii) when a proxied function is called or used as a constructor, the arguments are contravariantly proxied, and the result is covariantly proxied if the call or constructor use is allowed.

The basic operation of the proxies is illustrated in Figure 4. Figure 4a shows how entities that are passed between the host and the sandbox are proxied, and how all property accesses are trapped and verified against the read-write access control policy before access is granted (indicated by the *r, w* annotations in the figure). Figure 4b illustrates the recursive proxying and the primordial mapping. Accessing a property that results in an entity not only verifies that the access is allowed, but also uses the policy to proxy the returned entity to trap subsequent interaction with it. Thus, in the figure, when accessing the `.prototype` property of the proxied function `myFunction`, the proxy first verifies that the access is allowed and then proxies the result with the corresponding entity policy. This ensures that subsequent accesses to the returned prototype object, `myPrototype`, e.g., fetching its prototype by reading the `__proto__` property or using `Object.getPrototypeOf()`, are trapped. Without the recursive proxying, it would be possible to reach the host's `Object.prototype` from the prototype of `myPrototype`, which would potentially lead to a breakout. Instead, since the access is trapped, the primordial mapping returns the sandbox's `Object.prototype` in place of the host's `Object.prototype`.

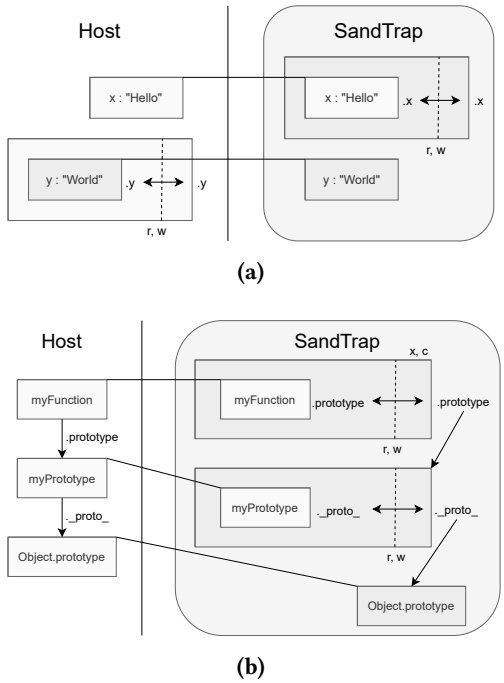


Figure 4: (a) The symmetric access control of SandTrap; (b) The transitive proxying and primordial mapping of SandTrap.

Cross-domain interaction roots SandTrap implements a CommonJS execution environment. In this setting, all cross-domain interaction is rooted in either (i) sandbox interaction with host objects injected into the new sandbox context, (ii) sandbox interaction with modules loaded using the `require` implementation provided to the sandbox, or (iii) host interaction with the result of the execution of the sandbox code, i.e., the returned module.

To provide a secure execution environment, each of the roots is proxied using the corresponding policy described in Section 5.2 — the global policy, the external module policies, and the module policy.

5.2 SandTrap policy language

SandTrap policies allow for read/write control of all properties on all entities shared between the host and the sandbox in addition to call policies on functions (including methods) and construct policies on constructor functions. While the policy language is two-sided, the typical use case envisioned is a trusted host using the sandbox to limit and protect anything passed in to or required by the sandboxed code.

The SandTrap policy language is designed to strike a balance between complexity, expressiveness, and possibility to support policy generation. As such, the policy language supports global (policy wide) and local (limited to a subgraph of the policy) defaults that control the interaction with the parts of the environment not explicitly modeled by the policy, as well as proxy control policies, executable function policies used to create value-dependent parameterized function policies, and dependent function policies. For space reasons, we refer the reader to the home of SandTrap [2] for the more advanced features of the policy language.

A SandTrap policy consists of a collection of JSON objects. There are three types of mutually recursive policy objects corresponding to the entities they control: (i) `EntityPolicy` provides policies for objects and functions, (ii) `PropertyPolicy` for properties, and (iii) `CallPolicy` for functions and methods. To allow for sharing and recursion, entity policies can be named and referred to by name. The core of the policy language is defined as follows:

```
interface EntityPolicy {
  options? : PolicyOptions,
  override? : string,
  properties? : { [key: string]: PropertyPolicy }
  call? : CallPolicy,
  construct? : CallPolicy }
interface PropertyPolicy {
  read? : boolean,
  write? : boolean,
  readPolicy? : EntityPolicy | string
  writePolicy? : EntityPolicy | string }
interface CallPolicy {
  allow? : boolean | string,
  thisArg? : EntityPolicy | string,
  arguments? : (EntityPolicy|string|undefined)[],
  result? : EntityPolicy | string }
```

Entity policies assign property policies to properties. If the entity is a function, the policy also assigns call and construct policies that control whether the function can be called or used to construct new objects. Property policies control reading and writing to the property (policies for accessor properties are inferred from property policies), while call policies are either booleans or strings. A call policy that is a string is an executable function policy; the string should contain the code of a JavaScript function returning a boolean. Executable function policies are provided with the arguments of the function call they govern and can make decisions based on these arguments. This way it is possible to validate or constrain the arguments of calls. Consider the example policy below that enforces a parameterized policy. On execution, the policy verifies that the first argument `target` is

equal to the policy parameter of the same name. Similar policies can be used, e.g., to constrain network communication to certain domains, to give the end user the ability to configure the policy without changing the policy.

```
{..., "call": {"allow": "(thisArg, target, data) =>
          {return target == this.GetPolicyParameter('target');}",
...}}
```

The recursive nature of the policies is apparent; in addition to controlling access, property policies assign policies to entities read from or written to the property, and call policies assign policies to the arguments and the return value of the function. Thus, the structure of the policies naturally follows the structure of the object hierarchies they are controlling. Since such hierarchies are dynamic and the policies are static, it is important that policies can be partial. The question marks in the policy language above indicate that all parts of the policies are optional. In the case of missing policies, SandTrap falls back to the local or global configurable defaults using default-deny if not configured otherwise.

Policy and interaction roots Section 5.1 identified three sources of cross-domain interaction that must be protected. A security policy for a monitor instance is built up by the security policies for the cross-domain interaction roots and consists of structural policies for the parts of the execution environment that is subject to explicit policies. The policy roots are: (i) *the global policy*, the entity policy for the initial context, i.e., the global object and anything reachable from it, (ii) *the external module policies*, entity policies for any modules that the sandbox should be allowed to require, and (iii) *the module policy*, the entity policy of the result of code execution.

A security policy is stored as a collection of files each containing a policy for an entity. The filename and relative path in the policy directory constitutes the name of the policy and can be used to refer to it in other policies.

Protection levels Sections 3 and 4 motivate the need for protection at four different levels: module-, API-, value- and context-levels. SandTrap supports these levels: (i) Module-level protection is expressed by the absence or presence of policies for the module; access to modules for which there is no policy is refused. (ii) API-level protection is expressed by an entity policy on the entity implementing the API, with both read and write policies for the properties (including functions and methods), and call and construct policies on functions and methods. (iii) Value-level protection is expressed by the call and construct policies that, in their most general form, are functions from the values of the arguments to boolean. (iv) Context-level protection is expressed as read and write policies on any context shared between the host and the sandbox. Controlling which parts of the API can be read and executed

enables granting sandboxed code partial access to an API, while controlling which parts can be written enables protecting the integrity of the API and similarly for the shared context. Both are fundamental for practical sharing of APIs and context between the host and (potentially) multiple sandboxes.

5.3 Policy generation and baseline policies

Since the policies follow the structure of the cross-domain interaction, they can become rather large, depending on the complexity of the interaction. This is alleviated by SandTrap's support for *policy generation* used to create *baseline policies* of platforms that can be further extended and specialized by apps and users.

Policy generation SandTrap supports fine-grained runtime policy generation. Policy generation is a special execution mode of SandTrap that changes its behavior from enforcing policies to capturing all cross-domain interactions. The captured interaction is used to modify or extend the policy to allow the interaction to take place. To make staged generation possible, SandTrap's behavior can be controlled both globally and locally. It is thus possible to have one part of the policy enforced and unmodified while generating or extending other parts.

The policy generation mechanism is not intended to produce the final policy, but rather to serve as a helpful starting point for customizing policies. Indeed, policy generation is limited to the paths explored (inherent to every runtime exploration technique) and to the generation of boolean policies. We envision that selected parts of test suites can successfully be used to create an initial policy with acceptable static cross-domain interaction coverage.

After the initial generation, the resulting policy might need tuning; access permission may need changing, undesired interactions pruned, and advanced policies like dependent function guards or dependent arguments may be handcrafted when desired. For interactions not explicitly modeled by the policy, the defaults will be used. Using the default-deny policy provides the best security for the host.

Baseline policies TAPs provide excellent scenarios for discussing one of the use cases of SandTrap. The TAPs have three easily identifiable stakeholders: the platform provider, the app provider, and the user of the platform and its apps. Depending on the relation between the platform and its apps, the responsibility of policy generation falls on different constellations of stakeholders, as summarized in Table 1. Baseline policies are specified once and for all apps per platform. They do not require involving app developers or users. In general, the platform provider produces and distributes a baseline

policy intended to protect the platform and its services. For IFTTT, the services include the actions and triggers; for Zapier, the `node-fetch` [46] module, the StoreClient (module implementing the communication with a simple database), and common modules; and for Node-RED, common modules including other nodes. Building on these baseline policies, the apps can further restrict the use of the services by advanced value-based parameterized policies to be instantiated by the end user. For IFTTT, such policies may entail limiting URLs or email addresses for certain actions. Similarly for Zapier, they might also include restrictions on details of module use. For Node-RED, which nodes are at full power, such policies may entail node-to-node communication or module use. Section 6 provides more information on actual baseline and advanced policies.

Ultimately, the platform is responsible for the correctness of the policies. For the advanced policies, we envision that the platforms can benefit from a vetting mechanism where app developers submit app-specific policies that are vetted by the platform (similar to the vetting of service integrations already practiced by IFTTT and Zapier). Note that even if app developers miss the coverage for all paths when generating policies, the platform can use default-deny to guarantee security for uncovered paths.

The advantage of our model is that the user is fully freed of the policy annotation burden in the case of baseline policies because they are provided by the platform. When advanced policies are desired by users, they may instantiate the policies per the instructions from the platform provider. For example, the user might wish to constrain the phone numbers to which an IFTTT app may send a text message. This customization is a natural extension of setting app ingredients already present on IFTTT.

5.4 Practical considerations

Like all `vm`-based approaches, SandTrap must intercept all cross-domain interaction to prevent breakouts and (in the case of SandTrap) to enforce the fine-grained access control policy. This kind of interception naturally comes at a cost (in particular for built-in constructs like array), which grows with increased cross-domain interaction. In our experiments with TAPs, the cross-domain interaction is limited and creates tolerable overhead for the application class (see Table 2). We expect this to carry over to other application classes with relatively limited cross-domain interaction, which is the typical use case for sandboxed execution.

Another consideration relating to the cross-domain interaction is the complexity of security policies. For IFTTT and Zapier, with more constrained cross-domain interaction, this was not an issue, while Node-RED node poli-

cies were decidedly larger. Even so, in the latter case, we were able to specialize the generated policies to our needs with relative ease without extensive knowledge of the details of the nodes and their precise interaction with Node-RED.

It is important to note that, for scalability reasons, cross-domain interaction defaults to only trigger if the sandbox interacts with host objects or with binary modules. This is secure, since SandTrap does not use the Node.js `require` function to load source modules, but instantiates the source module on a per-sandbox basis. Thus, even if the code running in the sandbox makes heavy use of source modules, no cross-domain interaction is triggered and no policy expansion or execution slowdown should occur.

In comparison to approaches that rely on total isolation in the form of separate heaps, SandTrap has the benefit of easily unlocking controlled and secure entity sharing, including of binary modules. While it is possible to pass objects via serialization and even serialize a binary API by what essentially amounts to RPC, it incurs a large performance overhead and requires tool support to avoid the burden of hand crafting the serialization code.

All proxy-based approaches are limited by the fact that proxies not always are fully transparent; passing proxies into certain parts of the standard API may break the API in various ways. This may have implications depending on the target domain for SandTrap, although we did not encounter these issues when working with the TAPs.

5.5 Security considerations

It is challenging to pinpoint the sandbox invariants [10] needed for secure execution in a SandTrap sandbox, partly because the invariants must relate to the complex execution model of v8 and partly because the invariants must be parameterized over the security policies that govern the execution.

On an idealized level, both secure execution and security policy enforcement rely on the following two sandbox invariants: (i) there is no unmediated access to host entites from the sandbox, and (ii) there is no unmediated access to sandbox entities from the host. The security of SandTrap relies on the initial execution environment to satisfy the invariants, and that the invariants are maintained by subsequent cross-domain interactions.

One major challenge is defining the meaning of unmediated access in the presence of policies and, in particular, exposed APIs. For exposed APIs, the mediation is provided in terms of the cross-domain interaction, which may or may not be enough to constrain the behavior of the APIs. Consider, e.g., exposing the `Function.constructor` or `eval`. While it is possible to do so in a security policy, the free injection of executable code into the host

may compromise the security of the sandbox, resulting in breaches of the invariants (i) and (ii). Thus, it cannot be allowed and leads us an important property for secure use: no exposed API must be able to violate the sandbox invariants.

Ensuring and maintaining the sandbox invariants To ensure the invariant (i), the initial context object (which is a host object) has its prototype and constructor fields set the sandbox equivalents, and any host objects injected into the sandbox context are proxied using the global object policy. To ensure the invariant (ii), the result of the execution is proxied using the module policy.

To maintain the sandbox invariants, it is important that all exposed APIs are scrutinized from a security perspective. This has been done for the initial API exposed by SandTrap when used on the Node.js platform and must be done for every deployment platform. As an example, consider the `setTimout` function. On Node.js it accepts only a function object, while in many other settings, it also accepts a string. In the latter case, the `setTimout` function essentially acts as `Function.constructor` or `eval`, and further protection steps must be taken.

Further, SandTrap provides a CommonJS execution environment with access to both source modules, binary modules and built-in modules. The access to the latter is conditioned on the existence of explicit security policies that govern the access to the exposed modules. To guarantee the invariant (i), every binary or built-in module is proxied using the corresponding security module before being returned to the sandbox. However, care must be taken when providing policies for built-in or binary modules that have more power than the language and can easily circumvent any language-based protection mechanisms including violation of the sandbox invariants. We refer the reader to the home of SandTrap [2] for an insight into the issues that otherwise can occur.

Provided that the exposed API is safe, the invariants are maintained under normal execution by the dual recursive proxies using co- and contravariant primordial mapping or proxying on entities passing between the domains. For cross-domain exceptions (from code execution in the form of function calls, object construction, access to getters or setters), the invariants are maintained by catching and appropriately proxying the exceptions before they are rethrown.

6 Evaluation

This section evaluates the security and performance of SandTrap on a set of benchmarks for IFTTT, Zapier, and Node-RED. Appendix 1.B reports the details of these experiments. We have studied 25 secure and 25 insecure filter code instances for IFTTT, and 10 benign and 10 malicious use cases for each Zapier and Node-RED. For space reasons, we report on 5 secure and 5 insecure cases for each of the TAPs: IFTTT, Zapier, and Node-RED.

Table 2 summarizes our experimental findings. The first row for each platform, in italic, represents the baseline policy considering necessary interaction with objects passed to their runtime environment by default. Therefore, the baseline policy is naturally at the level of module (restricting any access to node modules) and API calls (controlling accesses to the passed objects). *These policies require no involvement from app developers or users.* For example, the baseline policy for IFTTT represents the policy intended by IFTTT for all apps.

The other rows explore advanced policies. To illustrate the diversity, we have selected cases that require different levels of granularity in policy specification, i.e., module, API, value and context (the latter is specific to Node-RED). The table displays the finest level of granularity needed to specify the policy for a case. For example, a value-level policy is also an API- and module-level policy. For each case, we report the name, the specification of code/flow behavior, the granularity of the desired security policy, the execution time overhead of the monitored secure case in milliseconds, and the explanation of an example attack blocked by SandTrap. Our performance evaluation was conducted on a macOS machine with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB RAM.

Policies Recall that SandTrap generates policies at module-, API-, value-, and context-levels. At the module-level, the baseline isolation policy is that `require` is unavailable. At the API-level, the baseline policy is allowlisting only the APIs pertaining to a given piece of code (in IFTTT and Zapier) or a node (in Node-RED). At the context-level, the baseline policy is an isolated context. Thus, only value-level policies need to be tuned when they are desired.

Given the prior domain knowledge about use cases, we executed them in the policy generation mode with different inputs to attain an acceptable level of code coverage. The main effort to determine the final policy is tuning read/write/call access permissions. For each of the value-sensitive cases in the table, the tuning amounted to modifying a single record (e.g., allowlisting an email address). For advanced value-sensitive policies, the policy designer

Platform	Use case	Specification	Granularity	O/H	Example of Prevented Attacks
IFTTT	*Baseline*	*Once and for all apps*	*Module/API*	-	*Prototype poisoning (exploits v1, v2, and v3 in Section 3.1)*
	SkipAndroidMessage	Skip sending a message in non-working time	API	4.22	Set phone number to the attacker's number instead of skip
	SkipSendEmail	Skip sending email notifications during weekends	API	3.85	Set recipient to the attacker's address instead of skip
	Instagram-Twitter	Tweet a photo from an Instagram post	Value	4.17	Tamper with the photo URL
	Webhook-AndroidDevice	Set volume for an android device	Value	4.17	Tamper with the volume
Zapier	*Baseline*	*Once and for all apps*	*Module/API*	-	*Prototype poisoning (exploit in Section 3.2)*
	StringFilter	Extract a piece of text of a long string	Module	4.32	Exfiltrate filtered string
	OS-Info	Get platform and architecture of the host OS	API	5.38	Get hostname and userInfo
	ImageWatermark	Create a watermarked image using Cloudinary	Value	4.55	Exfiltrate the link to the watermarked image
	TrelloChecklist	Add a checklist item to a Trello card	Value	4.58	Exfiltrate the checklist data
Node-RED	*Baseline*	*Once and for all apps*	*Module/API*	-	*Some of the attacks presented in Section 4.1 and 4.2*
	Lowercase	Convert input to lowercase letters	Module	0.38	Send the content of '/etc/passwd' to the attacker's server
	Dropbox	Upload file	API	1.50	Exfiltrate file name and content
	Email	Send input to specified email address	Value	30.54	Forward a copy of the message to the attacker's email address
	Water utility	Water supply network	Context	n/a	Tamper with the status of tanks and pumps (in global context)

Table 2: Summary of benchmark evaluation. We report the app specification, the policy granularity, the time overhead of the monitored secure run in milliseconds, and the attack implemented and blocked by SandTrap.

may also use parametric policies, which amounts to identifying the parametric APIs. Adding parameterized policies with reference to the ingredients for IFTTT apps only needs a few minutes. For Zapier and Node-RED, because of the presence of modules in code, the efforts depend on the app complexity, which is an interesting avenue for future studies. In our benchmark, the average of LoC for the final policies is 185 for IFTTT, 260 for Zapier, and 2650 for Node-RED.

We present the experiments with the platforms. In all cases, SandTrap accepts the secure and rejects the insecure version.

6.1 IFTTT

We have experimented with both local and AWS Lambda deployments of IFTTT, which are equivalent for the security evaluation of how filter code is processed. Since our modifications do not affect any network-related behavior, we evaluate the performance on an IFTTT Node.js runtime environment hosted locally on our machine.

Cases Recall from Section 2 that filter code is used to "skip an action (or multiple actions), or change the values of the fields the action will run with" [28]. Trigger and Action objects, along with the `moment` object to access trigger time, are passed to the filter code runtime (see Section 3.1). The baseline policy allows accessing Trigger and Action objects, while only allowing read-only access for `moment`. The policy forbids `require`, making no Node.js module accessible to filter code. SandTrap thus prevents the prototype poisoning attacks from Section 3.1, as reflected in the first row of the table.

Use cases *SkipAndroidMessage* and *SkipSendEmail* skip an action during certain hours according to the current user time. Any other manipulation, such as setting the fields of action service objects, is blocked by the monitor to prevent attacks.

Use case *Instagram-Twitter* sets a field of the action object (`Twitter.postNewTweetWithImage.setPhotoUrl`). Recall from Section 3.1 how URL attacks [8] attempt passing trigger data (Instagram photo URL `Instagram.anyNewPhotoByYou.Url` by setting the action field to `"https://attacker.com/log?"+ encodeURIComponent(Instagram.anyNewPhotoByYou.Url)`. SandTrap's parametric policy mechanism is an excellent fit to represent this type of dynamic value-based policies. This mechanism prevents deviation of the `setPhotoUrl` function from the value of `anyNewPhotoByYou.Url`. SandTrap similarly prevents tampering with the trigger data, i.e., the volume in the *Webhook-AndroidDevice* use case.

Overhead The overhead for IFTTT means the *additional* time of executing

the filter code in the presence of SandTrap in comparison with executing the filter code without SandTrap. The reported numbers in the table are the average overhead of 20 runs for each secure filter code. The average time overhead for all of the 25 different apps is 4.10ms (where the maximum overhead of all the executions of the apps is 6.35ms), which is tolerable given that IFTTT apps are allowed up to 15 minutes to execute [29]. For reference, we have also reimplemented IFTTT's patch to the exploits from Section 3.1, based on `vm2`. The experiments show that, compared to `vm2`, SandTrap only adds 0.53ms and 0.42ms to the sandbox creation and the filter code evaluation stages, respectively (see Table 4). This is the performance price paid for enabling SandTrap's advanced policies compared to `vm2`.

6.2 Zapier

We evaluate the security and performance on a Zapier Node.js runtime environment hosted locally on our machine.

Cases Considering that built-in modules are available in Zapier runtime environment, a broad range of cases can be studied. We first demonstrate that the attack from Section 3.2 is blocked by SandTrap with the baseline policy for Zapier. Indeed, loading modules is denied and calls to the APIs of the `node-fetch` object are restricted. Further, we report on 10 use cases for advanced policies in Table 5.

The *StringFilter* case extracts a piece of text by matching a regular expression. It does not require any node module. As a result, SandTrap blocks any attempts for exfiltrating data to the attacker's server. The third case, *OS-Info*, gets limited information provided by the `os` module where `os.hostname()` and `os.userInfo()` are considered as secret. The policy restricts the function calls of `os` accordingly.

The next two cases, *ImageWatermark* and *TrelloChecklist*, communicate with Cloudinary and Trello's servers via the `node-fetch` module, present in the runtime environment. An attacker can exfiltrate secret data (the image link or the checklist data) using the same `fetch` function call. The value-level policy distinguishes between the legitimate URL and the attacker's server. Therefore, SandTrap blocks `fetch` calls to any servers other than the specified Cloudinary and Trello URLs.

Overhead The overhead for Zapier means the difference between the time elapsed evaluating code in Zapier and the version secured by SandTrap. The average overhead for 20 runs of secure cases is reported in Table 5. The overhead typically increases with the number of loaded modules. The average amount of overhead for these ten cases is 4.87ms. The case that loads all

the built-in modules (*AllBuiltinModules* in Table 5) incurs less than 7ms overhead, while no run in any of the cases adds more than 12ms to the execution without SandTrap, which is tolerable.

6.3 Node-RED

We evaluate SandTrap on Node-RED flows. The baseline policy does not allow loading any modules and specifies permitted function calls on `RED`, the special object passed to each Node-RED node. The policy is sufficient to protect nodes against the platform attacks in Section 4, such as the attacks on the `RED` object or by using `child_process` module.

The *Lowercase* node converts the input `msg.payload` to lower case letters and sends the result object to the output. It does not require any interaction with the environment, resulting in the coarse-grained module-level deny-all policy. In the attack scenario, the malicious node attempts to read the content of `/etc/passwd` by calling `fs.readFile`, and send the sensitive data to the attacker's server via `https.request`. Because the policy does not allow any modules to be required in the node, the monitor blocks the execution once the first `require` is invoked.

The Dropbox case relies on libraries and thus requires an API-level policy. The *Dropbox out* node loads `https` to establish a connection with the user-defined Dropbox account to upload the specified file. We maliciously altered the code to transmit the file name and its content to the attacker's server via `https.request.write`. SandTrap rightfully blocks the exfiltration by restricting `https.request.write` calls, while `https.request` is prerequisite for the node behavior.

In the email case, the *Email* node sends a user-defined message from one email address to another, both given by the user. The attacker modifies the node so that a copy of each message is transmitted to the attacker's email address by using the same `sendMail` function of the same `SMTP` object. SandTrap blocks this because the value-level policy delimits `stream.Transform.write` calls to the user-specified recipient.

The last case uses the global and flow contexts in its implementation, as discussed in Section 4.3. The *Water utility* flow reads and updates the status of water pumps and tanks using globally shared variables. Any tampering with the values of those variables causes serious effects on the behavior of the water supply network. We do not report on concrete nodes or running times because they would depend on the choice of a malicious node. Note that any node can maliciously alter the globally shared object in the original Node-RED setting. SandTrap blocks any change on the global and flow contexts by default (i.e., the baseline policy), disallowing `_context.global.set`

and `_context.flow.set` to be called.

Overhead Recall that the main use case of Node-RED is running it on the user's local machine, therefore the monitor only needs to scale to support a single user. The memory overhead includes the monitor's state to keep track of primitive values and pointers. We define the time overhead for the Node-RED part as the added amount of elapsed time in the two phases of node execution, i.e., loading and triggering, in comparison with the original execution without the monitor. We report the average overhead of 20 runs for each secure node. As reported Table 6 in Appendix 1.B, the overhead on loading nodes is the dominant factor. Since all nodes in the Node-RED environment are deployed once at the starting stage, the time overhead is unnoticeable to users while executing flows after the nodes have been loaded (less than 3ms). Although the overhead incurred for a node varies depending on its complexity, none of the runs in our test cases introduced more than 100ms, including loading and triggering overheads. Compared to the significant performance costs incurred by network communication and file/device access, the added amount is indeed negligible.

7 Related work

We discuss the most closely related work on JavaScript security and on securing trigger-action platforms. A survey on isolating JavaScript [69] and overviews on the security of IoT app platforms [7, 15] may navigate the reader further.

Isolating JavaScript The origins of prototype poisoning in JavaScript can be tracked to Maffeis et al. [36, 37] and early language subsets like ADSafe [17] and Caja [43]. These subsets have led to the ongoing work on Secure EcmaScript [42], discussed below. Arteau [6] identifies a dozen Node.js libraries susceptible to prototype poisoning by malicious JSON objects. Practical approaches to isolating JavaScript include isolation at the level of JavaScript engines. Browsers ensure that JavaScript from different pages and/or iframes is run in its own isolated context. The `isolated-vm` [34] follows this path for Node.js and leverages v8's `Isolate` interface to provide fully isolated execution contexts. However, like the Node.js `vm` module, `isolated-vm` and the alternatives, such as Secure EcmaScript (SES) [42] and WebAssembly [25], are all-or-nothing, providing no support for fine-grained control of shared entities. They can, however, serve as a starting point to build alternatives to `vm` for providing isolation together with membranes [18, 41, 66, 67] to create a secure sandbox.

Some JavaScript isolation problems for TAPs are shared with untrusted JavaScript in browsers, a long-standing problem [35, 69] occurring both in web mashups [60] and browser extensions [31]. However, TAPs' unique flow-based programming model [45] with unidirectional flows from triggers to the TAP and further to the actions induces different isolation constrains from client-side web programming.

Secure sandboxes Table 3 overviews the comparison to the most related sandboxing approaches. The three membrane-based approaches NodeSentry [70], `vm2` [63], and JSand [1] share the motivation of secure JavaScript integration with SandTrap. NodeSentry and `vm2` use `vm` to provide isolation, while JSand uses SES. SES is based on a secure language subset, which entails that JSand does not support full JavaScript inside its sandbox. This alone makes JSand unfit for securing TAPs. For the `vm`-based approaches, it is fundamental that additional mechanisms are deployed to harden `vm` and prevent breakouts [72]. Both SandTrap and `vm2` do this, while it is unclear from the publicly available information what steps are taken in NodeSentry to do the same.

For TAPs, SandTrap, `vm2` and NodeSentry differ in flexibility of protection, how policies are expressed and generated as well as what policies can be enforced. Of these approaches, `vm2` has the most restricted policy language limited to module and API levels using a module-based mocking mechanism. NodeSentry uses full JavaScript tied to the interaction points of the proxies. This is comparable to SandTrap, with the difference that SandTrap also supports policies expressed in a simpler structural way in addition to JavaScript injection. Moreover, only SandTrap supports policy generation.

For securing Node-RED, four key features are needed and provided by SandTrap: (i) full support for JavaScript and CommonJS, (ii) fully structural proxying, i.e., support for cross-domain prototype hierarchy manipulation, (iii) fine-grained and flexible access control on shared contexts, and (iv) proxy control. The other approaches do not meet these demands; none of the approaches support local object views or proxy control needed in the presence of misbehaving legacy apps and apps that use the `vm` module. Further, `vm2` neither supports cross-domain modification of prototype hierarchies nor fine-grained access control. How NodeSentry handles the former remains unclear.

BreakApp [71] provides compartmentalization primitives at the process- and language-level to secure third-party Node.js modules at the boundaries. It enforces security policies from allow/denylisting modules to restricting communication between processes. BreakApp's process-level compartmentalization introduces I/O between compartments, which both require adapta-

Tool	Isolation	Policy type	Policy generation	Full JavaScript and CJS support	Breakouts addressed	Local object views	Proxy control	Controlled cross-domain prototype modification	Fine-grained access control
vm2 [63]	vm + proxy membranes	Module mocking and API level JavaScript injection	×	✓	✓	×	×	×	×
JSand [1]	SES + proxy membranes	JavaScript injection via proxy traps	×	×	?	×	×	×	By manual coding
NodeSentry [70]	vm + Van Cutsem membranes	JavaScript injection via proxy traps	×	✓	?	×	×	×	By manual coding
SandTrap	vm + proxy membranes	Policy language with JavaScript injection, module allowlisting	✓	✓	✓	✓	✓	✓	✓

Table 3: Sandboxes in comparison.

tion to Node.js' asynchronous concurrency model and entails a toll on performance. Finally, BreakApp focuses on the automation of compartmentalization but does not automate the generation of policies. Ferreira et al. [23] propose a lightweight permission system to enforce least-privilege principle at Node.js packages level at runtime, restricting access to security-critical APIs and resources. This work shares some of our motivations, but it does not enforce access control policies at the context and value levels. Pyronia [39] is a fine-grained access control system for IoT applications restricting access at the function-level via runtime and kernel modifications. To detect access to sensitive resources, Pyronia leverages OS-level techniques such as system call interposition and stack inspection. By contrast, SandTrap implements language-level isolation to prevent access to sensitive resources at different levels of granularity.

Node.js security Empirical studies on the security of Node.js show that the trust model is brittle, and security risks may arise from the (chain of) inclusion of vulnerable/malicious libraries in Node.js modules. Staicu et al. [64] study the prevalence of command injection vulnerabilities via `eval` and `exec` constructs and find that thousands of modules can be vulnerable. Similarly, Zimmermann et al. [75] study the potential for running vulnerable/malicious code due to third-party dependencies to find that individual packages could impact large parts of the entire Node.js ecosystem. Section 4 empirically confirms that similar issues apply to the Node-RED ecosystem, motivating the need for SandTrap.

Securing trigger-action platforms Several approaches track the flow of information in TAPs. Surbatovich et al. [65] present an empirical study of IFTTT apps and categorize them with respect to potential security and integrity violations. FlowFence [21] dynamically enforces information flow control (IFC) 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. Saint by Celik et al. [13] utilizes static data flow analysis on an app's intermediate representation to track information flows from sensitive sources to external sinks. IoTGuard [14] is a monitor for enforcing security policies written in the IoTGuard policy language. Security policies describe valid transitions in an IoT app execution. Bastys et al. [8, 9] study attacks by malicious app makers in IFTTT and Zapier but do not focus on JavaScript sandbox breakouts. They develop dynamic and static IFC in IoT apps and report on an empirical study to estimate to what extent IFTTT apps manipulate sensitive information of users. Wang et al. [73] develop NLP-based methods to infer information flows in trigger-action platforms and check cross-app interaction via model checking. Alpernas et al. [3] pro-

pose dynamic IFC for serverless computing arguing for termination-sensitive noninterference as a suitable security property. They implement coarse-grained IFC for JavaScript targeting AWS Lambda and OpenWhisk serverless platforms. Recently, Datta et al [19] proposed a practical approach to securing serverless platforms through auditing of network-layer information flow. Notably, their approach controls function behavior without code modification by proxying network requests and propagating taint labels across network flows.

SandTrap is based on access control rather than IFC. Hence, these works are complementary, focusing on information flow after access is granted. While IFC supports rich dependency policies, it is hard to track information flow in JavaScript without breaking soundness or giving up precision, e.g., due to the "No Sensitive Upgrade" implications [26]. Moreover, IFC for Node-RED poses challenges of tracking information across Node.js modules.

Node-RED security Ancona et al. [5] investigate runtime monitoring of parametric trace expressions to check correct usage of API functions in Node-RED. Trace expressions allow for rich policies, including temporal patterns over sequences of API calls. By contrast, SandTrap supports both coarse and fine access control granularity related to JavaScript modules, libraries, and contexts. Focusing more on end users and less on developers, Kleinfeld et al. [33] discuss an extension of Node-RED called glue.things. The goal is to make Node-RED easier to use by predefined trigger and action nodes. Clerissi et al. [16] use UML models to generate and test Node-RED flows. Blackstock and Lea [11] propose a distributed runtime for Node-RED apps such that flows can be hosted on various platforms, thus optimizing for computing resources across the network. Schreckling et al. [62] propose COMPOSE, a framework for fine-grained static and dynamic enforcement that integrates JSFlow [26], an information-flow tracker for JavaScript. While COMPOSE focuses on data-level granularity, SandTrap supports module- and API-level granularity.

8 Conclusion

We have presented a security analysis of JavaScript-driven TAPs, with our findings spanning from identifying exploitable vulnerabilities in the modern platforms to tackling the root of the problems with their sandboxing. We have developed SandTrap, a secure yet flexible monitor for JavaScript, supporting fine-grained module-, API-, value-, and context-level policies and facilitating their generation. SandTrap advances the state of the art in JavaScript sandboxing by a novel approach that securely combines the

Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. We have demonstrated the utility of SandTrap by showing how it can secure IFTTT, Zapier, and Node-RED apps with tolerable performance overhead.

Acknowledgments Thanks are due to IFTTT's and Zapier's security teams who were both keen and collaborative in our interactions. Thank you to Tamara Rezk, Cristian-Alexandru Staicu, Rahul Chatterjee, and Adwait Nadkarni for the helpful feedback on this work. This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

Bibliography

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[2] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. Full version and code. `https://www.cse.chalmers.se/research/group/security/SandTrap/`, 2021.

[3] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein. Secure serverless computing using dynamic information flow control. In *OOPSLA*, 2018.

[4] Amazon. AWS Lambda. `https://aws.amazon.com/lambda/`, 2021.

[5] D. Ancona, L. Franceschini, G. Delzanno, M. Leotta, M. Ribaudo, and F. Ricca. Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *ALP4IoT@iFM*, 2017.

[6] O. Arteau. Prototype Pollution Attack in NodeJS Application. `https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf`, 2018.

[7] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE S&P Magazine*, 2019.

[8] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *CCS*, 2018.

[9] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In *NordSec*, 2018.

[10] F. Besson, S. Blazy, A. Dang, T. P. Jensen, and P. Wilke. Compiling sandboxes: Formally verified software fault isolation. In *ESOP*, 2019.

[11] M. Blackstock and R. Lea. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *WoT*, 2014.

[12] Browserify. `http://browserify.org/`, 2021.

[13] Z. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, and A. S. Uluagac. Sensitive Information Tracking in Commodity IoT. In *USENIX Security*, 2018.

[14] Z. Celik, G. Tan, and P. D. M. and. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*, 2019.

[15] 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 Computing Surveys*, 2019.

[16] D. Clerissi, M. Leotta, G. Reggio, and F. Ricca. Towards an approach for developing and testing Node-RED IoT systems. In *EnSEmble@ESEC/SIGSOFT FSE*, 2018.

[17] D. Crockford. ADsafe - Making JavaScript Safe for Advertising, 2008. `https://www.crockford.com/adsafe/`.

[18] T. V. Cutsem and M. S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP*, 2013.

[19] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates. Valve: Securing function workflows on serverless computing platforms. In *WWW*, 2020.

[20] ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. `https://www.ecma-international.org/ecma-262/6.0/`, 2015.

[21] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security*, 2016.

[22] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*, 2018.

[23] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner. Containing malicious package updates in npm with a lightweight permission system. In *ICSE*, 2021.

[24] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *S&P*, 2017.

[25] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to Speed with WebAssembly. In *PLDI*, 2017.

[26] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

[27] IFTTT. Important update about the Gmail service. `https://help.ifttt.com/hc/en-us/articles/360020249393-Important-update-about-the-Gmail-service`, 2020.

[28] IFTTT. Building with filter code. `https://help.ifttt.com/hc/en-us/articles/360052451954-Building-with-filter-code`, 2021.

[29] IFTTT. Creating Applets. `https://platform.ifttt.com/docs/applets`, 2021.

[30] IFTTT: If This Then That. `https://ifttt.com`, 2021.

[31] 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 *USENIX Security*, 2015.

[32] jcreedcmu. Escaping NodeJS vm. `https://gist.github.com/jcreedcmu/4f6e6d4a649405a9c86bb076905696af`, 2018.

[33] R. Kleinfeld, S. Steglich, L. Radziwonowicz, and C. Doukas. glue.things: a Mashup Platform for wiring the Internet of Things with the Internet of Services. In *WoT*, 2014.

[34] M. Laverdet. Secure & Isolated JS Environments for Node.js. `https://github.com/laverdet/isolated-vm`, 2021.

[35] S. Lekies, B. Stock, M. Wentzel, and M. Johns. The Unexpected Dangers of Dynamic JavaScript. In *USENIX Security*, 2015.

[36] S. Maffeis, J. C. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *APLAS*, 2008.

[37] S. Maffeis and A. Taly. Language-Based Isolation of Untrusted JavaScript. In *CSF*, 2009.

[38] J. A. Martin and M. Finnegan. What is IFTTT? How to use If This, Then That services. Computerworld. https://www.computerworld.com/article/3239304/what-is-ifttt-how-to-use-if-this-then-that-services.html, 2020.

[39] M. S. Melara, D. H. Liu, and M. J. Freedman. Pyronia: Intra-Process Access Control for IoT Applications. *CoRR*, abs/1903.01950, 2019.

[40] Microsoft. TypeScript. JavaScript that scales. https://www.typescriptlang.org/, 2021.

[41] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, 2006.

[42] M. S. Miller, J. Paradis, C. Patiño, P. Soquet, and B. Farias. Proposal for SES (Secure EcmaScript). https://github.com/tc39/proposal-ses, 2021.

[43] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - Safe Active Content in Sanitized JavaScript, 2008.

[44] Moment Timezone: Parse and display dates in any timezone. https://momentjs.com/timezone/, 2021.

[45] J. P. Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development.* CreateSpace, 2010.

[46] node-fetch. A light-weight module that brings the Fetch API to Node.js. https://github.com/node-fetch/node-fetch, 2021.

[47] Node-RED. Community node module catalogue. https://github.com/node-red/catalogue.nodered.org, 2021.

[48] Node-RED. https://nodered.org/, 2021.

[49] Node-RED. Securing Node-RED. https://nodered.org/docs/user-guide/runtime/securing-node-red, 2021.

[50] Node-RED. the RED object. https://github.com/node-red/node-red/blob/master/packages/node_modules/node-red/lib/red.js, 2021.

[51] Node-RED. Working with context. https://nodered.org/docs/user-guide/context, 2021.

[52] Node-RED Library. https://flows.nodered.org/, 2021.

[53] Node.JS. CommonJS. https://nodejs.org/api/modules.html, 2021.

[54] Node.JS. VM (executing JavaScript). https://nodejs.org/api/vm.html#vm_vm_executing_javascript, 2021.

[55] NPM. Node Package Manager. https://www.npmjs.com/, 2021.

[56] OWASP. NodeJS security cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html#do-not-use-dangerous-functions, 2021.

[57] Peter Braden. node-opencv. https://github.com/peterbraden/node-opencv, 2021.

[58] B. Pfretzschner and L. ben Othmane. Identification of Dependency-based Attacks on Node.js. In *ARES*, 2017.

[59] reddit. The semi-official subreddit for the popular automation service IFTTT. https://www.reddit.com/r/ifttt/, 2021.

[60] P. D. Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: A survey. In *NordSec*, 2010.

[61] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 1975.

[62] D. Schreckling, J. D. Parra, C. Doukas, and J. Posegga. Data-Centric Security for the IoT. In *IoT 360 (2)*, 2015.

[63] P. Simek. Proposal for VM2: Advanced vm/sandbox for Node.js. https://github.com/patriksimek/vm2, 2021.

[64] C. Staicu, M. Pradel, and B. Livshits. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*, 2018.

[65] 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 *WWW*, 2017.

[66] Tom Van Cutsem. Membranes in JavaScript. https://tvcutsem.github.io/js-membranes, 2012.

[67] Tom Van Cutsem. Isolating application sub-components with membranes. `https://tvcutsem.github.io/membranes`, 2018.

[68] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *CHI*, 2014.

[69] S. Van Acker and A. Sabelfeld. JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript. In *FOSAD*, 2016.

[70] N. van Ginkel, W. D. Groef, F. Massacci, and F. Piessens. A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries. *Secur. Commun. Networks*, 2019.

[71] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[72] VM2. Breakout reports on VM2. `https://github.com/patriksimek/vm2/issues?q=is%3Aissue`, 2021.

[73] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter. Charting the Attack Surface of Trigger-Action IoT Platforms. In *CCS*, 2019.

[74] Zapier. `https://zapier.com`, 2021.

[75] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security*, 2019.

Appendix

1.A Node-RED empirical study

We provide the details on trust propagation, present a security labeling of sources and sinks, and discuss exploiting shared resources.

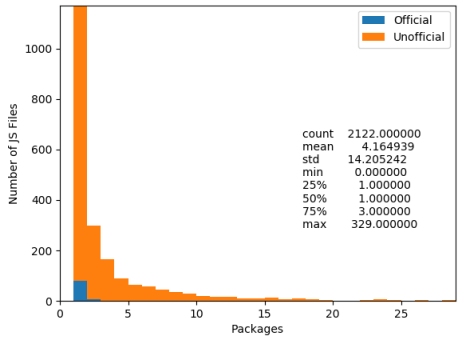
1.A.1 Trust propagation

Figures 5a and 5b illustrate the distribution of JavaScript files and lines of code from our dataset of 2122 packages. Our analysis shows that packages may contain complex JavaScript code. For example, we find nodes with 329 JavaScript files containing a total of 129,231 lines of code.

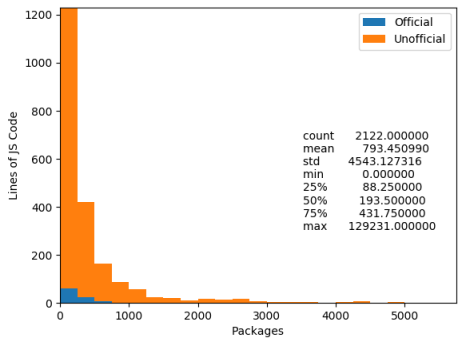
To understand the prevalence of sensitive APIs, we study the libraries included in Node-RED packages and first-party modules used in `require` statements. On average, a package has 1.85 direct dependencies on other Node.js packages, while Node-RED nodes do not typically use service-specific APIs (see Figure 5c and 6a). Specific services appear in the 23rd and 25th most popular entries, respectively *aws-sdk* and *node-red*. We draw the same conclusion while analyzing first-party modules included in `require` statements (Figure 6b). Popular package dependencies relate to resources such as HTTP requests (`request`) and other developer tools. This indicates that Node-RED is mainly focused on low-level customizable automation. More importantly, Node-RED provides access to powerful APIs that deal with the filesystem (`fs`), HTTP requests (`request`), OS features (`os`), thus enabling a malicious developer to compromise the security of users and devices.

1.A.2 Security labeling

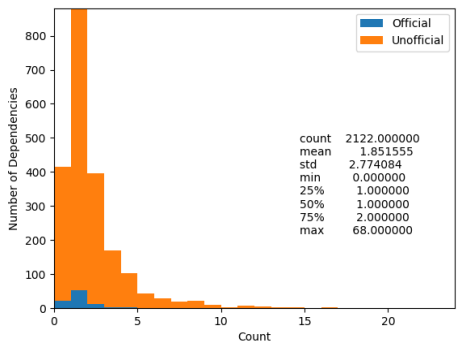
Following the approach used by Bastys et al. [8] for IFTTT, we estimate the impact of attacks on Node-RED. We manually inspect the sources and sinks of the top 100 Node-RED packages to assign a security labeling.



(a)



(b)



(c)

Figure 5: (a) JavaScript files per Node-RED package; (b) JavaScript lines of code (LoC) per Node-RED package; (c) Node.js dependencies per Node-RED package.

We label a node's sources as either private, public, or available representing node inputs with public, private, and available information. The latter contains public information, but the availability of this information is valu-

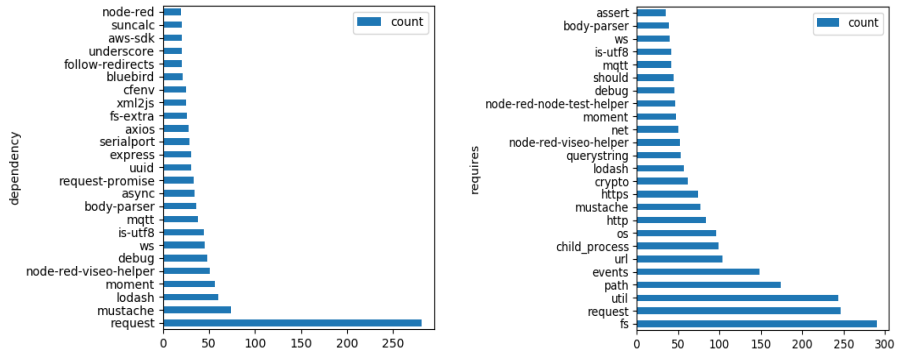
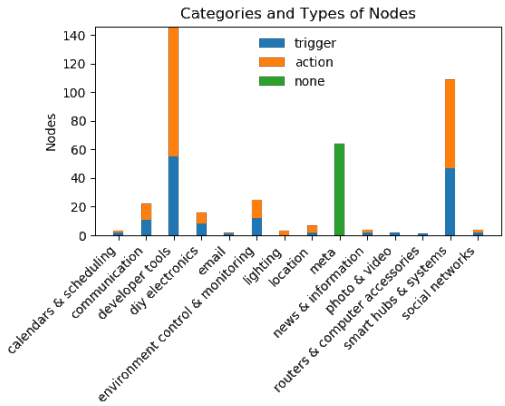


Figure 6: (a) Top 25 Node.js dependencies in Node-RED; (b) Top 25 *require* modules in Node-RED.

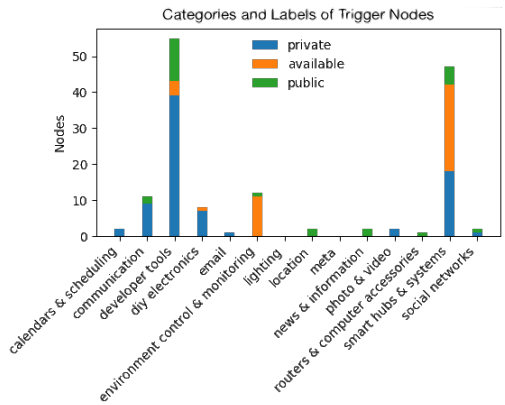
able to the user. Similarly, we label sinks as either public, untrusted, or available. Available sinks are those that will cause availability issues whenever the data is not delivered through the sink. Untrusted sinks may affect the integrity of the output, while public sinks can communicate with the public and can affect privacy. The labeling of sinks is cumulative; namely, a public sink is untrusted and available, and an untrusted sink is also available.

We also categorize node sources and sinks to understand the target domains of Node-RED applications, as well as to estimate the prevalence of attacks in these domains. While IFTTT already provides such categorization, we manually explore Node-RED nodes to classify their target domains. We assign nodes to categories by a series of steps: (i) reading the nodes' documentation, (ii) running the node in a flow, and (iii) manually reading the code defining the node. Figure 7a reports the categorization of Node-RED nodes in our dataset.

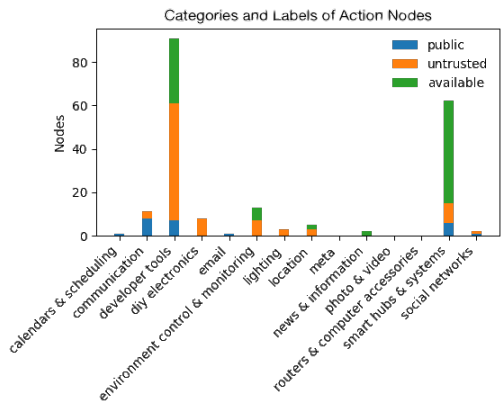
We conduct an empirical analysis of 408 node definitions for the top 100 Node-RED packages. We follow a set of heuristics to assign labels to nodes in a conservative manner. For example, a node that sends output to a Raspberry Pi's pins can be used in driving electronics like LEDs and motors; hence we label the sink as untrusted. These output pins can also be used for communicating with Internet-connected devices to exfiltrate data; hence we label the sink as public. Other general guidelines include labeling output to local as available, input from local as private, and databases as private and untrusted. Figures 7b and 7c illustrate our labeling for sources and sinks. Compared to the results of Bastys et al. [8] for IFTTT, we observe that Node-RED targets custom-built flows for nodes with low-level functionality. In fact, the major-



(a)



(b)



(c)

Figure 7: (a) Categorization of Node-RED sources and sinks; (b) Security labeling for Node-RED sources; (c) Security labeling for Node-RED sinks.

ity of nodes in our categorization belongs to the "developer tools" and "smart hubs & systems" categories.

Our analysis covers the top 100 packages, representing only 4.71% of all Node-RED packages and 7.67% percent of all node definitions (from 5316 total). This labeling completely covers 642 flows (54.36% of the 1181 total flows after pruning invalid flows), which we consider further in our experiment. We find possible security violations by tracing the graph for the descendants of source nodes and looking for the labels of these sink node descendants.

We find that privacy violations (private sources to public sinks) may occur in 70.40% of flows, integrity violations (any sources to available sinks) may occur in 76.46%, and availability violations (available sources to available sinks) may occur in 1.71%. A similar experiment on a dataset from IFTTT revealed 30% privacy violations, 98% integrity violations, and 0.5% availability violations [8]. The larger number of privacy violations in Node-RED reflects the power of malicious developers to exfiltrate private information.

1.A.3 Exploiting shared resources

Another usage of the context feature is to share resources such as common libraries. In addition to integrity and availability concerns, this pattern opens up possibilities for exfiltration of private data. An attacker can encapsulate the library such that it collects any sensitive information sent to this library.

For example, the flow "btsimonh's node-opencv motion detection (2017-11-02)" targets Raspberry Pi to implement a video stream for motion detection. It feeds the image frames into the computer vision library *openCV*, which is imported in the code snippet below:

```
var require = global.get('require');
...
// look for globally installed opencv
var cv = require.main.require('opencv');
if (!cv){
  // look for locally installed opencv
  cv = require('opencv');
}
...
var cvdesc = Object.keys(cv);
node.send([null, {payload:cvdesc}]);
flow.set('cv', cv);
```

The code contains two instances of disruptable libraries, `require` and `opencv`, which can be exploited by an attacker with access to the *Flow* or *Global* contexts. We find other flows that are subject to similar vulnerabilities, . We also find similar vulnerabilities in Node-RED nodes. For ex-

ample, the EmotivBCI Facial Expression node outputs the values of the trained detections originating from EMOTIV wearable brain sensing technology.

1.B Evaluation

Tables 4, 5, and 6 summarize the details of the evaluation of SandTrap in different use cases for IFTTT, Zapier, and Node-RED, respectively. In all cases, SandTrap accepts the secure version and rejects the insecure one. The time overhead is tolerable while enhancing the platforms with SandTrap.

1.B.1 IFTTT

We discuss 10 out of 25 cases of our IFTTT benchmark, pairs of benign and malicious filter code instances, and show how their executions are secured by SandTrap. In each case, we measure the time overhead compared to the original execution (without any monitor) and the time overhead compared to the deployment of IFTTT with `vm2`.

Use cases *SkipTodoistCreateTask* and *SkipNotification* are instances of filter code that skip an action during a time indicated by the user. Thus, they do not need loading modules. The baseline policy for IFTTT, i.e., disallowing any `require` calls, protects the user from exfiltration attacks through network.

The next three cases, *SkipAndroidMessage*, *SkipSendEmail*, and *Trello-SlackAndOffice365Mail* also skip an action with respect to some conditions. Since filter code enables modifying all fields of action services, an attacker can manipulate them instead of skipping the actions during the specified time. For example, private information can be sent to the attacker via setter functions, which are provided by the platform. The user's information will be sent to the attacker's phone or email address unnoticeably, while the user thinks the actions are skipped as specified. In the *Trello-SlackAndOffice365Mail* case, two action services are available and thus any modification to the fields of both is possible in the filter code. For filter code that only skips action(s), the policy should only permit `skip` calls; thus any invocation to the setter functions must be blocked.

The remaining use cases represent other patterns of filter code, in which values are passed to action services using the setter functions. For example, *Instagram-Twitter* and *Telegram-Tumblr* call `setPhotoURL` with the URL received from the trigger service. To make sure that the URL is not changed in the filter code, the policy should be value-sensitive. Thanks to the feature of parameterized policy in SandTrap, the user can specify dy-

Filter code	Specification	Granularity	O/H	O/H (vs. vm2)		Example of Prevented Attacks
				Creation	Eval	
SkipTodoistCreateTask	Skip creating a task in non-working hours	Module	4.11	0.44	0.29	Exfiltrate the content of task
SkipNotification	Skip IFTTT notifications on weekends	Module	4.14	0.58	0.51	Exfiltrate the content of notification
SkipAndroidMessage	Skip sending a message in non-working time	API	4.22	0.56	0.41	Set phone number to the attacker's number instead of skip
SkipSendEmail	Skip sending email notifications during weekends	API	3.85	0.52	0.35	Set recipient to the attacker's address instead of skip
Trello-SlackAndOffice365Mail	Skip posting Trello cards not including specific keyword to Slack; otherwise also send an email	API	4.24	0.62	0.38	Modify other properties of Trello cards such as ListName
Instagram-Twitter	Tweet a photo from an Instagram post	Value	4.17	0.64	0.40	Tamper with the photo URL
Webhook-AndroidDevice	Set volume for an android device	Value	4.17	0.75	0.36	Tamper with the volume
Telegram-Tumblr	Post a new Telegram channel photo to Tumblr	Value	4.06	0.55	0.45	Tamper with the photo URL
Life360-Dropbox	Upload a text file someone arrived at home	Value	3.99	0.43	0.47	Tamper with the filename
GoogleCalendar-iOSCalendar	Set the duration based on the start and end time	Value	4.07	0.54	0.25	Tamper with the duration

Table 4: IFTTT benchmark evaluation. We report the filter code specification, the policy granularity, the time overhead of the monitored secure run in milliseconds, the time overhead of the monitored secure run compared to `vm2` in two stages of sandbox creation and code evaluation in milliseconds, and the attack implemented and blocked by SandTrap.

namic policies with respect to the trigger data (e.g., the URL is accessible by `this.GetPolicyParameter('SourceUrl')`). The effort for tuning the policies is minimal; the call policy of the setter function should be updated to a JavaScript function that verifies the passed argument to be consistent with the trigger data.

Because the filter code cannot load any node modules, the time overhead is relatively constant (on average 4.10ms), which is tolerable given that IFTTT apps are allowed up to 15 minutes to execute. We also compare how `vm2` and SandTrap affect the execution time of the use cases and report on the added time by SandTrap compared to `vm2`. We split the time overhead into two stages: sandbox creation and filter code evaluation. The difference for each stage is always less than 1ms. Since IFTTT employs `vm2` already, it seems reasonable to upgrade the filter code evaluation module to employ SandTrap, thus bringing in a fine-grained security mechanism with negligible runtime overhead.

1.B.2 Zapier

We executed 10 different pairs of secure and insecure Zapier code under SandTrap. Unlike IFTTT, a list of node modules is available for the user code including the built-in modules. The first two cases *SimpleOutput* and *StringFilter* do not require any node module to run; hence a policy that denies loading any module is sufficient. Use case *AllBuiltinModules* (loading all built-in modules) is a crafted example to show that the time overhead incurred by SandTrap is tolerable even if the user code requires all built-in modules.

The two cases *Url-and-http* and *Os-info* need interaction with specific node modules (e.g., `url`, `http` and `os`) and their APIs. Hence, any other API calls like `os.userInfo()` must be stopped by the monitor. An auto-generated policy, without any additional effort, that lists all legal APIs of each node module is sufficient for SandTrap to enforce the desired security.

Use cases *SetStoreClient* and *FetchGet* employ the accessible objects `StoreClient` and `node-fetch`, respectively. These are the objects directly passed to the Zapier runtime environment to enable users to communicate data through network via the `node-fetch` object. Malicious code might exfiltrate sensitive data or affect the integrity of data stored in StoreClient, a utility to store and retrieve data. Similarly, the last three use cases demonstrate value-dependent policies. The policy for the use case `Fs-readdirsync` allows listing files in the current directory or nested ones, but it blocks browsing other directories like the parent directory, which in the Zapier environment contains the source code of the runtime. The cases *ImageWatermark* and *TrelloChecklist* are examples that use the `node-fetch` module to communicate

Zapier code	Specification	Granularity	O/H	Example of Prevented Attacks
SimpleOutput	Assign an object to the output variable	Module	3.86	Access to $PATH using child_process
StringFilter	Extract a piece of text of a long string	Module	4.32	Exfiltrate filtered string
AllBuiltinModules	Load all built-in modules	Module	6.80	Load any external module
Url-and-http	Parse a URL and list the http status codes	API	5.10	Create an http server
Os-info	Get platform and architecture of the host OS	API	5.38	Get hostname and userInfo
SetStoreClient	Set a specific property to the stored object	Value	4.08	Add a new property to the object in StoreClient
FetchGet	Get a JSON object using 'fetch'	Value	5.21	Exfiltrate the object via 'fetch'
Fs-readdirsync	List files of the current directory or nested ones	Value	4.80	List files of the parent directory
ImageWatermark	Create a watermarked image using Cloudinary	Value	4.55	Exfiltrate the link to the watermarked image
TrelloChecklist	Add a checklist item to a Trello card	Value	4.58	Exfiltrate the checklist data

Table 5: Zapier benchmark evaluation. We report the code specification, the policy granularity, the time overhead of the monitored secure run in milliseconds, and the attack implemented and blocked by SandTrap.

through HTTP requests. Any requests except for the ones needed for the functionality of the code should not be listed in the policies.

1.B.3 Node-RED

We evaluate the security and performance of SandTrap on a set of 20 Node-RED flows, 10 flows with secure nodes and 10 flows with malicious nodes.

For diversity, we have selected flows with nodes from both popular and less popular packages in terms of the number of downloads. Table 6 summarizes our experimental findings. Each row represents the use case of a flow, which is instantiated to a flow with secure nodes and a flow with a malicious node. For each use case, we report the flow name, the specification of flow behavior, the package and node identifier of the essential node, the number of package downloads, the granularity of the desired security policy (module-, API-, value-, or context-level), execution time overhead of the secure flow under the monitor in milliseconds, and the explanation of attack implemented by the malicious node and blocked by the monitor.

We briefly discuss experiments for each use case. The nodes in the first two cases, *Lowercase* and *Thermostat*, should be fully isolated as they do not need to interact with the environment. Therefore, the right policy is the coarse-grained module-level deny-all policy (which SandTrap implements by making `require` unavailable). The *Lowercase* node converts the input `msg.payload` to lower case and sends the result object to the output. In the attack scenario, the malicious node attempts to read the content of `/etc/passwd` by calling `fs.readFile`, and send the sensitive data to the attacker's server via `https.request`. Because the policy does not allow any libraries to get required in the node, the monitor blocks the execution once the first `require` is called.

In the thermostat use case, the *Thermostat* node gets a temperature input and switches the heater status depending on the defined low and high limits. Similar to the lowercase node, it does not require any node modules by nature. The attack exfiltrates the input temperature and the heater status, which we consider sensitive information. The monitor prevents the leakage because `https` module is not in the baseline allowlist policy.

The File and Dropbox cases rely on libraries and thus require API-level policies. The *File* node, one of the core nodes of Node-RED, writes the content of the input message `msg.payload` to a file specified by the user. Therefore, the `fs` module should be allowed by the policy, and it is indeed inferred by SandTrap's policy auto-generation feature. As an attack scenario, we added a single line `fs.rmdir("./", {recursive: true})` that removes the current directory, i.e., the Node-RED directory. The monitor rightfully blocks the exe-

Flow	Specification	Package:Node	Downloads	Granularity	O/H		Example of Prevented Attacks
					Load	Trigger	
Lowercase	Convert input to lowercase letters	node-red-contrib-lower :lower-case	5,842	Module	0.15	0.23	Send the content of '/etc/passwd' to the attacker's server
Thermostat	Switch heater on or off depending on the temperature	node-red-contrib-basic-thermostat:thermostat	303	Module	0.14	0.10	Exfiltrate the heater status and the temperature
File	Write input to file	node-red:file	core node	API	12.65	0.32	Remove the Node-RED directory
Dropbox	Upload file	node-red-node-dropbox :dropbox out	68,421	API	1.42	0.08	Exfiltrate file name and content
Calendar	Add event into calendar	node-red-contrib-google -calendar:addEvent	2,998	Value	30.91	1.38	Exfiltrate the calendar event
Email	Send input to specified email address	node-red-node-email :email	2,397,312	Value	30.25	0.29	Forward a copy of the message to the attacker's email address
Earthquake	Get earthquake data from specified URL	node-red:http request	core node	Value	9.33	2.10	Tamper with the specified URL
Baby monitor	Send alarm notification to SMS server	node-red:http request	core node	Value	9.33	2.10	Send the notification to the attacker's server
Water utility	Water supply network	n/a	n/a	Context	n/a	n/a	Tamper with the status of tanks and pumps (stored in the global context)
Motion detection	Motion detection by openCV	n/a	n/a	Context	n/a	n/a	Manipulate the 'require' object (stored in the global context)

Table 6: Node-RED benchmark evaluation. We report the specification of flow behavior, package and node identifier of the essential node, the number of package downloads, the policy granularity, the time overhead of the monitored secure run in milliseconds separated in the two stages of loading and triggering the node, and the attack implemented and blocked by SandTrap.

cution of the malicious node because the node policy introduces a subset of allowed `fs` functions, where `fs.rmdir` is not included.

Similarly, the *Dropbox out* node requires `https` to establish a connection with the user-defined Dropbox account and upload the specified file. We maliciously altered the code to transmit the file name and its content to the attacker's server via `https.request.write`. SandTrap blocks the exfiltration by restricting `https.request.write` calls, while `https.request` is a prerequisite for the node behavior.

In the calendar use case, the users add events to their Google calendar. A malicious modification of the *addEvent* allows passing the event data to the attacker's server. Note that the node demands communication with the Google API via the same function calls. Value-dependent policies enable us to include a fine-grained allowlist policy that restricts `https.request` from connecting to servers other than `"www.googleapis.com"`. As in the other cases, SandTrap accepts the secure version and rejects the insecure one.

In the email case, the *Email* node sends a user-defined message from one email address to another, where both are provided by the user. The attacker modifies the node so that a copy of each message is transmitted to the attacker's email address by using the same `sendMail` function of the same `SMTP` object. SandTrap blocks this attack because the value-level policy delimits `stream.Transform.write` calls to the user-specified recipient.

The *Earthquake* and *Baby monitor* flows employ *http request*, a general-purpose core node of Node-RED for setting up HTTP communication channels. The earthquake flow retrieves a list of significant earthquakes from the US Geological Survey website and outputs notifications for the ones with magnitudes greater than seven. A malicious node maker manipulates the user-defined URL in the source code of the node to perform an integrity attack. In the Baby monitor case, the node sends a request to an SMS server when an emergency occurs to the baby. The attacker is able to act as a person-in-the-middle, read the sensitive data, and falsify the status. We address this by making the `call` attribute of `url.parse` function in the policy value-dependent, which enforces the integrity of the URL.

The last cases use the global and flow contexts in their implementation, as discussed in Section 4.3. The *Water utility* flow reads and updates the status of water pumps and tanks using globally shared variables. Any tampering with the values of those variables may cause serious effects on the behavior of the water supply network. The *Motion detection* flow utilizes the `opencv` [57] module to enable a Raspberry Pi process images taken from the environment. To load `opencv` in a Function node, it obtains the `require` object from the global context. We do not report on concrete nodes or running

times because they would depend on the choice of a malicious node. Note that any node can maliciously alter the globally shared object in the original Node-RED setting. SandTrap blocks any change on the global and flow contexts by default, disallowing `_context.global.set` and `_context.flow.set` to be called.

The overhead columns of the table present the additional amount of elapsed time in the two phases of node execution, i.e., loading and triggering, in comparison with the original execution without the monitor. We report the average overhead of 20 runs for each secure node. Note that the Earthquake and Baby monitor flows use the same `http request` node, which explains the same reported overhead (9.33ms and 2.10ms for loading and triggering the node). The time overhead is unnoticeable to users in the setting of TAPs where the significant performance costs are incurred by network communication and file/device access.

2

Securing Node-RED Applications

Mohammad M. Ahmadpanah, Musard Balliu, Daniel Hedin, Lars Eric Olsson, and Andrei Sabelfeld

Protocols, Logic, and Strands: Festschrift in honor of Joshua Guttman'21

Abstract. Trigger-Action Platforms (TAPs) play a vital role in fulfilling the promise of the Internet of Things (IoT) by seamlessly connecting otherwise unconnected devices and services. While enabling novel and exciting applications across a variety of services, security and privacy issues must be taken into consideration because TAPs essentially act as persons-in-the-middle between trigger and action services. The issue is further aggravated since the triggers and actions on TAPs are mostly provided by third parties extending the trust beyond the platform providers.

Node-RED, an open-source JavaScript-driven TAP, provides the opportunity for users to effortlessly employ and link *nodes* via a graphical user interface. Being built upon Node.js, third-party developers can extend the platform's functionality through publishing nodes and their wirings, known as *flows*.

This paper proposes an essential model for Node-RED, suitable to reason about nodes and flows, be they benign, vulnerable, or malicious. We expand on attacks discovered in recent work, ranging from exfiltrating data from unsuspecting users to taking over the entire platform by misusing sensitive APIs within nodes. We present a formalization of a runtime monitoring framework for a core language that soundly and transparently enforces fine-grained allowlist policies at module-, API-, value-, and context-level. We introduce the monitoring framework for Node-RED that isolates nodes while permitting them to communicate via well-defined API calls complying with the policy specified for each node.

1 Introduction

Trigger-Action Platforms (TAPs) play a vital role in fulfilling the promise of the Internet of Things (IoT). TAPs empower users by seamlessly connecting otherwise unconnected *trigger* and *action* services. Popular TAPs like IFTTT [23] and Zapier [56], as well as open-source alternatives like Node-RED [35], offer users the ability to operate simple trigger-action *applications* (or, for short, *apps*) such as "Tweet your Instagrams as native photos on Twitter", "Get emails via Gmail with new files added to Dropbox", and "Covid-19 live Ticker via Twitter".

A TAP is effectively a "person-in-the-middle" between trigger and action services. While greatly benefiting from the possibility of apps to run third-party code, TAPs are subject to critical security and privacy concerns. Attacks by third-party app makers on the platform may lead to compromising the integrated trigger and action services. Figure 1 illustrates how a malicious app deployed by a user on a TAP like Node-RED can compromise the associated trigger and action services, another installed app, and the platform [2]. Depending on the security configuration of the TAP's deployment, the attacker may also compromise the underlying system.

In contrast to proprietary centralized platforms such as IFTTT and Zapier, Node-RED can be entirely run on a user's own server. Node-RED is an open-source platform built on top of Node.js, enabling users to inspect and

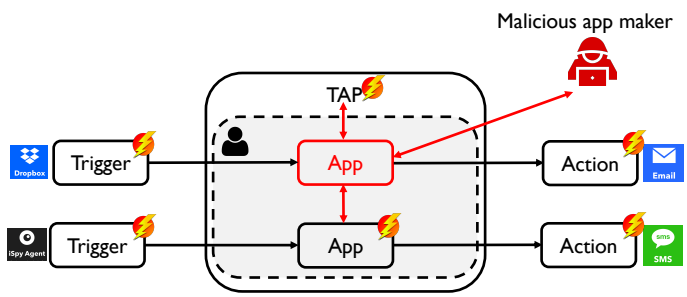


Figure 1: Threat model of a malicious app deployed on a single-user TAP [2].

customize the source code of the platform and the apps as desired. Moreover, Node-RED relies on JavaScript packages from third parties to facilitate the integration of new functionalities. In fact, Node.js *nodes* are the basic building blocks of Node-RED apps (also named as *flows*), freely available on the Node Package Manager (NPM) [42] and automatically added to the Node-RED Library [40]. Node-RED is inspectable and thus can be verified by users in terms of the platform's correctness and security. Third-party apps integrated into the underlying platform, however, can still threaten the security of the users and the entire system.

The starting point of this paper is the recently identified attacks on Node-RED by malicious nodes, ranging from exfiltrating users' sensitive data to taking over the platform and the host system [2]. A Node-RED flow is technically a static representation of how nodes are wired together; therefore, a malicious node controlled by an attacker can be employed in any user-defined or third-party flows, resulting in malicious behaviors.

This observation motivates the need for controlling APIs invoked in nodes to ensure the security of the platform and the users. Although the enforcement mechanism must guarantee security, it also should restrict access only if it is against the node's policy, according to the *least privilege principle* [46]. Only the APIs which are necessary for the intended functionality should be accessible in a node; thus, if none of the APIs of a module are required, loading of the module must be denied. In some cases, the interaction through APIs needs to be value-sensitive when an API call should be permitted only with a list of defined arguments, for instance, when it comes to allowing a node to make an HTTPS request to a specific trusted domain. Furthermore, Node-RED makes use of both message passing and the shared context [39] to exchange information between nodes and flows, and both types of exchange need to be secured. Previous work proposes SandTrap [2], a runtime monitor for JavaScript-driven TAPs. However, SandTrap's security guarantees are argued only informally.

Motivated by SandTrap, this work is a step toward formally understanding how to monitor Node-RED apps. We present a sound and transparent monitoring framework for Node-RED for enforcing fine-grained allowlist policies at module-, API-, value-, and context-level. In the following, we discuss Node-RED along with overviewing platform- and app-level vulnerabilities and attacks (Section 2); propose an essential model for Node-RED, suitable to reason about nodes and flows, be they benign, vulnerable, or malicious; and present a monitoring framework to express and enforce fine-grained security policies, proving its soundness and transparency (Section 3).

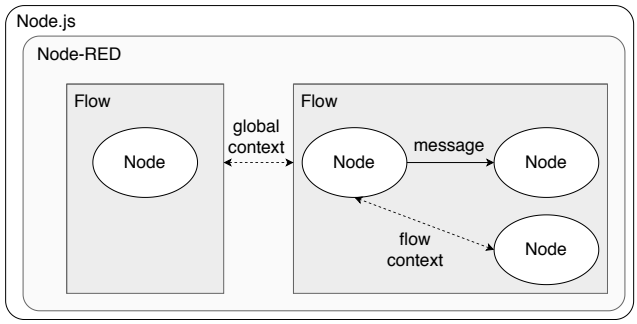


Figure 2: Node-RED architecture [2].

2 Node-RED Vulnerabilities

Node-RED is "a programming tool for wiring together hardware devices, APIs and online services", which provides a way of "low-code programming for event-driven applications" [35]. As an open-source platform, Node-RED is mainly targeted for deployment as a single-user platform, although it is also available on the IBM Cloud platform [22]. We overview the architecture of Node-RED (Section 2.1) and explain two types of vulnerabilities with respect to our attacker model, i.e., malicious app makers: (i) *platform-level isolation vulnerabilities* (Section 2.2) and (ii) *application-level context vulnerabilities* (Section 2.3). Our discussion expands the condensed presentation of these vulnerabilities from previous work [2].

2.1 Node-RED platform

Figure 2 illustrates the Node-RED architecture, consisting of a collection of apps, known as *flows*, linking components called *nodes*. The Node-RED runtime is built on the Node.js environment and can run multiple flows simultaneously. It supports inter-node and inter-flow communication via direct messages through the wiring between nodes in a flow, while the *flow* and the *global* contexts [39] are alternative communication channels between the nodes of a flow and across the nodes of different flows, respectively.

A node is a reactive Node.js application triggered by receiving messages on at most one input port (dubbed *source*) and sending the results of (side-effectful) computations on output ports (dubbed *sinks*), which can be potentially multiple, unlike the input port. Figure 3 illustrates the code structure of a Node-RED node. A special type of node without sources and sinks, called *configuration* node, is used for sharing configuration data, such as login credentials, between multiple nodes.

```
module.exports = function(RED){
  function NodeName(config){
    RED.nodes.createNode(this, config);
    var node = this;
    // register a callback when a message is received...
    node.on("input", function(msg){
      ... // functionality of node
      node.send(msg); // or an array of messages for multiple outputs
    });
  }
  RED.nodes.registerType("type-name", NodeName);
}
```

Figure 3: Node-RED node structure.

A flow is a representation of nodes connected together. End users can either create their own flows on the platform's environment or deploy existing flows provided by the official Node-RED catalog [32] and by third parties [40]. As shown in Figure 4, flows are JSON files wiring node sinks to node sources in a graph of nodes where messages, represented by JavaScript objects, are passed between. Multiple messages can be sent by any given node, although instances of a single message can be repeatedly sent to multiple nodes as well. To facilitate end-user programming [54], flows can be shown visually via a graphical user interface and deployed in a push-button fashion. As an example, Figure 5 demonstrates a flow that retrieves earthquake data for logging and notifying the user whenever the magnitude exceeds a threshold. Specifically, the flow retrieves data of the recent quakes (either periodically or by clicking on the button), parses the given CSV file, and shows the data (stored in `msg.payload`) to the user. For each magnitude value exceeding the specified threshold, it also branches and the payload triggers an alarm notification.

In Node-RED, *contexts* provide a shared communication channel between different nodes without using the explicit messages that pass through a flow [39]. Therefore the node wiring visible in the user interface reflects only a part of the information flows that are possible in the flow. It introduces an implicit channel that is not visible to the user via the graphical interface of a flow. Node-RED defines three scope levels for the contexts: (i) *Node*, only visible to the node that sets the value, (ii) *Flow*, visible to all nodes on the same flow, and (iii) *Global*, visible to all nodes on any flow. For instance, a sensor node may regularly update new values in one flow, while another flow may return the most recent value via HTTP. By storing the sensor reading in the global shared context, the data is accessible for the HTTP flow to return.

```
[                       // list of nodes
  {                     // node 0
    /* parameters of interest in every node */
    id: NODE0,          // unique ID of node, string
    type: function      // type of node, string
    wires: [            // array of array of strings
      [ NODE1 ],        // first output port to node 1
      [ NODE2, NODE3 ] // second output port to nodes 2 and 3
    ],
    ...                 // other parameters
  },
  ...                   // other nodes
]
```

Figure 4: Node-RED flow structure.

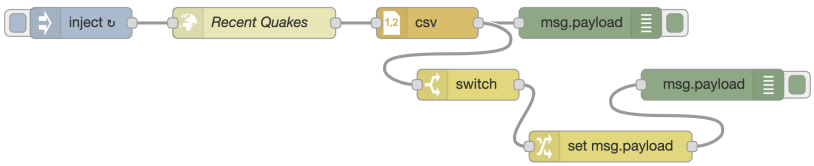


Figure 5: Earthquake notification and logging.

Node-RED security relies on the platform running on a trusted network, ensuring that users' sensitive data is processed in an environment controlled by the users. The official documentation [36] also includes programming patterns for securing Node-RED apps. These patterns include basic authentication mechanisms to control access to nodes and wires. The official node `Function` runs user-provided code in a `vm` sandbox [41], suggesting that it may protect the user from unauthorized access. However, the `vm`'s sandbox "is not a security mechanism" [41], and there are known breakouts [25].

TAPs generally lack the means to specify user's security policies [8]. Fortunately, Node-RED's user-centric setting enables us to *interpret* intended security policies. In fact, Node-RED's GUI for flows provides an intuitive way to interpret top-level user policies; it is reasonable to consider that the user endorses the flow of information between the nodes connected by the graph that depicts a flow in the GUI. For instance, the Earthquake notification flow in Figure 5 implies a policy where notification data may only flow to the notification message. Only the `Inject` node can trigger updates. The policy allows no other node (from any flow) to tamper with the `Recent Quakes` node, preventing any malicious node from corrupting the source of quake information. Such an interpretation provides us with a *baseline* security pol-

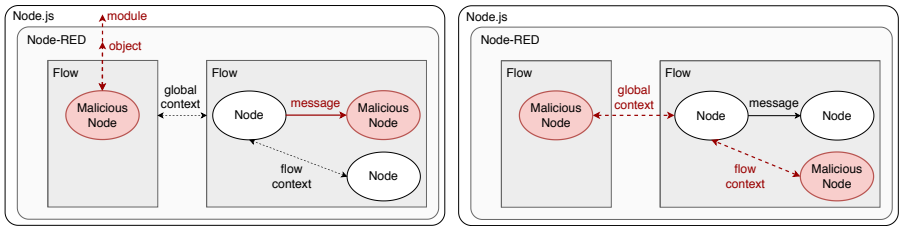


Figure 6: Node-RED vulnerabilities: (a) Isolation vulnerabilities; (b) Context vulnerabilities [2] .

icy. For more fine-grained policies, e.g., the list of permitted URLs to retrieve the recent quakes, it is reasonably presumed that the node developer designs these *advanced* policies since they know the precise specification of the node. The provided policies can later be vetted by the platform and the user, before deploying the node. SandTrap [2] offers a policy generation mechanism to aid developers in designing the policies, enabling both baseline and advanced policies customized by developers or users to express fine-grained app-specific security goals.

In the following, we discuss Node-RED attacks and vulnerabilities that motivate enriching the policy mechanism with different granularity levels. These policies will further be formalized in Section 3.

2.2 Platform-level isolation vulnerabilities

While facilitating the integration and automation of different services and devices, due to imposing insufficient restrictions on nodes, Node-RED is exploitable by malicious node makers. All APIs provided by the underlying runtimes, Node-RED and Node.js, are accessible for node developers, as well as the incoming messages within a flow. As shown in Figure 6a, there are various attack scenarios for malicious nodes [2]. At the Node.js level, an attacker can create a malicious Node-RED node including powerful Node.js libraries like `child_process`, allowing the attacker to execute arbitrary shell commands with `exec`, e.g., taking full control of the user's system [43]. Restricting library access is laborious in Node-RED; while access to a sensitive library like `child_process` is required for the functionality of Node-RED, attackers can exploit trust propagation due to transitive dependencies in Node.js [44, 57]. A malicious node enables the attacker to compromise the confidentiality and integrity of sensitive data and libraries stored by other flows in the global context. A malicious node within a sensitive flow may also indirectly read

and modify sensitive data by manipulating the flow context.

At the platform level, the main object in the Node-RED structure, named `RED` [38], is also vulnerable. There are different ways for a malicious node to misuse the `RED` object, such as aborting the server (e.g., by `RED.server._events = null`) or introducing a covert channel shared between multiple instances of the node in different flows by modifying existing properties or adding new properties to the `RED` object (like `RED.dummy`). Therefore, *access control at the level of modules and shared objects* is necessary for Node-RED nodes.

On the other hand, a malicious node can directly manipulate incoming messages resulting in accessing or tampering with the sensitive data. As a subtle example of this scenario to invade users' privacy, the official Node-RED email can be modified to send the email body to the original recipient and also forward a copy of the message to an attacker's address. The benign code sets the sending options `sendopts.to` to contain only the address of the intended recipient:

```
sendopts.to = node.name || msg.to; // comma separated list of addresses
```

It can be modified to the following by a malicious node maker to include the attacker's address as well:

```
sendopts.to = (node.name || msg.to) + ", me@attacker.com";
```

In this example, we demonstrate that an attacker can alter the value that is placed as the argument of an API call, which is necessary for the functionality of the node, to steal sensitive information of the user without being noticed. As a result, the combination of function identity and its arguments needs to be considered in security checks. This attack motivates us to provide *fine-grained access control at the level of APIs and their input parameters.*

We refer the interested reader to other types of investigated vulnerabilities in Node-RED [2], such as the impact of compromised package repository and *name squatting* [57] attack. The latter is critical since the "type" of nodes (what flows use to identify them) is simply a string, which multiple packages can possibly match. A flow defined by a third party can include the attacker's malicious node unless the user inspects each and every node to verify that there are no deviations from the expected "type" string.

The empirical study shows the implications of such attacks [2]: privacy violations may occur in 70.40% of Node-RED flows and integrity violations in 76.46%. The vast number of privacy violations in Node-RED reflects the power of malicious developers to exfiltrate private information.

2.3 Application-level context vulnerabilities

Node-RED uses various levels of the shared context to exchange data across nodes and flows in an implicit manner. Figure 6b depicts the attack scenarios to exploit vulnerabilities by reading and writing to libraries and variables shared in the global and flow contexts [2]. The *Node* context shares data with the node itself; thus only the shared contexts at the levels of *Flow* and *Global* are intriguing to investigate. Malicious nodes in these scenarios can exploit other vulnerable Node-RED nodes, even if the platform is secured against attacks in Section 2.2.

Several Node-RED core nodes [37] make use of the shared context for their purposes, including the nodes executing any JavaScript function (`Function`), triggering a flow (`Inject`), generating text to fill out a template (`Template`), routing outgoing messages to branches of a flow by evaluating a set of rules (`Switch`), and modifying message properties and setting context properties (`Change`). It is shown that more than 228 published flows utilize flow or global context in at least one of the member nodes and more than 153 of the published Node-RED packages directly read from or modify the shared context [2].

The main purpose of using the shared context is data communication between nodes. Malicious operations on the shared data, such as tampering, adding, or erasing, may lead to integrity and availability attacks, as well as to disrupting the functionality entirely. As a real-world example, the Node-RED flow "Water Utility Complete Example" is vulnerable considering misuse of the *Global* context. Targeting SCADA systems, this flow manages two tanks and two pumps; the first pump pumps water from a well into the first tank, and the second pump transfers water from the first to the second tank. The status of the tanks are stored in globally shared variables as follows:

```
global.set("tank1Level", tank1Level);
global.set("tank1Start", tank1Start);
global.set("tank1Stop",  tank1Stop);
```

Later, to determine whether a pump should start or stop, the flow retrieves the shared status from the *Global* context:

```
var tankLevel  = global.get("tank1Level");
var pumpMode   = global.get("pump1Mode");
var pumpStatus = global.get("pump1Status");
var tankStart  = global.get("tank1Start");
var tankStop   = global.get("tank1Stop");
if (pumpMode  ===  true && pumpStatus  ===  false && tankLevel <=
    tankStart){
  // message to start the pump
}
```

```
else if (pumpMode === true && pumpStatus === true && tankLevel >=
    tankStop){
  // message to stop the pump
}
```

A malicious node installed by the user and deployed in the platform could alter the context relating to the tank's reading to either exhaust the water flow (never start) or cause physical damage through continuous pumping (never stop).

One can also use the context feature to share resources such as common libraries. In addition to integrity and availability concerns, this approach opens up possibilities for exfiltrating private data. An attacker can encapsulate a library to collect any sensitive information sent to the library. For instance, by modifying the `opencv` shared library inside a malicious node, the attacker can exfiltrate private information of video streaming for motion detection. More details and examples of such vulnerabilities are also studied [2].

These vulnerabilities motivate the need for monitoring *access control at the level of context*.

3 Formalization

Section 2 motivates the need for secure integration of untrusted code in general and restricting node-to-node and node-to-environment communications (i.e., between nodes, library functions, and contexts) for Node-RED in particular. To achieve this, we propose a runtime monitoring framework capable of enforcing allowlist policies at the granularity of modules, APIs and their input parameters, and variables used in the shared context. Our runtime framework formalizes the core of the flow-based programming model of Node-RED and was the basis when developing the JavaScript monitor SandTrap [2].

This section presents a security model for Node-RED apps and characterizes the essence of a fine-grained access control monitor for the platform. We show how to formalize and enforce security policies for nodes at the level of APIs and their values, along with the access rights to the shared context. Our main formal results are the soundness and transparency of the monitor.

3.1 Language syntax and semantics

3.1.1 Syntax

We define a core language to capture the reactive nature of nodes and flows. Nodes are reactive programs triggered by input messages to execute the code of an event handler and potentially produce an output message. Flows model connections between nodes by specifying the destination nodes for each node's output port. Given the set of member nodes with their handlers, it is sufficient to state the successor nodes on each output port to construct a flow.

A flow is syntactically defined as a set of nodes, written $F = \{N_k \mid k \in K\}$, where K is a finite subset of $\mathbb{N}$, and k indicates a unique node identifier. A Node-RED environment may execute flows simultaneously and the global environment is defined by a set of flows, written $G = \{F_l \mid l \in L\}$, where L is a finite subset of $\mathbb{N}$, and l denotes a unique flow identifier. Based on a generalization of Node-RED nodes, Figure 7 presents the syntax of a reactive language inspired by Devriese and Piessens [16], where Val, Var, and Fun denote the set of all possible values, variables, and functions, respectively. A handler $handler\,(x)\{c\}$ is defined by an input parameter x, which is bound in a command c to perform a computation. While most commands are standard imperative constructs, we use command $send(e, i)$ to pass the value of expression e to the node's output port identified by i. For simplicity, we use functions $f(e)$ to model module imports, API calls, user-defined functions, and primitive operations such as addition and concatenation. To model the shared context, we distinguish between *node* variables Var_N, *flow* variables Var_F, and *global* variables Var_G such that $Var = Var_N \uplus Var_F \uplus Var_G$.

$v \in Val, x \in Var, f \in Fun, i \in \mathbb{N}$
$e ::= v \mid x \mid f(e)$
$c ::= skip \mid x := e \mid if\ e\ then\ c\ else\ c \mid while\ e\ do\ c \mid c; c \mid send\ (e, i)$
$h ::= handler\,(x)\{c\}$

Figure 7: Syntax of node handlers.

3.1.2 Semantics

We model the execution of Node-RED apps by defining the node semantics, flow semantics, and global semantics, respectively. Our trace-based semantics records the sequence of events produced during the execution of a flow. We use these events to define a semantic security condition that our monitor will enforce in a sound and transparent manner.

Node Semantics A node $N = \langle config, wires, l \rangle_k$ is defined by a node configuration $config$, an array $wires$ that specifies the connected nodes in the flow associated with output ports, an identifier l that indicates the flow that the node belongs to, and a unique node identifier k. Index k refers to an element of node N_k, as in $config_k$ for the configuration of node k.

A node configuration $config = \langle c, M, I, O \rangle$ stores the state of the node during the execution, where c is a command, a handler, or a termination signal (*stop*), $M = [m_N, m_F, m_G]$ represents the memory for the three scopes of node ($m_N : Var_N \to Val$), flow ($m_F : Var_F \to Val$), and global ($m_G : Var_G \to Val$), where Var_N, Var_F, and Var_G are disjoint sets, I is the input channel, and O is the array of output channels, reflecting that a node has one input port and as many output ports as it requires. We model an input (output) channel as a sequence of values that a node receives (sends). A class of nodes, called *inject* nodes, is triggered by external events such as button click or time. Inject nodes send new messages to a flow, thus triggering the execution of the flow. The $wires$ array records the nodes that can read the content of the output channel for the corresponding output port. A node receives a message if the node identifier is listed in $wires$ among the recipients of the output port assigned in a send command.

Trace-based Semantics Figure 8 illustrates the small-step semantics of nodes. We annotate transitions with the trace of events thus generated, where $\to \subseteq Config \times Config$ and $\Downarrow : (Exp \times Mem) \to Val$. A trace T is a finite sequence of events $t_k \in E$ defined by variable reads $R_k(x)$, variable writes $W_k(x)$, or function calls $f_k(v)$ generated by the execution of node k in a flow.

Expression evaluation is standard and records the sequence of events produced during the evaluation, where M_k denotes the memory M in $\langle c, M, I, O \rangle_k$. Command evaluation models the execution of a node's handler. The handler executes whenever there is a message in the input channel I by consuming the message and updating the memory accordingly. Assignments operate in a similar manner and record the trace of events produced by variable reads and writes. An assignment updates the memory M_k to M'_k, subsequently triggering an update of the flow and global memories, as stated in the rule (Step) in Figure 9 and in the rule (Global) in Figure 10. Send commands evaluate the expression e in the current memory, update the associated output channel, and record the trace of events. The index k distinguishes between events of different nodes. We write $\to^*$ for the reflexive and transitive closure of the $\to$ relation, and $\to^n$ for the n-step execution of $\to$.

Expression Evaluation

$$\frac{}{\langle v, M_k\rangle \Downarrow v} \text{ (Value)}$$

$$\frac{\langle e, M_k\rangle \Downarrow^{T_k} v}{\langle f(e), M_k\rangle \Downarrow^{T_k.f_k(v)} \bar{f}(v)} \text{ (Call)} \qquad \frac{}{\langle x, M_k\rangle \Downarrow^{R_k(x)} M_k(x)} \text{ (Read)}$$

Command Evaluation

$$\frac{I = I'.v \qquad x \in Var_N}{\langle \mathit{handler}(x)\{c\}, M, I, O\rangle_k \rightarrow \langle c, M[x \mapsto v], I', O\rangle_k} \text{ (Input)}$$

$$\frac{}{\langle \mathit{skip}, M, I, O\rangle_k \rightarrow \langle \mathit{stop}, M, I, O\rangle_k} \text{ (Skip)}$$

$$\frac{\langle e, M_k\rangle \Downarrow^{T_k} v \qquad M'_k = M_k[x \mapsto v]}{\langle x := e, M, I, O\rangle_k \xrightarrow{T_k.W_k(x)} \langle \mathit{stop}, M', I, O\rangle_k} \text{ (Write)}$$

$$\frac{c = \mathit{if}\ e\ \mathit{then}\ c_{true}\ \mathit{else}\ c_{false} \qquad \langle e, M_k\rangle \Downarrow^{T_k} b}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle c_b, M, I, O\rangle_k} \text{ (If)}$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{body} \qquad \langle e, M_k\rangle \Downarrow^{T_k} \mathit{true}}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle c_{body}; c, M, I, O\rangle_k} \text{ (While-T)}$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{body} \qquad \langle e, M_k\rangle \Downarrow^{T_k} \mathit{false}}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle \mathit{stop}, M, I, O\rangle_k} \text{ (While-F)}$$

$$\frac{\langle c_1, M, I, O\rangle_k \xrightarrow{T_k} \langle c'_1, M', I', O'\rangle_k}{\langle c_1; c_2, M, I, O\rangle_k \xrightarrow{T_k} \langle c'_1; c_2, M', I', O'\rangle_k} \text{ (Seq-1)}$$

$$\frac{}{\langle \mathit{stop}; c, M, I, O\rangle_k \rightarrow \langle c, M, I, O\rangle_k} \text{ (Seq-2)}$$

$$\frac{c = \mathit{send}(e, i) \qquad \langle e, M_k\rangle \Downarrow^{T_k} v \qquad O'[i] = O[i].v}{\langle c, M, I, O\rangle_k \xrightarrow{T_k} \langle \mathit{stop}, M, I, O'\rangle_k} \text{ (Output)}$$

Figure 8: Node semantics.

$$\frac{\begin{array}{c} I_l = v_l \qquad \forall N_k \in (Nodes(F_l) \setminus N_l).\, I_k = \varnothing \qquad M_l = [m_N, m_F, m_G] \\ M_l' = [m_N', m_F, m_G] \qquad config_l = \langle \textbf{handler}(x)\{c\}, M, I, O\rangle_l \\ config_l' = \langle c, M[x \mapsto v_l], \varnothing, O\rangle_l \\ config_l \rightarrow config_l' \\ N_l = \langle config_l, wires, l\rangle_l \qquad N_l' = \langle config_l', wires, l\rangle_l \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \rightarrow \langle m_F, (Nodes(F_l) \setminus \{N_l\}) \cup \{N_l'\}\rangle_l} \quad \text{(Init)}$$

$$\frac{\begin{array}{c} I_l = \varnothing \qquad M_k = [m_N, m_F, m_G] \qquad M_k' = [m_N', m_F', m_G'] \\ config_k = \langle c, M, I, O\rangle_k \qquad config_k' = \langle c', M', I', O\rangle_k \\ config_k \xrightarrow{T_k} config_k' \\ N_k = \langle config_k, wires, l\rangle_k \qquad N_k' = \langle config_k', wires, l\rangle_k \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \xrightarrow{T_k} \langle m_F', (Nodes(F_l) \setminus \{N_k\}) \cup \{N_k'\}\rangle_l} \quad \text{(Step)}$$

$$\frac{\begin{array}{c} config_k = \langle \textbf{send}(e, i); c, M, I, O\rangle_k \qquad config_k' = \langle \textbf{stop}; c, M, I, O'\rangle_k \\ O_k'[i] = O_k[i].v \qquad config_k \xrightarrow{T_k} config_k' \\ N_k = \langle config_k, wires, l\rangle_k \qquad N_k' = \langle config_k', wires, l\rangle_k \\ \omega = \{N_k\} \cup \{N_j \mid j \in wires_k[i]\} \\ \omega' = \{N_k'\} \cup \{N_j' \mid j \in wires_k[i],\, I_j' = v.I_j\} \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \xrightarrow{T_k} \langle m_F, (Nodes(F_l) \setminus \omega) \cup \omega'\rangle_l} \quad \text{(Send)}$$

$$\frac{\begin{array}{c} config_k = \langle \textbf{stop}, M, I, O\rangle_k \qquad config_k' = \langle \textbf{handler}(x)\{c\}, M, I, O\rangle_k \\ N_k = \langle config_k, wires, l\rangle_k \qquad N_k' = \langle config_k', wires, l\rangle_k \end{array}}{\langle m_F, Nodes(F_l)\rangle_l \rightarrow \langle m_F, (Nodes(F_l) \setminus \{N_k\}) \cup \{N_k'\}\rangle_l} \quad \text{(Term)}$$

Figure 9: Flow semantics.

$$\frac{\begin{array}{c} M_k = [m_N, m_F, m_G] \qquad M_k' = [m_N', m_F', m_G'] \\ F_l = \langle m_F, Nodes(F_l)\rangle_l \qquad F_l' = \langle m_F', Nodes(F_l')\rangle_l \\ F_l \xrightarrow{T_k} F_l' \end{array}}{\langle m_G, Flows(G)\rangle \xrightarrow{T_k} \langle m_G', (Flows(G) \setminus \{F_l\}) \cup \{F_l'\}\rangle} \quad \text{(Global)}$$

Figure 10: Global semantics.

Flow and Global Semantics We lift node semantics to formalize the semantics of flows and the environment. A global configuration $G = \langle m_G, \{F_l \mid l \in L\}\rangle$ consists of a global shared memory m_G and a finite set of flows that are executing concurrently, where $L \subset \mathbb{N}$ is the set of flow identifiers. A flow configuration $F = \langle m_F, \{N_k \mid k \in K\}\rangle_l$ is a tuple consisting of a flow shared memory m_F, a finite set of nodes where $K \subset \mathbb{N}$ is the set of node identifiers, and l is the flow identifier. We use $Nodes(F_l)$ for the set of nodes in a specific flow and $Flows(G)$ for the set of flows in the environment. Nodes are distinguished by unique node identifiers in the environment and the node N_k can be present in only one flow. To unify the trigger point of the flow, we assume that a flow has only one inject node and denote it by N_l where $N_l \in Nodes(F_l)$; in practice, it can be considered as a dummy node which is the predecessor of all the inject nodes of the flow.

We model a flow by linking the output channels of a node to the input channels of the next ones based on the flow specification. Note that a node can have more than one output channel but only one input channel. The inject node of a flow, which does not appear in any of the *wires* arrays, triggers the flow execution by injecting a new message. An initial value is assigned to the input channel of the inject node to model the behavior of the external event such as a button click. We write $Exec(F_l, v_l)$ to refer to executions of a flow F_l with an initial value v_l. Also, $Exec(G, V)$ denotes executions of the environment G with the set of initial values $V = \{(N_l, v_l) \mid F_l \in Flows(G)\}$ for the member flows.

We remark that message passing in Node-RED is asynchronous and message objects traverse the graph in a non-deterministic manner, as reported in the documentation ("no assumptions should be made about ordering once a flow branches" [34] and "flows can be cyclic" [33]). Hence, we model the execution of nodes in a flow and the environment, as shown in Figures 9 and 10, respectively. We overload the notation $\rightarrow$ for transitions between flow and global configurations. In a nutshell, the flow and global semantics implements the non-deterministic behavior of flows and the environment, and lifts the node semantics to ensure that the flow of messages follows the flow specification.

The intuition of the rules is that the inject node of a flow, i.e., the node N_l of the flow F_l, starts the execution by consuming the initial value (rule Init), and then the execution continues according to the node semantics (rule Step). When a node reaches a send command, it adds the output value to the input channels of the next nodes in the flow; the output value transmits out to the output channel indicated by the send command and the input channels of all nodes in the corresponding elements of the array *wires* get updated with

the value (rule SEND); $wires_k$ denotes the array $wires$ in $\langle config, wires, l\rangle_k$. The execution proceeds until it terminates and gets back to the initial state, ready to consume the next value in the input channel (rule TERM). Note that nodes are running concurrently; any of the ready nodes can make one execution step. The only rule in the global semantics (rule GLOBAL) shows that any of the flows with at least one ready node can make an execution step.

Generally speaking, any node that is able to progress continues the execution for one execution step, and it might affect the flow and global contexts. An execution step of a node corresponds to one execution step of the flow it belongs to and one execution step of the environment. Considering the non-deterministic behavior of Node-RED's scheduler, any ready node can be selected for the next execution step.

3.2 Security condition and enforcement

We leverage our trace-based semantics to define a semantics-based security condition. The condition is parametric on node-level security policies, represented as allowlists of API calls and accesses to the shared context. Then, we present the semantics of a fine-grained node-level monitor and prove its soundness and transparency with respect to the security condition.

3.2.1 Security condition

We extend the definition of nodes with allowlist policies $N = \langle config, wires, l, P, V, S\rangle_k$, where $P \subseteq APIs \subseteq Fun$ describes permitted API functions, $V : P \to 2^{Val}$ defines the allowlist of arguments for each API function, and S specifies read/write permissions on the shared global and flow variables, such that $S = \{(x, RW) \mid x \in Var_F \uplus Var_G, RW \in \{R, W\}\}$.

The security condition matches the trace of events produced by the semantics with the allowlist policies to check that any event produced by an execution is permitted by the policy.

Definition 1 (*Event Security*)**.** Let t_k be an event emitted from an execution of node N_k. We define a secure event with respect to $\langle P_k, V_k, S_k\rangle$, written $secure(t_k, \langle P_k, V_k, S_k\rangle)$, as follows:

$$secure(R_k(x), \langle P_k, V_k, S_k\rangle) \triangleq x \in Var_F \cup Var_G \Rightarrow (x, R) \in S_k$$
$$secure(W_k(x), \langle P_k, V_k, S_k\rangle) \triangleq x \in Var_F \cup Var_G \Rightarrow (x, W) \in S_k$$
$$secure(f_k(v), \langle P_k, V_k, S_k\rangle) \triangleq f \in APIs \Rightarrow f \in P_k \wedge v \in V_k(f).$$

We lift the security of events to define the security condition for node traces $secure(T_N)$, flows traces $secure(T_F)$, and global traces $secure(T_G)$ as expected. A finite sequence of events forms a trace. Hence a trace is secure if all its events are secure. We define trace security by the conjunction of security checks on the composing events.

Definition 2 (*Trace Security*)**.** Trace T is secure, written $secure(T)$, if

$$T = t_k.T' \Rightarrow secure(t_k, \langle P_k, V_k, S_k\rangle) \wedge secure(T').$$

A node starts executing when it receives a value over its input channel. An execution of a node is secure if the corresponding trace is secure, according to the node policy.

Definition 3 (*Node-Level Security*)**.** The execution of a node $N_k = \langle config, wires, l, P, V, S\rangle_k$ with an input message $I = v$ is secure with regard to $\langle P_k, V_k, S_k\rangle$ if each step of the node execution complies with $\langle P_k, V_k, S_k\rangle$, i.e.,

$$\forall \langle c', M', I', O'\rangle_k.\, \langle handler(x)\{c\}, M, v, O\rangle_k \xrightarrow{T_k}{}^{*} \langle c', M', I', O'\rangle_k \Rightarrow secure(T_k).$$

We now define the security of Node-RED app executions based on the flow and global semantics. The inject node of a flow initiates the flow execution, and it triggers other nodes by traversing the flow graph. At the global level, nodes in flows generate events while they are executing concurrently in the environment. We present flow and global execution security for the trace of events produced by their nodes at each execution step.

Definition 4 (*Flow-Level Security*)**.** Let F_l be a flow and v_l be an initial value for the inject node of the flow, i.e., $N_l = \langle\langle handler(x)\{c\}, M, v_l, O\rangle_l, wires, l\rangle_l$. We define flow executions $Exec(F_l, v_l)$ secure if

$$N_l \in Nodes(F_l) \wedge \forall F_l'.\, F_l \xrightarrow{T_F}{}^{*} F_l' \Rightarrow secure(T_F).$$

The trace T_F is secure if $secure(T_F)$ holds, i.e., every event of the trace is secure according to the security policy of the corresponding node.

Definition 5 (*Global-Level Security*)**.** Let G be an environment and V_{init} be a set of initial values for the inject nodes of the flows in G, i.e., $\forall (N_j, v_j) \in V_{init}.\quad F_j \in Flows(G) \wedge N_j \in Nodes(F_j) \wedge N_j = \langle\langle handler(x)\{c\}, M, v_j, O\rangle_j, wires, j\rangle_j$. We define global executions $Exec(G, V_{init})$ secure if

$$\forall G'.\, G \xrightarrow{T_G}{}^{*} G' \Rightarrow secure(T_G).$$

Expression Evaluation

$$\frac{secure(R_k(x), \langle P_k, V_k, S_k\rangle)}{\langle x, M_k\rangle \Downarrow_{\mathcal{M}}^{R_k(x)} M_k(x)} \quad (\textsc{Read}_{\mathcal{M}})$$

$$\frac{\langle e, M_k\rangle \Downarrow^{T_k} v \qquad secure(f_k(v), \langle P_k, V_k, S_k\rangle)}{\langle f(e), M_k\rangle \Downarrow_{\mathcal{M}}^{T_k.f_k(v)} \bar{f}(v)} \quad (\textsc{Call}_{\mathcal{M}})$$

Command Evaluation

$$\frac{secure(W_k(x), \langle P_k, V_k, S_k\rangle) \qquad \langle e, M_k\rangle \Downarrow^{T_k} v \qquad M' = M[x \mapsto v]}{\langle x := e, M, I, O\rangle_k \xrightarrow{T_k.W_k(x)}_{\mathcal{M}} \langle stop, M', I, O\rangle_k} \quad (\textsc{Write}_{\mathcal{M}})$$

Figure 11: Excerpt of monitor semantics.

3.2.2 Enforcement Mechanism

Figure 11 presents the core of our fine-grained monitor to enforce the above-mentioned security condition with respect to allowlist policies. We annotate evaluation relations with $\mathcal{M}$ to distinguish between the monitored behavior and the original one. We only present the rules that differ from the semantic rules given in Figure 8; we replace $\rightarrow$ with $\rightarrow_{\mathcal{M}}$, and $\Downarrow$ with $\Downarrow_{\mathcal{M}}$. We add security constraints to the semantic rules for reading a variable from the shared context (rule $\textsc{Read}_{\mathcal{M}}$), calling an API function (rule $\textsc{Call}_{\mathcal{M}}$), and writing to a shared variable (rule $\textsc{Write}_{\mathcal{M}}$).

For the email example in Section 2, the policy requires allowlisting the API for sending the message and the list of intended recipients. The monitor intervenes whenever the API is called and ensures that the recipient is in the allowlist policy. An execution of a flow containing the malicious email node will be suppressed because the attacker's email address is not listed in the permitted values of the API call. The malicious event is detected by the rule $\textsc{Call}_{\mathcal{M}}$, i.e., $\texttt{sendMail} \in P_k \wedge \texttt{"me@attacker.com"} \notin V_k(\texttt{sendMail})$.

For context vulnerabilities, such as Water Utility Complete Example, the allowlist consists of access rights to shared variables for each node deployed in the environment. The monitor observes the interaction of nodes with the shared context and blocks the execution whenever the allowlist policy does not permit access to the shared variable. The attack scenario in the vulnerable water utility flow can also be prevented by specifying an allowlist policy $(\texttt{tank1Level}, W)$ only for the nodes that must write to a shared variable, which stops any attempt from other nodes to write to the global context

(rule $\textsc{Write}_{\mathcal{M}}$).

We prove the soundness and transparency properties of our monitor. The soundness theorem shows that any global traces produced by an execution of the monitor are secure with respect to the allowlist policy.

Theorem 1 (Soundness)**.** *The monitor enforces global-level security for any finite executions,*

$$\forall(G, V).\, \forall G'.\, G \xrightarrow{T_G}{}^{*}_{\mathcal{M}} G' \Rightarrow secure(T_G).$$

The transparency theorem shows that if a monitored execution is secure, the monitor semantics and the original semantics generate the same trace. Moreover, if both semantics run under the same scheduler, the monitor preserves the longest secure prefix of a trace.

Theorem 2 (Transparency)**.** *The monitor preserves the longest secure prefix of a trace yielded by an execution,*

$$\forall(G_0, V).\, \forall n \in \mathbb{N}.\, G_0 \xrightarrow{T}{}^{n} G_n \Rightarrow \exists m \leq n.\, G_0 \xrightarrow{T'}{}^{m}_{\mathcal{M}} G_m \wedge$$

$$\Big[\Big(secure(T) \Rightarrow T = T' \wedge n = m\Big) \vee$$

$$\Big(\Big(\exists i < n.\, G_0 \xrightarrow{T_{pre}}{}^{i} G_i \wedge G_i \xrightarrow{T_i} G_{i+1} \wedge G_{i+1} \xrightarrow{T_{post}}{}^{n-i-1} G_n \wedge secure(T_{pre}) \wedge$$
$$\neg secure(T_i)\Big) \Rightarrow T' = T_{pre} \wedge i = m\Big)\Big].$$

The proofs of the theorems are reported in Appendix 2.A.

4 Related work

We discuss the most closely related work on Node-RED security and modeling, monitor implementation, and securing trigger-action platforms in general. We refer the reader to surveys on the security of IoT app platforms [6, 13] for further details.

Node-RED security and modeling Ancona et al. [4] investigate runtime monitoring of parametric trace expressions to check the correct usage of API functions in Node-RED. Trace expressions allow for rich policies, including temporal patterns over sequences of API calls. By contrast, our monitor supports both coarse and fine access control granularity of modules, functions, and contexts. Schreckling et al. [48] propose COMPOSE, a framework for fine-grained static and dynamic enforcement that integrates JSFlow [20],

an information-flow tracker for JavaScript. COMPOSE focuses on data-level granularity, whereas our monitoring framework supports module- and API-level granularity.

Clerissi et al. [14] use UML models to generate and test Node-RED flows to provide early system validation. A preliminary set of guidelines has also been proposed to assist Node-RED flow makers in terms of user comprehension and for testing activities [15]. Focusing more on end users and less on developers, Kleinfeld et al. [26] introduce an extension of Node-RED called glue.things, enabling Node-RED easier to use by predefined trigger and action nodes. Blackstock and Lea [11] propose a distributed runtime for Node-RED apps such that flows can be hosted on various platforms. Tata et al. [52] propose a formal modeling for decomposing process-aware applications deployed in IoT environments using Petri nets; Node-RED indeed fits in this setup, thus extended as a prototype for their approach [24].

In terms of modeling, Node-RED can be intrinsically seen as a concurrent system, thus our approach shares similarities with the broad range of formal approaches such as process calculi [7, 45], CSP [21], and CCS [30]. In the same spirit, our formalization is targeted to capture the execution model of Node-RED flows consisting of concurrent node executions that trigger the execution of code upon receiving messages, and modify, create, and dispatch messages to the next nodes. In contrast, our modeling is explicit and it captures the essence of the execution semantics of Node-RED. Focusing on security policies in concurrent systems, KLAIM [10, 31] is a programming language providing a mechanism to customize access control policies. The mechanism, based on a hierarchical capability-based type system, enforces policies that control resource usage and authorize migration and execution of processes. While KLAIM is designed for programming distributed applications with agents and code mobility, our Node-RED model is simple and expressive enough to describe the API-based access control enforcement mechanism.

Monitor implementation Regarding the possible candidates for implementing our theoretical framework, it should be noted that the dynamic nature of JavaScript requires more precise analysis provided by dynamic approaches. Andreasen et al. [5] survey available methods for dynamic analysis for JavaScript, and outline three general categories: runtime instrumentation, source code instrumentation, and metacircular interpreters.

DProf [18] and NodeProf [51] are two well-known runtime instrumentation tools. DProf instruments a program at the instruction level, targeting a variety of languages, including JavaScript. NodeProf instead instruments a program at the abstract syntax tree (AST) level and is specifically made as a

dynamic analysis framework for Node.js. However, some important Node.js features, such as `module.exports`, commonly used in Node-RED nodes, are not supported by NodeProf yet. In addition, to obtain the desired results, it requires the instrumentation of the entire Node-RED environment. NodeMOP [47] is a Monitoring-Oriented Programming (MOP) tool built on top of NodeProf that also looks interesting for our purposes, while the challenges in practice remain unresolved.

Ferreira et al. [17] propose a lightweight permission system to enforce the least-privilege principle at the Node.js packages level at runtime, restricting access to security-critical APIs and resources. Sharing some of our motivations, however, this work does not enforce access control policies at the context and value levels. Pyronia [28] is a fine-grained access control system for IoT applications restricting access at the function level via runtime and kernel modifications. To detect access to sensitive resources, Pyronia leverages OS-level techniques such as system call interposition and stack inspection. By contrast, our monitor needs to be implemented in language-level isolation to prevent access to sensitive resources at different levels of granularity.

Membrane-based approaches [1, 2, 19, 29, 49] seem to be the most promising compared to other techniques. Membranes are a "defensive programming pattern used to intermediate between sub-components of an application" [53]. This pattern is implemented in Node.js by recursively wrapping an object in a proxy with respect to prototype hierarchies such that the wrapped object can only be modified in protected ways. Staicu et al. [50] provide an example of this technique applied to Node.js, isolating libraries to extract taint specifications automatically.

SandTrap [2] combines the Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. SandTrap has been integrated with Node-RED and evaluated on a set of flows while enforcing a variety of policies yet incurring negligible runtime overhead. Our framework is a step toward providing the formal grounds for characterizing the soundness and transparency of the SandTrap instantiation to Node-RED. The formalization can be further enhanced by modeling the Node.js environment and full-featured JavaScript [27].

Securing trigger-action platforms IoTGuard [12] is a monitor for enforcing security policies written in the IoTGuard policy language. Security policies describe valid transitions in an IoT app execution. Bastys et al. [8, 9] study attacks by malicious app makers in IFTTT and Zapier. They develop dynamic and static information flow control (IFC) in IoT apps and report on an empirical study to estimate to what extent IFTTT apps manipulate sensitive information of users. Wang et al. [55] develop NLP-based methods

to infer information flows in trigger-action platforms and check cross-app interaction via model checking. Alpernas et al. [3] propose dynamic coarse-grained IFC for JavaScript in serverless platforms. Our presented monitor is based on access control rather than IFC. Hence, these works are complementary, focusing on information flow after access is granted. IFC supports rich dependency policies, yet arduous to track information flow in JavaScript without breaking soundness or giving up precision.

5 Conclusion

We have investigated the security of Node-RED, an open-source JavaScript-driven trigger-action platform. We have expanded on the recently-discovered critical exploitable vulnerabilities in Node-RED, where the impact ranges from massive exfiltration of data from unsuspecting users to taking over the entire platform. Motivated by the need for a security mechanism for Node-RED, we have proposed an essential model for Node-RED, suitable to reason about nodes and flows, be they benign, vulnerable, or malicious. We have formalized a principled framework to enforce fine-grained API control for untrusted Node-RED applications. Our formalization for a core language shows how to soundly and transparently enforce global security properties of Node-RED applications by local access checks, supporting module-, API-, value-, and context-level policies.

Acknowledgments This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

Bibliography

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete Client-side Sandboxing of Third-party JavaScript without Browser Modifications. In *ACSAC*, 2012.

[2] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security*, 2021.

[3] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein. Secure Serverless Computing using Dynamic Information Flow Control. In *OOPSLA*, 2018.

[4] D. Ancona, L. Franceschini, G. Delzanno, M. Leotta, M. Ribaudo, and F. Ricca. Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *ALP4IoT@iFM*, 2017.

[5] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, 2017.

[6] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE S&P Magazine*, 2019.

[7] M. Balliu, M. Merro, M. Pasqua, and M. Shcherbakov. Friendly Fire: Cross-app Interactions in IoT Platforms. *ACM Trans. Priv. Secur.*, 2021.

[8] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *CCS*, 2018.

[9] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In *NordSec*, 2018.

[10] L. Bettini, V. Bono, R. D. Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim Project: Theory and Practice. In *Global Computing*, 2003.

[11] M. Blackstock and R. Lea. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *WoT*, 2014.

[12] Z. Celik, G. Tan, and P. D. M. and. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*, 2019.

[13] 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 Computing Surveys*, 2019.

[14] D. Clerissi, M. Leotta, G. Reggio, and F. Ricca. Towards An Approach for Developing and Testing Node-RED IoT Systems. In *EnSEmble@ESEC/SIGSOFT FSE*, 2018.

[15] D. Clerissi, M. Leotta, and F. Ricca. A Set of Empirically Validated Development Guidelines for Improving Node-RED Flows Comprehension. In *ENASE*, 2020.

[16] D. Devriese and F. Piessens. Noninterference through Secure Multi-Execution. In *S&P*, 2010.

[17] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *ICSE*, 2021.

[18] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[19] W. D. Groef, F. Massacci, and F. Piessens. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *ACSAC*, 2014.

[20] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

[21] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 1978.

[22] IBM Cloud. `https://cloud.ibm.com/`, 2021.

[23] IFTTT: If This Then That. `https://ifttt.com`, 2021.

[24] R. Jain, K. Klai, and S. Tata. Formal Modeling and Verification of Scalable Process-Aware Distributed IoT Applications. In *ISPA/BDCloud/SocialCom/SustainCom*, 2019.

[25] jcreedcmu. Escaping NodeJS vm. `https://gist.github.com/jcreedcmu/4f6e6d4a649405a9c86bb076905696af`, 2018.

[26] R. Kleinfeld, S. Steglich, L. Radziwonowicz, and C. Doukas. glue.things: a Mashup Platform for Wiring the Internet of Things with the Internet of Services. In *WoT*, 2014.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *APLAS*, 2008.

[28] M. S. Melara, D. H. Liu, and M. J. Freedman. Pyronia: Intra-Process Access Control for IoT Applications. *CoRR abs/1903.01950*, 2019.

[29] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[30] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982.

[31] R. D. Nicola, G. L. Ferrari, and R. Pugliese. Programming access control: The KLAIM experience. In *CONCUR*, 2000.

[32] Node-RED. Community Node Module Catalogue. `https://github.com/node-red/catalogue.nodered.org`, 2021.

[33] Node-RED. Cyclic Flows. `https://groups.google.com/g/node-red/c/C6M3HokoSTI/m/B2tqcb_cAQAJ`, 2021.

[34] Node-RED. Making Flows Asynchronous by Default. `https://nodered.org/blog/2019/08/16/going-async`, 2021.

[35] Node-RED. `https://nodered.org/`, 2021.

[36] Node-RED. Securing Node-RED. `https://nodered.org/docs/user-guide/runtime/securing-node-red`, 2021.

[37] Node-RED. The Core Nodes. `https://nodered.org/docs/user-guide/nodes`, 2021.

[38] Node-RED. The RED Object. `https://github.com/node-red/node-red/blob/master/packages/node_modules/node-red/lib/red.js`, 2021.

[39] Node-RED. Working with Context. `https://nodered.org/docs/user-guide/context`, 2021.

[40] Node-RED Library. `https://flows.nodered.org/`, 2021.

[41] Node.JS. VM (executing JavaScript). `https://nodejs.org/api/vm.html#vm_vm_executing_javascript`, 2021.

[42] NPM. Node Package Manager. `https://www.npmjs.com/`, 2021.

[43] OWASP. NodeJS Security Cheat Sheet. `https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html#do-not-use-dangerous-functions`, 2021.

[44] B. Pfretzschner and L. ben Othmane. Identification of Dependency-based Attacks on Node.js. In *ARES*, 2017.

[45] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.

[46] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 1975.

[47] F. Schiavio, H. Sun, D. Bonetta, A. Rosà, and W. Binder. NodeMOP: Runtime Verification for Node.js Applications. In *SAC*, 2019.

[48] D. Schreckling, J. D. Parra, C. Doukas, and J. Posegga. Data-Centric Security for the IoT. In *IoT 360 (2)*, 2015.

[49] P. Simek. Proposal for VM2: Advanced vm/sandbox for Node.js. `https://github.com/patriksimek/vm2`, 2021.

[50] C. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting Taint Specifications for JavaScript Libraries. In *ICSE*, 2020.

[51] H. Sun, D. Bonetta, C. Humer, and W. Binder. Efficient Dynamic Analysis for Node.js. In *CC*, 2018.

[52] S. Tata, K. Klai, and R. Jain. Formal Model and Method to Decompose Process-Aware IoT Applications. In *OTM*, 2017.

[53] Tom Van Cutsem. Isolating Application Sub-components with Membranes. `https://tvcutsem.github.io/membranes`, 2018.

[54] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman. Practical Trigger-Action Programming in the Smart Home. In *CHI*, 2014.

[55] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter. Charting the Attack Surface of Trigger-Action IoT Platforms. In *CCS*, 2019.

[56] Zapier. `https://zapier.com`, 2021.

[57] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security*, 2019.

Appendix

2.A Proofs

To prove the soundness theorem, we show that each execution step of a node under the monitor generates secure events.

Lemma 1. *Let $N_k = \langle config, wires, l, P, V, S\rangle_k$ be a node. Any semantic step of N_k under the monitor produces a secure trace with regard to $\langle P_k, V_k, S_k\rangle$, i.e., $\forall N_k.\ config_k \xrightarrow{T_k}_{\mathcal{M}} config'_k \Rightarrow secure(T_k)$.*

Proof. First we show that any trace produced from the expression evaluation rules is secure. By induction on the derivation $\langle e, M_k\rangle \Downarrow_{\mathcal{M}} v$:

- The rule (Value) generates an empty (secure) trace.

- The rule (Read$_{\mathcal{M}}$) only generates the event $R_k(x)$ if it meets the security condition for reading a variable, i.e., $secure(R_k(x), \langle P_k, V_k, S_k\rangle)$.

- In the rule (Call$_{\mathcal{M}}$), by the induction hypothesis, $\langle e, M_k\rangle \Downarrow_{\mathcal{M}}^{T_k} v \Rightarrow secure(T_k)$. Then, the trace $T_k.f_k(v)$ is generated if the API call and the value of the expression e obeys the security condition for API calls, i.e., $secure(f_k(v), \langle P_k, V_k, S_k\rangle)$. Therefore, $secure(T_k) \wedge secure(f_k(v), \langle P_k, V_k, S_k\rangle) \Rightarrow secure(T_k.f_k(v), \langle P_k, V_k, S_k\rangle)$.

Next, by induction on the derivation $config_k \xrightarrow{T_k}_{\mathcal{M}} config'_k$, we prove the lemma:

- Rules (Input), (Skip), and (Seq-2) generate empty traces, which are trivially secure.

- Rules (If), (While-T), (While-F) and (Output) generate the same trace resulting from the expression evaluation $\langle e, M_k\rangle \Downarrow_{\mathcal{M}}^{T_k} v \Rightarrow secure(T_k)$, because of the proof above.

- The trace T_k generated in Rule (Seq-1) is secure, based on the induction hypothesis.

- The rule (Write$_{\mathcal{M}}$) emits a secure trace since $\langle e, M_k\rangle \Downarrow_{\mathcal{M}}^{T_k} v \Rightarrow secure(T_k)$, and $secure(T_k) \wedge secure(W_k(x), \langle P_k, V_k, S_k\rangle) \Rightarrow$

$secure(T_k.W_k(x), \langle P_k, V_k, S_k\rangle)$. Because any trace generated by the rules of expression evaluation $\langle e, M_k\rangle \Downarrow_{\mathcal{M}} v$ is secure, and the write event is produced only if it complies with the security condition for writing into a variable, i.e., $secure(W_k(x), \langle P_k, V_k, S_k\rangle)$. □

We have proved the node-level security as a corollary of Lemma 1. Hence, the generated trace from a transition between any two node configurations is secure. Next, we prove that any trace generated by a flow execution under the monitor is secure.

Lemma 2. *Any semantic step of a flow F_l under the monitor produces a secure trace,* $\forall F_l, F_l'.\ F_l \xrightarrow{T_F}_{\mathcal{M}} F_l' \Rightarrow secure(T_F)$.

Proof. By case analysis on the flow semantics rules:

- The rules (INIT) and (TERM) yield empty (secure) traces, which are trivially secure.

- The rules (STEP) and (SEND) repeat the same trace generated from the corresponding transition between node configurations. Lemma 1 demonstrates that $\forall N_k.\ config_k \xrightarrow{T_k}_{\mathcal{M}} config_k' \Rightarrow secure(T_k)$. Thus, the theorem also holds for these cases. □

Lemma 3. *Let G be a global configuration. Any semantic step of G under the monitor is secure,* $\forall G, G'.\ G \xrightarrow{T_G}_{\mathcal{M}} G' \Rightarrow secure(T_G)$.

Proof. The single rule in the global semantics replicates the trace produced by the transition between the two flow configurations. Lemma 2 shows flow transitions are secure under the monitor, thus the global transitions. Because $\Big(\forall G, G'.\ G \xrightarrow{T_G}_{\mathcal{M}} G' \Rightarrow secure(T_G)\Big) \Leftrightarrow \Big(\forall F_l, F_l'.\ F_l \xrightarrow{T_F}_{\mathcal{M}} F_l' \Rightarrow secure(T_F)\Big)$. □

Theorem 1. By using the lemma 3 and multiple repetitions of the single rule of the global semantics, the soundness theorem is proven as a corollary. □

To prove the transparency theorem, we show that the monitor preserves the secure events emitted from a node.

Lemma 4. *Any semantic step in the original execution of a node that emits a secure trace remains the same in the monitor semantics,* $\forall N_k, N_k'.\ conf_k \xrightarrow{T_k} conf_k' \wedge secure(T_k) \Rightarrow conf_k \xrightarrow{T_k}_{\mathcal{M}} conf_k'$.

Proof. By induction on $\langle e, M_k\rangle \Downarrow v$, we observe that there is a one-to-one mapping from the rules for $\Downarrow$ and $\Downarrow_{\mathcal{M}}$ if the security conditions $secure(R_k(x), \langle P_k, V_k, S_k\rangle)$ and $secure(f_k(v), \langle P_k, V_k, S_k\rangle)$ hold.

By induction on the derivation $conf_k \xrightarrow{T_k} conf'_k$, again we can see a one-to-one correspondence between the rules for $\rightarrow$ and $\rightarrow_{\mathcal{M}}$, as a result of the induction on $\langle e, M_k\rangle \Downarrow v$, and the comparison between the rule (Write) in the standard semantics and the rule ($\text{Write}_{\mathcal{M}}$) in the monitor semantics, which requires $secure(W_k(x), \langle P_k, V_k, S_k\rangle)$ to be held. □

We assume utilizing a deterministic order-preserving scheduler that both the original semantics and the monitor employ. The non-deterministic scheduler might affect the order of events generated by the global and flow transitions.

Lemma 5. *Any semantic step of the global configuration that generates a secure trace remains the same in the monitor semantics,* $\forall G, G'.\ G \xrightarrow{T_k} G' \wedge secure(T_k) \Rightarrow G \xrightarrow{T_k}_{\mathcal{M}} G'$.

Proof. The standard and the monitor semantics use the same global and flow semantics. With the assumption of employing an identical deterministic scheduler and using lemma 4, we can write $\forall G, G'.\ G \xrightarrow{T_k} G' \wedge secure(T_k) \Rightarrow \exists! F_l, N_k, F'_l, N'_k.\ F_l \in Flows(G) \wedge N_k \in Nodes(F_l) \wedge F'_l \in Flows(G') \wedge N'_k \in Nodes(F'_l) \wedge conf_k \xrightarrow{T_k}_{\mathcal{M}} conf'_k$. Similarly, the statement holds for $\xrightarrow{T_k}_{\mathcal{M}}$ in the other way. □

Theorem 2. Starting with the initial configuration (G_0, V_{init}) and using the global semantics, there are two cases:

- Case 1 (the trace is secure): If $secure(T)$, using the lemma 5 for *n*-times results $T = T' \wedge n = m$.

- Case 2 (the trace is not secure): If $T = T_{pre}.\, T_i.\, T_{post}$ where $secure(T_{pre}) \wedge \neg secure(T_i)$, then using the lemma 5 for *i* times concludes $T' = T_{pre} \wedge i = m$. Thereafter, no semantic rule applies for the transition $G_i \xrightarrow{T_{pre}}{}^{i} G_{i+1}$ in the monitor semantics. □

3

Nontransitive Policies Transpiled

Mohammad M. Ahmadpanah, Aslan Askarov, and Andrei Sabelfeld

EuroS&P'21

Abstract. Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) are a new security condition and enforcement for policies which, in contrast to Denning's classical lattice model, assume no transitivity of the underlying flow relation. Nontransitive security policies are a natural fit for coarse-grained information-flow control where labels are specified at module rather than variable level of granularity.

While the nontransitive and transitive policies pursue different goals and have different intuitions, this paper demonstrates that nontransitive noninterference can in fact be reduced to classical transitive noninterference. We develop a lattice encoding that establishes a precise relation between NTNI and classical noninterference. Our results make it possible to clearly position the new NTNI characterization with respect to the large body of work on noninterference. Further, we devise a lightweight program transformation that leverages standard flow-sensitive information-flow analyses to enforce nontransitive policies. We demonstrate several immediate benefits of our approach, both theoretical and practical. First, we improve the permissiveness over (while retaining the soundness of) the nonstandard NTT enforcement. Second, our results naturally generalize to a language with intermediate inputs and outputs. Finally, we demonstrate the practical benefits by utilizing state-of-the-art flow-sensitive tool JOANA to enforce nontransitive policies for Java programs.

1 Introduction

Modern approaches to secure information flow follow Denning's classical model [8]. This model maps information to security levels and uses a flow relation that regulates how information can move between the levels. Under Denning's model, when data moves from one security level to another one, it effectively looses its original security classification. Denning therefore argues that in such a model, the flow relation must be transitive, which has been the convention for a large body of work on information flow control [13, 26, 32].

Nontransitive policies In recent work, Lu and Zhang [17] observe that in certain scenarios, the transitivity requirement is in fact undesirable. This is most apparent when security policies are specified in a coarse-grained manner, i.e., at the level of mutually-distrustful components in an application. For example, "component Alice may trust only another component Bob with her information, however due to implied transitive relations, her information may flow not only to Bob but also indirectly to all components that Bob trusts, which is undesirable for Alice" [17]. Another, more fine-grained example, is that of user policies in a social network stipulating that "my friends can access my personal data but not friends of my friends". To semantically characterize such security requirements, Lu and Zhang propose the notion of *nontransitive* noninterference (NTNI) and propose a specially designed type system to statically enforce it.

Nontransitive noninterference is not to be confused with *intransitive* noninterference [18, 23, 25, 30], a popular model for declassification. Although both nontransitive and intransitive policies assume flow relations are not transitive, there is a conceptual difference between them. Assuming a flow relation with flows from A to B and from B to C but not from A to C, intransitive noninterference allows A's information to indirectly flow to C as long as the information passes through a declassifier. In contrast, nontransitive policy forbids all flows from A to C. Section 7 elaborates the relation in detail.

NTNI is introduced by a nonstandard security characterization and a spe-

cialized type system [17]. The question remains open whether the mainstream machinery of information-flow control reasoning and enforcement can be leveraged for tracking NTNI.

This paper answers this question positively by showing how to encode nontransitive noninterference via classical transitive noninterference. Our encoding makes it possible to use standard transitive techniques for information-flow control to enforce nontransitive policies and thus address the coarse-grained scenarios that motivate them. This has substantial practical benefits, making it possible to deploy information-flow concepts and tools to achieve nontransitive security.

We argue that flow-sensitive analysis is a natural fit for the component-based scenario, where developers are not required to provide fine-grained annotations at the level of variables. We devise a lightweight program transformation to leverage flow-sensitive information-flow analysis to enforce NTNI. Thanks to the flow-sensitivity of the analysis, the type system verifies which variables are affected by what components, enforcing component-level security. We implement a prototype of the transpiler, i.e., program transformer and policy translator, and leverage flow-sensitive static tool JOANA [11] to demonstrate our approach in practice.

Contributions The contributions of this paper are:

- We show that the definition of NTNI can be reduced to classical transitive noninterference through a lattice encoding (Section 2).
- We leverage our encoding to show how an existing flow-sensitive information-flow type system can enforce the coarse-grained policies that motivate NTNI in the first place (Section 3).
- We extend our results to a language supporting interaction through input and output commands (Section 4).
- We develop a prototype that translates NTNI policy to a classical transitive setting and uses JOANA static analysis tool (Section 5).

2 Security characterization transpiled

All permitted flows between security levels are expressed explicitly under nontransitive policies, as opposed to the traditional way [8] of policy specification where security levels constitute a partially ordered set. Nontransitive policies only have reflexive property, yet expressive enough to include other properties such as transitivity and antisymmetry among arbitrary selections of levels.

This section shows how nontransitive noninterference can be modeled as transitive noninterference using a power-lattice encoding. Throughout the

```
Alice {
  data;
  main() {
    Bob.receive(data);
    Bob.good();
    Bob.bad();
  }
}
Bob {
  data1;
  data2;
  receive(x) { data1 = x; }
  good() { Charlie.receive(data2); }
  bad() { Charlie.receive(data1); }
}
Charlie {
  data;
  receive(x) { data = x; }
}
```

Figure 1: Running example [17].

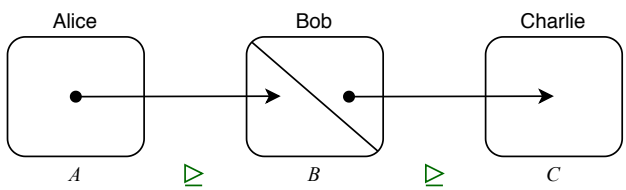


Figure 2: Nontransitive policy for the running example.

paper, we use a running example adopted from Lu and Zhang [17] to discuss how the transpilation works. We formalize the security notions and prove the relation between these two approaches to define a security policy.

Running example Figure 1 shows our running example consisting of three components named `Alice`, `Bob`, and `Charlie`. The security policy stipulates that `Bob` is allowed to read `Alice`'s information and `Charlie` is allowed to read `Bob`'s information. At the same time, no information flow from `Alice` is allowed to `Charlie`.

Based on the policy, `Bob` can only send information to `Charlie` if it is not influenced by `Alice`, as illustrated in Figure 2. A transitive policy would presume that if information may flow from `Alice` to `Bob` and from `Bob` to `Charlie`, then it may also flow from `Alice` to `Charlie`. This is not the case

in this example. Since nontransitive policies specify all permitted flows explicitly, the information flow from `Alice` to `Charlie` would be considered as desired only if it was explicitly stated in the policy. It is indeed easy to see that nontransitive policies are a generalization of transitive ones because transitive closures can be stated as permitted flows to preserve the transitive property.

Using a coarse-grained information-flow control is sufficient to specify the intended policy. Consider the labels A, B, and C for the components `Alice`, `Bob`, and `Charlie`, respectively. We specify the nontransitive policy using an arbitrary information flow relation $\trianglerighteq$[1], written $A \trianglerighteq B$ and $B \trianglerighteq C$, which specifies that information from security level A can flow to security level B and from B to C. It also stipulates any other information flows between the levels are disallowed. For instance, information from security level A must not flow to C, directly or through any other components.

For the sake of simplicity, we rewrite the example program in a model language (without support for object-orientation) that demonstrates the explicit flows arisen from data dependencies between component variables. In the program shown in Figure 3, `Comp.var` denotes the variable `var` belongs to the component `Comp`.

```
1 // Bob.receive(data)
2 Bob.data1 := Alice.data;
3 // Bob.good()
4 Charlie.data := Bob.data2;
5 // Bob.bad()
6 Charlie.data := Bob.data1;
```

Figure 3: Simplified version of the running example.

To track flows between component variables, we label all variables of a component with the security label of the component. By extending the labeling function for variables of components, we classify `Alice.data` as A, `Bob.data1` and `Bob.data2` as B, and `Charlie.data` as C. The program does not satisfy nontransitive noninterference because there is an illegal flow from A to C; the content of `Alice.data` is directly transmitted to `Charlie.data` via `Bob.data1`. If the program, however, did not include the `bad` method in `Bob`, it would be secure with respect to the nontransitive policy.

[1] As a visual cue, we will use the green color for nontransitive and blue color for transitive notions throughout the paper.

2.1 Security notions

We now present our model language and formal definitions of security notions, i.e., transitive and nontransitive noninterference for programs. To model the essence of these characterizations, we assume a simple batch-job setting where only the initial and final memories are observable (before and after program execution). We will show how to extend our results to a language with I/O in Section 4.

Programs consist of multiple code components and a memory $M : \mathit{Var} \to \mathit{Val}$, a (total) mapping from a set of variables Var to a set of values Val, partitioned by components Cmp of the program. A variable $x_\alpha \in \mathit{Var}$ denotes x is allocated at $\alpha \in \mathit{Cmp}$. We write x where the component name is unused. Using coarse-grained labeling, each component maps to a security label, written $\Gamma_{\mathit{Cmp}} : \mathit{Cmp} \to L$. As a result, all variables of a component are annotated with the same label. Formally, $\forall \alpha \in \mathit{Cmp}. \forall x_\alpha \in \mathit{Var}. \Gamma(x_\alpha) = \Gamma_{\mathit{Cmp}}(\alpha)$ where $\Gamma : \mathit{Var} \to L$. Note that we use Var_c for the set of variables that exist in program c.

Figures 4 and 5 illustrate the syntax and semantics of our model language. An execution configuration $\langle c, M \rangle$ is a pair of a command c and a given memory M, and $\to$ introduces the transition relation between configurations. For expressions, $\langle e, M \rangle \Downarrow v$ denotes an expression e evaluates to a value v under a memory M. We write $\to^*$ for the reflexive and transitive closure of the $\to$ relation, and $\to^n$ for the n-step execution of $\to$.

We adopt *termination-insensitive* [32] noninterference that ignores information leaks resulted from termination behavior of the given program. NTNI is introduced by a termination-insensitive notion for batch-job programs [17]. We extend the model language to support I/O and lift the security notion to progress-insensitive [3].

Note that the choices of termination- and progress-sensitivity are orthogonal to nontransitivenesses of policies. Our results (in particular, the lattice encoding) can be thus replayed for other variants of noninterference.

Transitive Noninterference (TNI) For a given program, classical noninterference guarantees if two memories agree on variables at level ℓ and lower, memories after the execution of the program also agree on the variables at level ℓ and lower. Accordingly, an observer at level ℓ can see the values of the variables labeled as ℓ or lower, called ℓ*-observable* values. Transitive noninterference stipulates ℓ-observable final values of a program only depend on initial values from ℓ or lower levels.

A transitive security policy is a triple $\mathcal{T} = \langle L_\mathcal{T}, \sqsubseteq, \Gamma_\mathcal{T} \rangle$ where $L_\mathcal{T}$ is a set of security labels and $\sqsubseteq \subseteq L_\mathcal{T} \times L_\mathcal{T}$ is a binary relation that forms a partially ordered set (reflexivity, asymmetry, transitivity) on $L_\mathcal{T}$ and specifies permit-

$$e ::= v \mid x \mid e \oplus e$$
$$c ::= \mathit{skip} \mid x := e \mid \mathit{if}\ e\ \mathit{then}\ c\ \mathit{else}\ c \mid \mathit{while}\ e\ \mathit{do}\ c \mid c; c$$

Figure 4: Language syntax.

Expression Evaluation

$$\frac{}{\langle v, M\rangle \Downarrow v} \quad \text{(Value)}$$

$$\frac{}{\langle x, M\rangle \Downarrow M(x)} \quad \text{(Read)}$$

$$\frac{\langle e_1, M\rangle \Downarrow v_1 \qquad \langle e_2, M\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, M\rangle \Downarrow v_1 \oplus v_2} \quad \text{(Operation)}$$

Command Evaluation

$$\frac{}{\langle \mathit{skip}, M\rangle \rightarrow \langle \mathit{stop}, M\rangle} \quad \text{(Skip)}$$

$$\frac{\langle e, M\rangle \Downarrow v \qquad M' = M[x \mapsto v]}{\langle x := e, M\rangle \rightarrow \langle \mathit{stop}, M'\rangle} \quad \text{(Write)}$$

$$\frac{c = \mathit{if}\ e\ \mathit{then}\ c_{\mathit{true}}\ \mathit{else}\ c_{\mathit{false}} \qquad \langle e, M\rangle \Downarrow b}{\langle c, M\rangle \rightarrow \langle c_b, M\rangle} \quad \text{(If)}$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{\mathit{body}} \qquad \langle e, M\rangle \Downarrow \mathit{true}}{\langle c, M\rangle \rightarrow \langle c_{\mathit{body}}; c, M\rangle} \quad \text{(While-T)}$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{\mathit{body}} \qquad \langle e, M\rangle \Downarrow \mathit{false}}{\langle c, M\rangle \rightarrow \langle \mathit{stop}, M\rangle} \quad \text{(While-F)}$$

$$\frac{\langle c_1, M\rangle \rightarrow \langle c_1', M'\rangle}{\langle c_1; c_2, M\rangle \rightarrow \langle c_1'; c_2, M'\rangle} \quad \text{(Seq-I)}$$

$$\frac{}{\langle \mathit{stop}; c, M\rangle \rightarrow \langle c, M\rangle} \quad \text{(Seq-II)}$$

Figure 5: Language semantics.

ted flows between security levels. A labeling function $\Gamma_{\mathcal{T}}$: $Var \to L_{\mathcal{T}}$ maps a variable to a security label.

Transitive indistinguishability relation ($=_{\mathcal{T}}$) for a security label $\ell \in L_{\mathcal{T}}$ is defined as follows. Two memories are indistinguishable at level ℓ if and only if values of variables observable at the level ℓ and lower are the same.

Definition 1 (*Transitive Memory Indistinguishability*)**.** Two memories M_1 and M_2 are transitively indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $M_1 \overset{\ell}{=}_{\mathcal{T}} M_2$ if and only if $\forall x \in Var. \Gamma_{\mathcal{T}}(x) \sqsubseteq \ell \implies M_1(x) = M_2(x)$.

We define transitive noninterference based on the indistinguishability relation between memories. A (batch-job) program c satisfies *termination-insensitive transitive noninterference*, written $TNI_{TI}(\mathcal{T}, c)$, when for any two memories indistinguishable at level $\ell \in L_{\mathcal{T}}$, the computation of the program c terminates for both and the ℓ-observer cannot distinguish the final memories.

Definition 2 (*Termination-Insensitive Transitive Noninterference*)**.** A program c satisfies $TNI_{TI}(\mathcal{T}, c)$ if and only if $\forall \ell \in L_{\mathcal{T}}. \forall M_1, M_2. \Big(M_1 \overset{\ell}{=}_{\mathcal{T}} M_2 \wedge \langle c, M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle c, M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle\Big) \implies M_1' \overset{\ell}{=}_{\mathcal{T}} M_2'$.

Nontransitive Noninterference (NTNI) The nontransitive notion of noninterference demands that for a given program, changes on variables at security level ℓ can only influence variables at the levels allowed by the policy. In this condition, *ℓ-observable* values are the content of variables labeled as ℓ. Hence, nontransitive noninterference ensures that ℓ-observable final values are only dependent on those initial values that *can flow* to ℓ, as stated in the policy.

A nontransitive security policy is a triple $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$ where $L_{\mathcal{N}}$ is a set of security labels, $\Gamma_{\mathcal{N}}$: $Var \to L_{\mathcal{N}}$ is a labeling function, and $\trianglerighteq$ is an arbitrary flow relation specifying permitted flows (*can-flow-to* relation [8]). We define $C(\ell) = \{\ell' | \ell' \trianglerighteq \ell\}$ as the set of levels that can flow to ℓ, including itself. The only condition for the relation is to be reflexive; no other properties, such as transitivity, are required.

Nontransitive indistinguishability relations ($=_{\mathcal{N}}$) for a security label $\ell \in L_{\mathcal{N}}$ and a set of security labels $\mathcal{L} \subseteq L_{\mathcal{N}}$ are defined below. Two memories are indistinguishable at level ℓ if variables of the level ℓ have the same values in those two. Consistently, the relation holds for a set of labels if variables of any level existing in the set be mapped to same values in the two memories.

Definition 3 (*Nontransitive Memory Indistinguishability*)**.** Two memories M_1 and M_2 are nontransitively indistinguishable at level $\ell \in L_{\mathcal{N}}$, written

$M_1 \overset{\ell}{=}_{\mathcal{N}} M_2$, if and only if $\forall x \in Var. \Gamma_{\mathcal{N}}(x) = \ell \implies M_1(x) = M_2(x)$. The memories are indistinguishable for a set of security levels $\mathcal{L} \subseteq L_{\mathcal{N}}$, written $M_1 \overset{\mathcal{L}}{=}_{\mathcal{N}} M_2$, if and only if $\forall x \in Var. \Gamma_{\mathcal{N}}(x) \in \mathcal{L} \implies M_1(x) = M_2(x)$.

We use the indistinguishability relation between memories to define nontransitive noninterference. A (batch-job) program c satisfies *termination-insensitive nontransitive noninterference*, written $NTNI_{TI}(\mathcal{N}, c)$, if for any two memories indistinguishable for the set of levels may influence variables at $\ell \in L_{\mathcal{N}}$, the program c gets terminated for both and the ℓ-observer cannot distinguish the final memories.

Definition 4 (*Termination-Insensitive Nontransitive Noninterference*)**.** A program c satisfies $NTNI_{TI}(\mathcal{N}, c)$ if and only if

$$\forall \ell \in L_{\mathcal{N}}. \forall M_1, M_2. \Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \wedge \langle c, M_1 \rangle \rightarrow^* \langle \text{stop}, M_1' \rangle \wedge \langle c, M_2 \rangle \rightarrow^* \langle \text{stop}, M_2' \rangle \Big) \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'.$$

2.2 Relationship between *NTNI* and *TNI*

We first prove that NTNI is a generalization of TNI, and then for the other side, we introduce the transpilation from NTNI to TNI and discuss how a nontransitive policy can be seen as transitive. We present an encoding to convert nontransitive policies to transitive ones and show if a program is secure with respect to a nontransitive policy, then a semantically equivalent program satisfies an equivalent transitive policy and vice versa.

Theorem 1 (*From TNI_{TI} to $NTNI_{TI}$*)**.** *For any program c and any transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$, there exists a nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}} \rangle$ where $L_{\mathcal{N}} = L_{\mathcal{T}}$, $\trianglerighteq = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$ such that $TNI_{TI}(\mathcal{T}, c) \iff NTNI_{TI}(\mathcal{N}, c)$. Formally,*

$$\forall c. \forall \mathcal{T}. \exists \mathcal{N}. \, TNI_{TI}(\mathcal{T}, c) \iff NTNI_{TI}(\mathcal{N}, c).$$

Proof. The proofs of all statements can be found in Appendix 3.C. □

The transpilation from NTNI to TNI includes mapping the nontransitive policy to the corresponding transitive one and rewriting the given program to be compatible with the policy encoding. We establish a powerset lattice with the set of security levels. To connect these two policies together, we should map the components and their variables to the transitive labels. Prior to labeling variables, a transformation in the program is needed, which we call *canonicalization*.

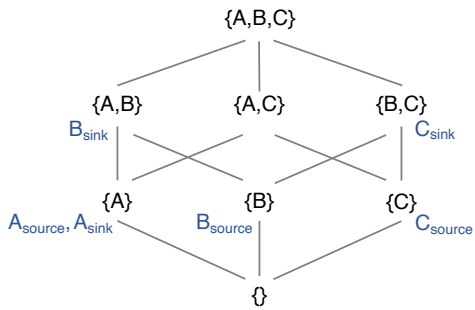


Figure 6: The powerset lattice for the running example.

In nontransitive policies, $A \trianglerighteq B$ means information from the source level A can flow to the sink level B. Therefore, we allocate two fresh variables for each component variable to capture the source and sink of information. We prepend a sequence of assignments from source variables to the component variables, and we append assignments from the component variables to sink variables. Then, we can label source and sink variables separately with respect to the encoding to preserve the notion of nontransitive policy.

Running example We describe the transpilation from NTNI to TNI for the running example shown in Figure 3. We form the powerset lattice of labels used in the nontransitive policy as the set of labels for the corresponding transitive policy, i.e., $L_T = \wp(\{A, B, C\})$ and $\sqsubseteq = \subseteq$ (see Figure 6). We transform the program to be able to capture the notion of nontransitive noninterference by assigning labels to variables. We add two fresh variables for each component variable in the given program to differentiate the source and sink of information and label them according to the definition of NTNI.

Figure 7 demonstrates the program after the transformation, which we call it the *canonical* version of the program. It consists of three sections: (1) initial assignments from a (source) variable to a `temp` variable (lines 2-5), (2) a copy of the program where variables are replaced by `temp` variables (lines 7-9), and (3) final assignments from `temp` to `sink` variables (lines 12-15). It is obvious that the meaning of the program is preserved in the transformation.

Next, we define the new labeling function for component variables. As illuminated by annotations in Figure 6, for any component variable `Comp.x` that the component `Comp` is labeled as ℓ in nontransitive policy, we label (source) variables `Comp.x` as $\{\ell\}$, `Comp.x_temp` as the top element of the security lattice, i.e., the set of all nontransitive labels, and `Comp.x_sink` as the set of nontransitive labels that can flow to the variable, i.e., $C(\ell)$. Thus, information flows from source variables (labeled $\{\ell\}$) to sink variables (labeled $C(\ell)$) are carried through internal $temp$ variables. In Section 3, we show how the presented

```
1  // init
2  Alice.data_temp := Alice.data;
3  Bob.data1_temp := Bob.data1;
4  Bob.data2_temp := Bob.data2;
5  Charlie.data_temp := Charlie.data;
6
7  Bob.data1_temp := Alice.data_temp;
8  Charlie.data_temp := Bob.data2_temp;
9  Charlie.data_temp := Bob.data1_temp;
10
11 // final
12 Alice.data_sink := Alice.data_temp;
13 Bob.data1_sink := Bob.data1_temp;
14 Bob.data2_sink := Bob.data2_temp;
15 Charlie.data_sink := Charlie.data_temp;
```

Figure 7: Canonical version of the running example.

type system updates the type of *temp* variables based on data and control flows and verifies whether the final assignments are secure.

Having the described labeling function, the canonical version of the given program does not satisfy the transitive policy. By tracking the sequence of lines 2, 7, 9, and 15 in Figure 7, an explicit flow from $\{A\}$ (level of `Alice.data`) to $\{B, C\}$ (level of `Charlie.data_sink`) is identified, which is not permitted with respect to the transitive policy ($\{A\} \not\subseteq \{B, C\}$). However, similar to the original program and the nontransitive policy, if the program did not include the undesired flow, the program would be considered secure.

Algorithm 1: Canonicalization algorithm for batch-job programs.

Input : Program c
Output: Program $Canonical(c)$
$init :=$ “”
$final :=$ “”
foreach $x \in Var_c$ **do**
  $c[x \mapsto x_{temp}]$
  $init := init$ ++ “$x_{temp} := x;$”
  $final := final$ ++ “$; x_{sink} := x_{temp}$”
end
$Canonical(c) := init$ ++ c ++ $final$
return *Canonical(c)*

Program canonicalization Algorithm 1 explains the transformation for batch-job programs. First, for each variable x in the program, we allocate two fresh variables $x_{temp}, x_{sink} \in Var \setminus Var_c$, and then apply the following transformation on the given program. We use ++ to denote the operator for string concatenation and the notation $c[x \mapsto x_{temp}]$ indicates renaming all occurrences of x in program c to x_{temp} (in a capture-avoiding manner). We use Var_{temp} and Var_{sink} to point to the set of *temp* and *sink* variables, respectively.

We prove that the canonical version of the program keeps the meaning and termination behavior of the original program, yet the final values of variables are in the $sink$ variables.

Lemma 1 (*Semantic Equivalence Modulo Canonicalization*)**.** *For any program c, the semantic equivalence $\simeq_C$ between the programs c and $Canonical(c)$ holds, where*

$$c \simeq_C c' \triangleq \forall M.\Big(\langle c, M\rangle \rightarrow^* \langle stop, M'\rangle \iff \langle c', M\rangle \rightarrow^* \langle stop, M''\rangle\Big) \wedge$$
$$\forall x \in Var_c.\Big(M'(x) = M''(x_{temp}) = M''(x_{sink}) \wedge M(x) = M''(x)\Big).$$

The following lemmas are intermediate steps to show how a nontransitive policy on a given program is reduced to a transitive policy using the powerset lattice resulted from the set of nontransitive labels in combination with the canonical version of the program. Lemma 2 proves that the transformation holds a program secure with respect to a nontransitive policy if and only if the original program is secure.

Lemma 2 ($NTNI_{TI}$ *Preservation under Canonicalization*)**.** *Any program c is secure with respect to a nontransitive security policy $\mathcal{N}$ if and only if the canonical program $Canonical(c)$ is secure where $\forall x \in Var_c.\Gamma_{\mathcal{N}}(x_{temp}) = \Gamma_{\mathcal{N}}(x_{sink}) = \Gamma_{\mathcal{N}}(x)$. Formally,*

$$\forall c.\forall \mathcal{N}.NTNI_{TI}(\mathcal{N}, c) \iff NTNI_{TI}(\mathcal{N}, Canonical(c)).$$

We define the powerset encoding of a nontransitive policy to a transitive policy for canonical programs as follows.

Definition 5 (*Transitive Encoding of Nontransitive Policies*)**.** Given a nontransitive policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$ and a program c, the corresponding transitive policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ on the canonical version of the program is $L_{\mathcal{T}} = \wp(L_{\mathcal{N}}), \sqsubseteq = \subseteq$, and

$$\forall x \in Var_c.\begin{cases} \Gamma_{\mathcal{T}}(x) & = & \{\Gamma_{\mathcal{N}}(x)\} \\ \Gamma_{\mathcal{T}}(x_{temp}) & = & L_{\mathcal{N}} \\ \Gamma_{\mathcal{T}}(x_{sink}) & = & C(\Gamma_{\mathcal{N}}(x)) \end{cases}.$$

As stated in Definition 5, the initial and final values of an ℓ-observable variable x of the given program are $\{\ell\}$- and $C(\ell)$-observable in the canonical version, respectively. Also, *temp* variables are internal and the top-level observer only can see their final values, thus $L_{\mathcal{N}}$-observable. The next lemma demonstrates for any canonical program satisfying a nontransitive policy, the program also complies with a corresponding transitive policy and vice versa.

Lemma 3 (*From $NTNI_{TI}$ to TNI_{TI} for Canonical Programs*)**.** *Any canonical program $Canonical(c)$ is secure with respect to a nontransitive security policy $\mathcal{N}$ where $\forall x \in Var_c.\, \Gamma_{\mathcal{N}}(x_{temp}) = \Gamma_{\mathcal{N}}(x_{sink}) = \Gamma_{\mathcal{N}}(x)$ if and only if the canonical program is secure according to the corresponding transitive security policy $\mathcal{T}$. We write $\forall c. \forall \mathcal{N}. \exists \mathcal{T}.\ NTNI_{TI}(\mathcal{N}, Canonical(c)) \iff TNI_{TI}(\mathcal{T}, Canonical(c))$.*

Finally, by connecting the previous lemmas, we prove that any nontransitive policy on a given program can be modeled as a transitive policy on the canonical version of the program. Given Theorems 1 and 2, the two notions of *transitive* and *nontransitive* noninterference coincide.

Theorem 2 (*From $NTNI_{TI}$ to TNI_{TI}*)**.** *For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}} \rangle$, there exist a semantically equivalent (modulo canonicaliztion) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$, as specified in Definition 5, such that $NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c')$. Formally,*

$$\forall \mathcal{N}. \forall c. \exists \mathcal{T}. \exists c'.\, c \simeq_C c' \wedge NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c').$$

3 Enforcement transpiled

The proposed enforcement mechanism for nontransitive policies [17] is a type system that does not use subtyping, the classical way to check transitive types, for information flow verification. Instead, it tracks dependencies between program variables and collects all security labels of flows into a component variable throughout the program. Then it checks whether the flows comply with the specified policy. Therefore, the type system can enforce both nontransitive and transitive policies.

To enforce a nontransitive policy, however, we can benefit from the transpilation introduced in Section 2 and devise a transitive type system for canonical programs. We employ a (vanilla) flow-sensitive type system [14] enforcing the corresponding transitive policy on transformed programs. The

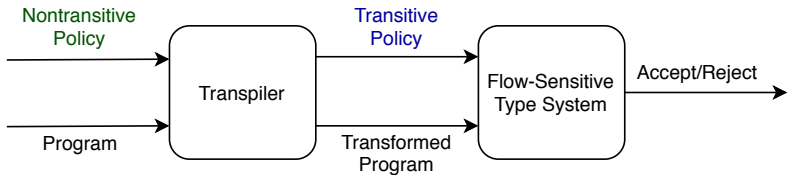


Figure 8: Composition of transpiler and enforcement mechanism.

flow-sensitivite type system investigates how components influence variables of the program. Figure 8 illustrates the composition of the transpiler and the enforcement mechanism.

We prove soundness of our transitive type system (Figure 9) and investigate how it relates to the nontransitive type system. Inspired by the notion, we present a nontransitive type system for our model language (Figure 10) and prove the soundness property. Then, we show that the flow-sensitive transitive type system accepts more secure programs compared to the nontransitive one.

3.1 Enforcement mechanism

We present a flow-sensitive type system that enforces transitive policies for canonical programs. The type system allows updates of security types through typing the program. When an expression is assigned to a variable, the security type of the variable changes to the join of security types of the expression and the program counter, to capture explicit and implicit flows (arisen from control dependencies) to the variable.

For a command c, judgments are in the form of $pc \vdash \Gamma\{c\}\Gamma'$, where $pc \in L_{\mathcal{T}}$ is the program counter label and the typing environment $\Gamma : Var \to L_{\mathcal{T}}$ will be updated to Γ' after execution of c. We make use of the structure of canonical programs in the typing rules, presented in Figure 9. The two rules for assignments (rules TT-Write-I and TT-Write-II) represent the essence of the type system. We know that only $temp$ and $sink$ variables can be on the left-hand side of an assignment in a canonical program. Assignments to $sink$ variables occur at the end of the program, i.e., the final section, where the right-hand side of assignments are $temp$ variables (rule TT-Write-II). The type system allows changes to the security types, except for $sink$ variables, whose initial types must be kept (rule TT-Sub). Otherwise, upgrading security levels of $sink$ variables might violate the soundness property of the type system.

$$\frac{}{\Gamma \vdash v : \bot} \quad \text{(TT-Value)}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \text{(TT-Read)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \sqcup t_2} \quad \text{(TT-Operation)}$$

$$\frac{}{pc \vdash \Gamma\{skip\}\Gamma} \quad \text{(TT-Skip)}$$

$$\frac{\Gamma \vdash e : t \qquad x \in Var_{temp}}{pc \vdash \Gamma\{x := e\}\Gamma[x \mapsto pc \sqcup t]} \quad \text{(TT-Write-I)}$$

$$\frac{x' \in Var_{temp} \qquad x \in Var_{sink} \qquad pc \sqcup \Gamma(x') \sqsubseteq \Gamma(x)}{pc \vdash \Gamma\{x := x'\}\Gamma} \quad \text{(TT-Write-II)}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{c_{true}\}\Gamma' \qquad pc \sqcup t \vdash \Gamma\{c_{false}\}\Gamma'}{pc \vdash \Gamma\{\mathit{if}\ e\ \mathit{then}\ c_{true}\ \mathit{else}\ c_{false}\}\Gamma'} \quad \text{(TT-If)}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{c_{body}\}\Gamma}{pc \vdash \Gamma\{\mathit{while}\ e\ \mathit{do}\ c_{body}\}\Gamma} \quad \text{(TT-While)}$$

$$\frac{pc \vdash \Gamma\{c_1\}\Gamma' \qquad pc \vdash \Gamma'\{c_2\}\Gamma''}{pc \vdash \Gamma\{c_1; c_2\}\Gamma''} \quad \text{(TT-Seq)}$$

$$\frac{pc_1 \vdash \Gamma_1\{c\}\Gamma_1' \qquad pc_2 \sqsubseteq pc_1 \qquad \Gamma_2 \sqsubseteq \Gamma_1 \qquad \Gamma_1' \sqsubseteq \Gamma_2' \qquad \forall x \in Var_{sink}.\, \Gamma_1(x) = \Gamma_2(x) = \Gamma_1'(x) = \Gamma_2'(x)}{pc_2 \vdash \Gamma_2\{c\}\Gamma_2'} \quad \text{(TT-Sub)}$$

Figure 9: Transitive typing rules.

$$\frac{}{\Gamma \vdash v : \varnothing} \quad \text{(NT-Value)}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \text{(NT-Read)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \cup t_2} \quad \text{(NT-Operation)}$$

$$\frac{}{\mathcal{P}, \Gamma, pc \vdash skip : t} \quad \text{(NT-Skip)}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash x : t \qquad \forall \ell \in t \cup pc . \ell \in \Gamma(x) \wedge \ell \trianglerighteq \mathcal{P}(x)}{\mathcal{P}, \Gamma, pc \vdash x := e : t} \quad \text{(NT-Write)}$$

$$\frac{\Gamma \vdash e : t_1 \qquad \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{true} : t_2 \qquad \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{false} : t_2}{\mathcal{P}, \Gamma, pc \vdash \mathit{if}\ e\ \mathit{then}\ c_{true}\ \mathit{else}\ c_{false} : t_1 \cup t_2} \quad \text{(NT-If)}$$

$$\frac{\Gamma \vdash e : t_1 \qquad \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{body} : t_2}{\mathcal{P}, \Gamma, pc \vdash \mathit{while}\ e\ \mathit{do}\ c_{body} : t_1 \cup t_2} \quad \text{(NT-While)}$$

$$\frac{\mathcal{P}, \Gamma, pc \vdash c_1 : t_1 \qquad \mathcal{P}, \Gamma, pc \vdash c_2 : t_2}{\mathcal{P}, \Gamma, pc \vdash c_1 ; c_2 : t_1 \cup t_2} \quad \text{(NT-Seq)}$$

$$\frac{\Gamma \vdash e : t_1 \qquad t_1 \subseteq t_2}{\Gamma \vdash e : t_2} \quad \text{(NT-Sub-I)}$$

$$\frac{\mathcal{P}, \Gamma, pc_1 \vdash c : t_1 \qquad pc_2 \subseteq pc_1 \qquad t_1 \subseteq t_2}{\mathcal{P}, \Gamma, pc_2 \vdash c : t_2} \quad \text{(NT-Sub-II)}$$

Figure 10: Nontransitive typing rules.

Running example Given the policy specified in the running example, the type system rejects the canonical program shown in Figure 7. The initial types of the variables are the sets of labels introduced in Definition 5. Applying the typing rules, the types of the variables `Alice.data_temp`, `Bob.data1_temp`, and `Charlie.data_temp` are (at least) the same as the type of `Alice.data`, which is $\{A\}$. The assignments in the final section are well-typed except for the last one, where the type of `Charlie.data_sink` is the set of labels can flow to C, i.e., $\{B, C\}$. Since $\{A\} \not\subseteq \{B, C\}$, the program is ill-typed with respect to the given nontransitive policy. We will discuss more examples in Section 5.

The next theorem states soundness of the flow-sensitive type system, which means if the type system accepts a canonical program, then the program satisfies the transitive noninterference, and consequently, the original program complies with the nontransitive policy.

Theorem 3 (*Soundness of Flow-Sensitive Transitive Type System*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\{Canonical(c)\}\Gamma' \implies TNI_{TI}(\mathcal{T}, Canonical(c)).$$

3.2 Relationship between nontransitive and flow-sensitive transitive type systems

The core idea of Lu and Zhang's type system [17] is tracking data and control dependencies between program variables through type inference on information propagation history. Then it guarantees flow relations from inferred labels of dependencies to the specified label of the variable are stated in the policy. Their flow-insensitive type system captures all possible dependencies to a variable; thus it becomes less permissive in comparison with a flow-sensitive type system. Given the semantic relationship between nontransitive and transitive policies, we demonstrate our flow-sensitive transitive type system accepts all the well-typed programs in the nontransitive type system, and more secure programs.

We present a nontransitive type system for our imperative model language based on the essence of their type system. It aggregates security labels of data and control dependencies of variables through the program. For each assignment $x := e$, the type system checks permission of information flows from the collected labels of the expression e and the program counter to the specified label of the variable x.

Typing judgments are in the form of $\mathcal{P}, \Gamma, pc \vdash c : t$ that indicates the type t is assigned to the command c with respect to the program counter label $pc \subseteq L_{\mathcal{N}}$ in the typing environments $\mathcal{P} : Var \to L_{\mathcal{N}}$ and $\Gamma : Var \to \wp(L_{\mathcal{N}})$

where $\forall x \in Var.\mathcal{P}(x) \in \Gamma(x)$. Figure 10 illustrates the typing rules where $\mathcal{P}$ specifies the nontransitive levels of variables and Γ predicts the set of labels that might influence the final value of a variable in the program.

The most important rule is the one for typing assignments (rule NT-Write). The set $\Gamma(x)$ must contain all possible information flows to the variable x in the program, which is checked in the premise ($t \cup pc \subseteq \Gamma(x)$), and then the type system verifies whether those are permitted flows or not ($\forall \ell \in t \cup pc.\ell \trianglerighteq \mathcal{P}(x)$). Note that the specified label of a variable $\mathcal{P}(x)$ must be present in the set of dependencies $\Gamma(x)$ because the $\trianglerighteq$ relation is reflexive.

Running example The nontransitive type system tracks and collects all the security labels a variable has a dependency on through the program and checks whether they are compliant with the permitted flows. Therefore, the program presented in Figure 3 is rejected by the type system because the type of `Bob.data1` must be (at least) $\{A, B\}$ to record the type of `Alice.data`, which is $\{A\}$ and $A \trianglerighteq B$ exists in the policy. Consequently, the last assignment is ill-typed respecting the typing environment and absence of $A \trianglerighteq C$ in the policy.

In the following, we prove soundness of the nontransitive type system. Any well-typed program with respect to the nontransitive typing rules satisfies nontransitive noninterference.

Theorem 4 (*Soundness of Nontransitive Type System*)**.**

$$\mathcal{P}, \Gamma, pc \vdash c : t \implies NTNI_{TI}(\mathcal{N}, c).$$

On closer inspection, both type systems are sound but the nontransitive type system is not as permissive as the flow-sensitive mechanism. The flow-sensitive transitive type system updates the labels of variables based on the flow of the program in a more precise manner. The next theorem shows if a program is secure under the nontransitive type system, the flow-sensitive type system accepts the canonical version of the program as well.

Theorem 5 (*Flow-Sensitive Type System Covers Nontransitive Type System*)**.**

$$\mathcal{P}, \Gamma_1, pc \vdash c : t \implies pc \vdash \Gamma_2\{Canonical(c)\}\Gamma_3,$$

where $\forall x \in Var_c.\Gamma_3(x_{temp}) \sqsubseteq \Gamma_1(x) \wedge \Gamma_2(x) = \Gamma_3(x) = \{\mathcal{P}(x)\} \wedge \Gamma_2(x_{temp}) = L_{\mathcal{N}} \wedge \Gamma_2(x_{sink}) = \Gamma_3(x_{sink}) = C(\mathcal{P}(x))$.

The counterexample program in Figure 11 demonstrates the theorem does not hold in the other direction; there is a well-typed program according to the flow-sensitive rules, which gets rejected by the nontransitive type system. If we swap the last two statements of the running example, as shown

in Figure 11, the nontransitive type system still rejects the program; types of both sides of an assignment must be the same (rule NT-Write). The flow-sensitive type system, however, accepts the program because it detects that the last assignment overwrites the final value of `Charlie.data` and updates the label accordingly (rule TT-Write-I). It can be shown that adding flow-sensitivity flavor to the nontransitive type system enhances precision to the same level offered by the flow-sensitive transitive type system.

```
1 // Bob.receive(data)
2 Bob.data1 := Alice.data;
3 // Bob.bad()
4 Charlie.data := Bob.data1;
5 // Bob.good()
6 Charlie.data := Bob.data2;
```

Figure 11: An example that shows the flow-sensitive type system is more permissive than the nontransitive type system.

4 Extension with I/O

We extend the model language to support input and output commands. In this setting, sources and sinks of information are more tangible, as a better fit for real-world programs with third-party components. Interestingly, we will observe a more natural correspondence between nontransitive and transitive security notions.

4.1 Security notions

Programs can receive inputs and produce outputs at any step of computation. We include two new constructs $input(x, \ell)$ and $output(x, \ell)$ for reading a value from the input channel at security level ℓ and sending a value to the output channel at level ℓ, respectively. This model entails a revision on security notions where intermediate output values are observable as well as the termination behavior of a program.

We naturally choose another notion of noninterference named *progress-insensitive* [3, 13] (corresponding to *CP-security* for reactive programs [5]) that demands if two program inputs agree on values at security levels may influence variables at ℓ, the output sequence observable at level ℓ remains the same up to the point that one of the executions diverges silently (without producing any output). Transitive policies define an input/output value

ℓ-*observable* if the value is at level ℓ or lower, while an ℓ-observer in a nontransitive policy only sees values at level ℓ. Note that the termination behavior of a program is observable for all security levels in both security notions.

Running example Recall the nontransitive policy of the running example in Section 2: $A \trianglerighteq B$ and $B \trianglerighteq C$. The program in Figure 12 violates progress-insensitive nontransitive noninterference due to the presence of an implicit flow from the input value of `Alice.data` with security level A to the observable output at level C. Based on the input value, the program sends an output value at level B or C. Therefore, the observable outputs are different at levels B and C, depending on the input value at level A.

```
1 input(Alice.data, A);
2 Bob.data1 := Alice.data;
3 if Bob.data1 then
4   output(Bob.data2, B);
5 else
6   output(Charlie.data, C);
```

Figure 12: Running example with I/O.

Figure 13 illustrates the syntax of our model language supporting I/O. Evalution rules for input and output commands are presented in Figure 14. We refer to Figure 24 (in Appendix) for the complete set of semantic rules. An execution configuration $\langle c, M, I, O\rangle$ is a tuple consists of a command c, a memory M, an input function I that maps security levels to input channels, and an output channel O. The relation $\rightarrow$ defines transitions between configurations. We assume the environment is input total. We model program inputs as a mapping from security levels to sequences of values, written $I(\ell) = v.\sigma$, where $\ell \in L$, $v \in Val$, and σ is a sequence of values. We define output behavior of a program recursively by $O = \varnothing \mid \circlearrowleft \mid v_\ell.\, O$, where $\circlearrowleft$ denotes silent divergence. Based on the language semantics, we abstract away details of computation steps and define output evaluation of an execution. Definition 6 introduces the new relation $\rightsquigarrow$ that indicates an initial configuration $\langle c, M, I, \varnothing\rangle$ evaluates to O.

$$
\begin{aligned}
e &::= v \mid x \mid e \oplus e \\
c &::= \mathit{skip} \mid x := e \mid \mathit{if}\ e\ \mathit{then}\ c\ \mathit{else}\ c \mid \mathit{while}\ e\ \mathit{do}\ c \mid c; c \mid \\
&\quad \mathit{input}(x, \ell) \mid \mathit{output}(x, \ell)
\end{aligned}
$$

Figure 13: Language syntax with I/O.

$$\frac{c = input(x, \ell) \qquad I(\ell) = v.\sigma \qquad I' = I[\ell \mapsto \sigma] \qquad M' = M[x \mapsto v]}{\langle c, M, I, O\rangle \to \langle stop, M', I', O\rangle} \text{ (IO-Input)}$$

$$\frac{c = output(x, \ell) \qquad M(x) = v \qquad O' = O.\, v_\ell}{\langle c, M, I, O\rangle \to \langle stop, M, I, O'\rangle} \text{ (IO-Output)}$$

Figure 14: Language semantics with I/O (selected rules).

Definition 6 (*Output Behavior of A Program Execution*)**.** The output behavior O generated by an initial execution configuration $\langle c, M, I, \varnothing\rangle$, written $\langle c, M, I, \varnothing\rangle \rightsquigarrow O$, is defined as follows:

$$\frac{\langle c, M, I, \varnothing\rangle \to^* \langle stop, M', I', O\rangle}{\langle c, M, I, \varnothing\rangle \rightsquigarrow O}$$

$$\frac{\langle c, M, I, \varnothing\rangle \to^* \langle c', M', I', O\rangle \qquad \forall n \in \mathbb{N}. \langle c', M', I', O\rangle \to^n \langle c_n, M_n, I_n, O\rangle \wedge c_n \neq stop}{\langle c, M, I, \varnothing\rangle \rightsquigarrow O.\circlearrowleft}.$$

Transitive Noninterference (TNI) Classical noninterference guarantees ℓ-observable output behavior of a program only depends on inputs from ℓ or lower levels. A transitive security policy $\mathcal{T} = \langle L_\mathcal{T}, \sqsubseteq, \Gamma_\mathcal{T}\rangle$ is a triple where $L_\mathcal{T}$ is a set of security labels and $\sqsubseteq \subseteq L_\mathcal{T} \times L_\mathcal{T}$ is a binary relation that specifies permitted flows between security levels forming a partially ordered set on $L_\mathcal{T}$. A labeling function $\Gamma_\mathcal{T}$: $Var \to L_\mathcal{T}$ maps a variable to a security label.

The definition of progress-insensitive noninterference relies on the definition of indistinguishability relations for inputs and outputs. To define the relations, we should first describe observable inputs and outputs at a security level ℓ. An ℓ-observer can see the content of input channels at the security level ℓ and lower. We define observable output behavior at a level $\ell \in L_\mathcal{T}$ by purging the values from an output sequence which are not at the level ℓ or lower.

Definition 7 (*Transitive Observable Output Behavior*)**.** Given an output behavior O including a sequence of output values and termination behavior of a program execution. The subsequence of the output behavior observable at

a security level $\ell \in L_{\mathcal{T}}$ is defined below:

$$O|_{\ell}^{\mathcal{T}} = \begin{cases} O & O = \varnothing \vee O = \circlearrowleft \\ v_{\ell'}.\, O'|_{\ell}^{\mathcal{T}} & O = v_{\ell'}.\, O' \wedge \ell' \sqsubseteq \ell\,. \\ O'|_{\ell}^{\mathcal{T}} & otherwise \end{cases}$$

We call two program inputs indistinguishable at level $\ell \in L_{\mathcal{T}}$ if input sequences of the levels ℓ are the same as well as lower levels.

Definition 8 (*Transitive Input Indistinguishability*)**.** Two program inputs I_1 and I_2 are indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $I_1 \overset{\ell}{=}_{\mathcal{T}} I_2$, if and only if $\forall \ell' \sqsubseteq \ell.\, I_1(\ell') = I_2(\ell')$.

Two program outputs are indistinguishable at level ℓ when the sequences of observable outputs are exactly the same up to the silent divergence in one of them. In other words, if both of the output behaviors are terminating, then the ℓ-observable subsequences must be identical. Otherwise, the subsequences must be the same until one of them reaches the $\circlearrowleft$ event.

Definition 9 (*Transitive Output Indistinguishability*)**.** Two program outputs O_1 and O_2 are indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $O_1 \overset{\ell}{=}_{\mathcal{T}} O_2$, if and only if $O_1|_{\ell}^{\mathcal{T}} = O_2|_{\ell}^{\mathcal{T}} \vee (\exists O, O'.\, O_1|_{\ell}^{\mathcal{T}} = O.\, \circlearrowleft \wedge O_2|_{\ell}^{\mathcal{T}} = O.\, O') \vee (\exists O, O'.\, O_1|_{\ell}^{\mathcal{T}} = O.\, O' \wedge O_2|_{\ell}^{\mathcal{T}} = O.\, \circlearrowleft)$.

Given the indistinguishability definitions, we are ready to define the security condition. A program c satisfies *progress-insensitive transitive noninterference*, written $TNI_{PI}(\mathcal{T}, c)$, when for any two program inputs indistinguishable at level $\ell \in L_{\mathcal{T}}$, the output behaviors resulted from the execution of the program are indistinguishable for the ℓ-observer.

Definition 10 (*Progress-Insensitive Transitive Noninterference*)**.** A program c satisfies $TNI_{PI}(\mathcal{T}, c)$ if and only if $\forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2.\, I_1 \overset{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1 \implies \exists O_2. \langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \wedge O_1 \overset{\ell}{=}_{\mathcal{T}} O_2$.

Nontransitive Noninterference (NTNI) The nontransitive notion of noninterference stipulates that ℓ-observable output behavior of a given program is only dependent on those inputs that *can flow* to ℓ, as stated in the policy. A nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$ is a triple where $L_{\mathcal{N}}$ is a set of security labels, $\trianglerighteq$ is an arbitrary flow relation specifying permitted flows, and $\Gamma_{\mathcal{N}}$: $Var \to L_{\mathcal{N}}$ is a labeling function.

Similar to the transitive notion, we define indistinguishability relations for program inputs and outputs with respect to definitions of observable inputs and outputs at a security level, respectively. An ℓ-observer can see the content of the input channel at the level ℓ and the subsequence of output values at the level ℓ as well as the divergence event.

Definition 11 (*Nontransitive Observable Output Behavior*)**.** Given an output behavior O including a sequence of output values and termination behavior of a program execution. The subsequence of the output behavior observable at a security level $\ell \in L_{\mathcal{N}}$ is defined as follows:

$$O|_\ell^{\mathcal{N}} = \begin{cases} O & O = \varnothing \vee O = \circlearrowleft \\ v_\ell . O'|_\ell^{\mathcal{N}} & O = v_\ell . O' \\ O'|_\ell^{\mathcal{N}} & otherwise \end{cases} .$$

Two program inputs are indistinguishable for a set of levels $\mathcal{L} \subseteq L_{\mathcal{N}}$ if input sequences of the levels member of $\mathcal{L}$ are identical with each other.

Definition 12 (*Nontransitive Input Indistinguishability*)**.** Two program inputs I_1 and I_2 are indistinguishable for a set of levels $\mathcal{L} \subseteq L_{\mathcal{N}}$, written $I_1 \overset{\mathcal{L}}{=}_{\mathcal{N}} I_2$, if and only if $\forall \ell \in \mathcal{L}. I_1(\ell) = I_2(\ell)$.

Similar to Definition 9, two program outputs are indistinguishable at level $\ell \in L_{\mathcal{N}}$ if the sequences of observable outputs are the same until one of the executions diverges silently.

Definition 13 (*Nontransitive Output Indistinguishability*)**.** Two program outputs O_1 and O_2 are indistinguishable at level $\ell \in L_{\mathcal{N}}$, written $O_1 \overset{\ell}{=}_{\mathcal{N}} O_2$, if and only if $O_1|_\ell^{\mathcal{N}} = O_2|_\ell^{\mathcal{N}} \vee (\exists O, O'. O_1|_\ell^{\mathcal{N}} = O.\circlearrowleft \wedge O_2|_\ell^{\mathcal{N}} = O.O') \vee (\exists O, O'. O_1|_\ell^{\mathcal{N}} = O.O' \wedge O_2|_\ell^{\mathcal{N}} = O.\circlearrowleft)$.

Having the indistinguishability relations in hand, we define the noninterference notion for the nontransitive setting. A program c satisfies *progress-insensitive nontrasnitive noninterference*, written $NTNI_{PI}(\mathcal{N}, c)$, when for any two program inputs indistinguishable for the set of levels may influence variables at level $\ell \in L_{\mathcal{N}}$, the output behaviors resulted from the execution of the program are indistinguishable for the ℓ-observer.

Definition 14 (*Progress-Insensitive Nontransitive Noninterference*)**.** A program c satisfies $NTNI_{PI}(\mathcal{N}, c)$ if and only if

$$\forall \ell \in L_{\mathcal{N}}. \forall M. \forall I_1, I_2. I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \wedge \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1 \implies \exists O_2. \langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \wedge O_1 \overset{\ell}{=}_{\mathcal{N}} O_2.$$

4.2 Relationship between *NTNI* and *TNI*

We follow the same pattern to relate nontransitive and transitive security definitions together. Constructing the power-lattice encoding remains as before, although the transformation algorithm is more straightforward for programs with input/outputs. Before we see that, the next theorem confirms NTNI is still a generalization of TNI using the progress-insensitive notion in the security definitions.

Theorem 6 (*From TNI_{PI} to $NTNI_{PI}$*). *For any program c and any transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$, there exists a nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}} \rangle$ where $L_{\mathcal{N}} = L_{\mathcal{T}}, \trianglerighteq = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$ such that $TNI_{PI}(\mathcal{T}, c) \iff NTNI_{PI}(\mathcal{N}, c)$. Formally,*

$$\forall c. \forall \mathcal{T}. \exists \mathcal{N}.\ TNI_{PI}(\mathcal{T}, c) \iff NTNI_{PI}(\mathcal{N}, c).$$

We introduce the transpilation for programs with intermediate input/outputs. Similar to the batch-job style, we establish the powerset lattice out of nontransitive labels, i.e., $L_{\mathcal{T}} = \wp(L_{\mathcal{N}})$ and $\sqsubseteq = \subseteq$. However, the transformation algorithm is quite simpler than canonicalization; only input and output commands are required to be rewritten because of the new security definition that considers only the relation between program inputs and outputs.

Program transformation As explained in Algorithm 2, we label sources of information at a security level $\ell \in L_{\mathcal{N}}$ as the singleton set of a security level ($\{\ell\}$) and annotate sinks as the set of labels that can flow to ℓ, or $C(\ell)$. More precisely, we replace $input(x, \ell)$ commands with $input(x, \{\ell\})$, and also $output(x, \ell)$ commands with $output(x, C(\ell))$ in the program.

Algorithm 2: Transformation algorithm for programs with I/O.

Input : Program c
Output: Program $Transform(c)$
foreach $x \in Var_c$ **do**
  $c[input(x, \ell) \mapsto input(x, \{\ell\})]$
  $c[output(x, \ell) \mapsto output(x, C(\ell))]$
end
$Transform(c) := c$
return *Transform(c)*

Running example Figure 15 demonstrates how the transformation works on the running example. Each output command explicitly specifies the set

```
input(Alice.data, {A});
Bob.data1 := Alice.data;
if Bob.data1 then
  output(Bob.data2, {A,B});
else
  output(Charlie.data, {B,C});
```

Figure 15: Transformed version of running example with I/O.

of labels that are permitted to influence the output value. The transformed program does not satisfy transitive noninterference because the presence of output value at level $\{B, C\}$ depends on an input value at level $\{A\}$, which are incomparable in the security lattice. However, the flow from the input value to the output value at level $\{A, B\}$ is permitted because $\{A\} \subseteq \{A, B\}$.

It is obvious that the transformed version of a given program preserves the meaning and termination behavior of the original program, yet it changes the channel of output values. The input and output values at the level ℓ can be found on the input channel with label $\{\ell\}$ and the output channel labeled as $C(\ell)$ in the canonical version of the given program. The next lemma shows the semantic relation between a given program and the transformed one.

Lemma 4 (*Semantic Equivalence Modulo Transformation*)**.** *For any program c, the semantic equivalence $\simeq_T$ between the programs c and $Transform(c)$ holds where $c \simeq_T c' \triangleq \forall M.\forall I.\exists I'.\Big(\forall \ell.I(\ell) = I'(\{\ell\})\Big) \wedge \langle c, M, I, \varnothing\rangle \rightsquigarrow O \wedge \langle c', M, I', \varnothing\rangle \rightsquigarrow O' \wedge O' = O\,[v_\ell \mapsto v_{C(\ell)}]$.*

Then, we prove a nontransitive policy on a given program (with intermediate inputs/outputs) can be reduced to a transitive policy on the transformed version of the program. Theorems 6 and 7 demonstrate the mutual relationship between NTNI and TNI holds, even for programs with intermediate observable values.

Theorem 7 (*From $NTNI_{PI}$ to TNI_{PI}*)**.** *For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$, there exist a semantically equivalent (modulo transformation) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ where $c' = Transform(c)$, $L_{\mathcal{T}} = \wp(L_{\mathcal{N}})$, $\sqsubseteq = \subseteq$ and $\forall x \in Var_c.\,\Gamma_{\mathcal{T}}(x) = \{\Gamma_{\mathcal{N}}(x)\}$ such that $NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c')$. Formally,*

$$\forall \mathcal{N}.\,\forall c.\,\exists \mathcal{T}.\,\exists c'.\,c \simeq_T c' \wedge NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c').$$

4.3 Enforcement mechanism

Figure 16 illustrates an excerpt from a flow-sensitive type system enforcing transitive policies on transformed programs. We refer to Figure 25 (in Appendix) for the complete set of typing rules. The type system defines judgments of the form $pc \vdash \Gamma\{c\}\Gamma'$ where $pc \in L_{\mathcal{T}}$ is the program counter label, and the typing environments $\Gamma : Var \to L_{\mathcal{T}}$ and Γ' describe the security levels of variables before and after executing the command c, respectively. Security types of the variables get updated freely through the program and capture the information flows to the variable (rule IO-TT-Write).

$$\frac{\Gamma \vdash e : t}{pc \vdash \Gamma\{x := e\}\Gamma[x \mapsto pc \sqcup t]} \quad \text{(IO-TT-Write)}$$

$$\frac{pc \sqsubseteq \ell}{pc \vdash \Gamma\{input(x, \ell)\}\Gamma[x \mapsto \ell]} \quad \text{(IO-TT-Input)}$$

$$\frac{pc \sqcup \Gamma(x) \sqsubseteq \ell}{pc \vdash \Gamma\{output(x, \ell)\}\Gamma} \quad \text{(IO-TT-Output)}$$

Figure 16: Flow-sensitive typing rules with I/O (selected rules).

The rules for typing input and output commands are the most important ones. The typing environments before and after executing an output command stay the same if the explicit flows ($\Gamma(x)$) and implicit flows (pc) are permitted to the level of the specified output channel (rule IO-TT-Output). For an input command $input(x, \ell)$, the level of variable x is updated to ℓ if the program context does not make an illegal implicit flow (rule IO-TT-Input). Otherwise, it might violate soundness of the enforcement mechanism for programs like Figure 17, where the execution of an input command in a high context influences the received value of the next input command at the same level.

Running example Given the policy specified in the running example, the type system rejects the transformed program shown in Figure 15. The initial types of the variables are the singleton set of the nontransitive security label. Following the typing rules, the types of the variables `Alice.data_temp` and `Bob.data1_temp` are (at least) $\{A\}$. The rule for output commands demands that the specified level of the output value must be higher than union of the level of the program context and the level of variable x. The *if* branch is well-typed because $\{A\} \sqcup \{B\} \sqsubseteq \{A, B\}$, yet the type system cannot offer a suitable type for the *else* branch where $\{A\} \sqcup \{B\} \not\sqsubseteq \{B, C\}$.

```
1 if High.h then
2   input(Low.x,{L})
3 else
4   skip;
5 input(Low.y,{L});
6 output(Low.y,{L});
```

Figure 17: An example that shows an implicit flow by input commands.

Theorem 8 states soundness of the type system. If a transformed program is well-typed, then it satisfies the transitive noninterference, and by the result of Theorem 7, the original program complies with the corresponding nontransitive policy.

Theorem 8 (*Soundness of Flow-Sensitive Type System for Programs with I/O*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\{Transform(c)\}\Gamma' \implies TNI_{PI}(\mathcal{T}, Transform(c)).$$

5 Case study with JOANA

We develop a prototype of our transpiler to analyze Java programs. We follow the architecture illustrated in Figure 8 to implement a program canonicalizer and an input script generator for JOANA [11], a flow-sensitive information-flow analyzer for Java programs. The transpiler gets a path to a Java project and generates the canonical version of the program using Spoon [21], a library for transforming Java programs. The user defines a nontransitive policy by labeling the components (i.e., classes) of the program. Then, our tool generates a script as the input of JOANA, which detects possible illegal flows in the program. Our proof-of-concept implementation can support as many programs as JOANA may allow, as long as they are batch-job programs.

We evaluate our tool on four examples of nontransitive policies to demonstrate the benefits of the reduction from nontransitive to transitive policies in practice: Alice-Bob-Charlie (the running example), Confused deputy, Bank logger, and Low-High. The source code and materials of case studies are available online [1]. We discuss the details of transpilation and the JOANA's script for the running example, and to conserve space, we only report analysis results for the next cases. In Appendix 3.B, the source code of the programs in question is presented.

```
public class Alice {
  private int data_source = 0,data,data_sink;
  private Bob b;
  public void initiator(){ data = data_source; }
  public Alice(){ initiator(); b = new Bob(); }
  public static void main(String[] args){
    Alice a = new Alice();
    a.operation();
    a.finalizer();
  }
  private void operation(){
    b.receive(data);
    b.good();
    b.bad();
  }
  public void finalizer(){ data_sink = data;  b.finalizer(); }
}
```

```
public class Bob {
  private int data1_source=0,data1,data1_sink;
  private int data2_source=1,data2,data2_sink;
  private Charlie c;
  public void initiator(){
    data1 = data1_source;
    data2 = data2_source;
  }
  public Bob(){ initiator();  c = new Charlie(); }
  public void receive(int x){ data1 = x; }
  public void good(){ c.receive(data2); }
  public void bad(){ c.receive(data1); }
  public void finalizer(){
    data1_sink = data1;
    data2_sink = data2;
    c.finalizer();
  }
}
```

```
public class Charlie {
  private int data_source, data, data_sink;
  public void initiator(){data = data_source;}
  public Charlie(){ initiator(); }
  public void receive(int x){ data = x; }
  public void finalizer(){ data_sink = data; }
}
```

Figure 18: The canonical version of Alice-Bob-Charlie.

5.1 Alice-Bob-Charlie (the running example)

We start with the running example as the first case, introduced in Figure 1. To model the batch-job style, we modify the code to include instances of components as fields of Java classes. Following standard practices in object-oriented programming, our prototype leverages *composition* relationship [24] between classes where an object is a part of another object. This leads to a hierarchy of objects, in which each object is responsible for creation and deletion of required objects of other classes. Assuming that no local variable creates a new instance of a class, the execution starts from the main object and continues in the underlying ones.

Given the main method as the starting point of the program, constructors naturally provide placeholders for inserting the init section (*initiator* methods), while the last line of the main method is the placeholder for final assignments existing in *finalizer* methods. By calling the finalizer method of the main object, following the composition hierarchy, objects invoke the finalizers as a chain. In the end, all of the *sink* fields are assigned.

The transpiler suffices to inject the initiator and finalizer methods per class. For readability, we slightly modify the canonicalization algorithm. We add a *source* field assigned to the initial value of the field in the original program, instead of replacing occurrences of the variable with *temp* variables. As an example, the canonical version of the program is shown in Figure 18.

Considering the labels A, B, and C, for `Alice`, `Bob`, and `Charlie` and with respect to the permitted flow ($A \trianglerighteq B, B \trianglerighteq C$), the transpiler also generates the input script for JOANA. Figure 19 displays the important snippet of it.

The first line describes the power-lattice, where `e` denotes the empty set as the bottom element. It is followed by the list of annotations on field variables to distinguish $sources$ and $sinks$ of information per class. For

```
1  setLattice e<=A,e<=B,e<=C,A<=AB,A<=AC,B<=AB,
2  B<=BC,AB<=ABC,C<=AC,C<=BC,AC<=ABC,BC<=ABC
3  source Alice.data_source   A
4  sink   Alice.data_sink     A
5  source Bob.data1_source    B
6  sink   Bob.data1_sink      AB
7  source Bob.data2_source    B
8  sink   Bob.data2_sink      AB
9  source Charlie.data_source C
10 sink   Charlie.data_sink   BC
11 run    classical-ni
```

Figure 19: A snippet of JOANA script for Alice-Bob-Charlie.

example, the line `sink Charlie.data_sink BC` means `Charlie.data_sink` is a sink variable with the security level BC (the set of nontransitive labels can flow to C). The last command of the script triggers the flow-sensitive information flow analysis. As the result of the analysis, JOANA reports the security violation `Illegal flow from Alice.data_source to Charlie.data_sink, visible for BC`, which captures the undesired explicit flow.

Omitting invocation of the `bad` method yields a secure program. In this case, JOANA reports `No violations found` after running the same script on the canonical version of the secure program.

5.2 Confused deputy

We benefit from the fact that nontransitive information flow control supports enforcing both confidentiality and integrity policies. The confused deputy problem [12] occurs in a situation when an untrusted component is able to manipulate a trusted component and misuse its authority to execute a sensitive operation. It is an integrity problem since the policy states if the attacker is not permitted to alter a resource, then there must not be any way to do so, directly or by using a deputy. We adopt Lu and Zhang's code [17] as a starting point to represent the confused deputy problem.

Figure 20 illustrates the skeleton of the source code. We make use of four classes: `Library`, `Service`, `Downloaded_Code`, and `Trusted_Code`. Values in `Library` are protected and only `Service` is privileged to access them. The class `Downloaded_Code` is third-party code that cannot access to `Library`, while `Trusted_Code` is completely trusted. Invoking `addLog` method of `Service` is permitted because it updates a non-executable log file in `Service`, but the `process` method of `Library` must not be called with data from `Downloaded_Code` via `Service`. To rephrase the integrity policy, `Downloaded_Code` should not have any effects on the sensitive component `Library`, directly or indirectly, while `Trusted_Code` can. Given the initial letters of the component names as their labels, the specified policy is $D \trianglerighteq S$, $S \trianglerighteq L$, $T \trianglerighteq S$ and $T \trianglerighteq L$.

On the other hand, `Downloaded_Code` must not retrieve `Library`'s information through invoking the `query` method by `Service`. Taking confidentiality policies into account, we add flow relations $L \trianglerighteq S$, $S \trianglerighteq D$, $L \trianglerighteq T$, and $S \trianglerighteq T$ to exclude the illegal flows from `Library` to `Downloaded_Code` violating data secrecy. To sum up, the intended policy is the aggregation of the integrity and confidentiality policies, which are defined uniformly by the aforementioned nontransitive flows.

```
public class Library {
  private int someValue = 5, printValue = 0;
  ...
  public void process(int src){
    printValue = src;
  }
  public int retrieve(int key){
    return someValue;
  }
}
```

```
public class Service {
  private int logFile = 0;
  private Library library;
  ...
  public void addLog(int x, int y){
    logFile += x + y ;
  }
  public void print(int data){
    library.process(data);
  }
  public int query(int key){
    return library.retrieve(key);
  }
}
```

```
public class Downloaded_Code {
  private int data = 7, key = 4, result;
  private Service service;
  ...
  public static void main(String[] args){
    Downloaded_Code dc = new Downloaded_Code();
    dc.operation();
  }
  private void operation(){
    service.addLog(data, key);
    service.print(data);
    result = service.query(key);
  }
}
```

Figure 20: The skeleton of Confused deputy source code.

The transpiler generates the canonical version of the program and annotates sources and sinks of information in classes. JOANA discovers the violations in the program and reports the two existing illegal flows: `Illegal flow from Downloaded_Code.data_source to Library.printValue_sink, visible for LS` (integrity) and `Illegal flow from Library.someValue_source to Downloaded_Code.result_sink, visible for DS` (confidentiality).

A secure version of the program is the one without calling `service.print(data)` and `service.query(key)` in the `operation` method. Now information from `Downloaded_Code` (as $\{D\}$) influences only `logFile` in `Service` (as {D, L, S, T}), which is allowed by the policy. JOANA also confirms security of the program by running the same script on the canonical version of the revised program.

5.3 Bank logger

We discuss another example in which two bank services for processing customers' information (`Bank`) and logging their public information (`Logger`) are totally separated. A client component (`BankLog`) is developed to communicate with both services at the same time. Figure 21 focuses on the important parts of the source code. The two components `Bank` and `BankLog` can mutually access each other's information, although `Logger` may read insensitive information. Thus, `Logger` must not interfere with `Bank` directly or indirectly. We label `Bank`, `Logger`, and `BankLog` components as B, L, and C, respectively. Consequently, the intended policy is $C \trianglerighteq B$, $B \trianglerighteq C$, and $C \trianglerighteq L$.

The current implementation of the program violates the policy by two implicit flows. The `getBalance` method checks whether the `id` exists, and `BankLog` only requests for logging if the sensitive value `balance` is positive. Executing the JOANA script on the canonical version of the program generates the following report: `Illegal flow from Bank.id_source to Logger.logFile_sink, visible for CL` (flow #1) and `Illegal flow from Bank.balance_source to Logger.logFile_sink, visible for CL` (flow #2).

To secure the program, the log content must not be influenced by sensitive information. One possible way to repair the program is logging the number of accesses to the client component `BankLog`. Hence, we replace lines 7 and 8 of `BankLog` (in the `operation` method) with `l.append(1)`. With this change, JOANA accepts the canonical version of the program using the same script.

```
1  public class Bank {
2    private int id = 20;
3    ...
4    public int getBalance(int x){
5      if (x == id) return balance; //flow #1
6      return 0;
7    }
8  }
```

```
1  public class BankLog {
2    private int userId = 20, balance;
3    private Bank b; private Logger l;
4    ...
5    private void operation(){
6      balance = b.getBalance(userId);
7      if (balance > 0) //flow #2
8      l.append(userId);
9    }
10 }
```

Figure 21: An excerpt from Bank logger source code.

5.4 Low-High

The previous examples included more than two components, which allowed us to contrast transitive and nontransitive policies. The following example demonstrates the compatibility with the baseline case of the two-level security policy. The program (in Appendix 3.B) contains two components `Alice` and `Bob`, where `Alice` updates her data influenced by `Bob`'s secret value. We define the nontransitive policy $L \trianglerighteq H$ such that L is the label of `Alice` and H is for `Bob`.

The transpiler transforms the program and generates the input script for JOANA, as can be seen in Figure 22. Therefore, JOANA analyzes the program and reports message `Illegal flow from Bob.secret_source to`

```
1  setLattice e<=L,e<=H,L<=LH,H<=LH
2  source Alice.data_source L
3  sink   Alice.data_sink   L
4  source Bob.secret_source H
5  sink   Bob.secret_sink   LH
6  source Bob.data_source   H
7  sink   Bob.data_sink     LH
```

Figure 22: A snippet of JOANA script for Low-High.

`Alice.data_sink, visible for L` expresses the security violation caused by the implicit flow.

Removing the illegal flow (line 13 in `Alice`) makes the program secure, which is verified by running the JOANA script on the canonical version of the modified program.

6 Alternative policies and encodings

Fine-grained policies While the main motivation for nontransitive types is enforcing coarse-grained information-flow policies, where labels represent components, the notion of nontransitive security is not limited to module separation [17]. Other real-world scenarios such as policies in social media (e.g., "only my friends can see my photo but not friends of my friends") also naturally match nontransitive policies. Our framework can thus be generalized to decouple the flow-to relation from component labels, allowing fine-grained nontransitive policies.

Scalability The proposed transpiler employs the power-lattice encoding that expands the number of security levels exponentially. For the type system, however, its time and space complexity do not depend on the size of the lattice. The reason is that we never need to store the lattice, as the flow-to relation is implicitly derived from its elements. In an off-the-shelf deployment of JOANA, there is no time blowup, but we cannot avoid the space blowup because JOANA is lattice-agnostic. Making JOANA aware of the power-lattice nature of the lattice (e.g., in the style of DLM [19]) can help avoiding the blowup in the current implementation.

Alternative encodings A power-lattice encoding enables us to support declassification and dynamic policies. However, when such generality is not needed, we can reduce the size of the lattice by alternative encodings, with the cost of losing granularity of information stored in security labels.

We identify the soundness constraint for a nontransitive-to-transitive policy encoding as $\ell \trianglerighteq \ell' \iff \ell_{source} \sqsubseteq \ell'_{sink}$, where source and sink variables of a component are labeled as ℓ_{source} and ℓ_{sink}, respectively, when the component has label ℓ in the nontransitive setting (recall that $\trianglerighteq$ is reflexive). Note that the powerset lattice encoding indeed meets the condition because $\forall \ell, \ell' \in L_{\mathcal{N}}.\ell_{source} = \{\ell\} \land \ell_{sink} = C(\ell) \land \Big(\ell \trianglerighteq \ell' \iff \{\ell\} \subseteq C(\ell')\Big)$ (see Figure 6). Among various lattices satisfying the constraint, a minimal one is desirable, i.e., the one with the smallest set of labels.

We present a so-called *source-sink* lattice encoding that satisfies the soundness constraint and reduces the size of the lattice from exponential to

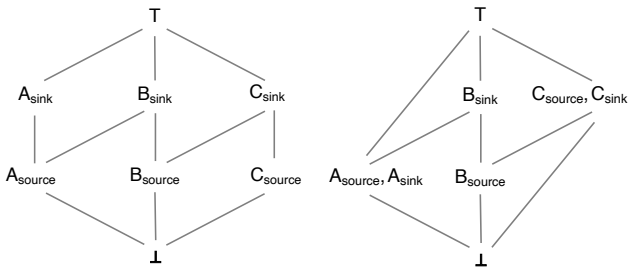


Figure 23: (a) A source-sink lattice encoding for the running example; (b) A minimal lattice.

polynomial. We start with a source-sink partial order where for all $\ell \in L_{\mathcal{N}}$, there are $\ell_{src}, \ell_{snk} \in L_{\mathcal{T}}$ such that $\ell_{src} \sqsubseteq \ell_{snk}$, due to reflexivity of the $\trianglerighteq$ relation. Then, according to the soundness constraint, we include transitive relations between levels based on the specified nontransitive flows. Since the security levels must constitute a lattice, we apply the Dedekind–MacNeille completion algorithm [4] to compute the smallest lattice containing the partial order. If a unique least upper (resp. greatest lower) bound for any pairs of source (resp. sink) levels does not exist, it adds an intermediary level between two source and two sink levels such that the intermediary level is the lub of the source levels and the glb of the sinks. It also makes one top and one bottom element for the lattice. Figure 23a illustrates the resulting source-sink lattice for the running example ($A \trianglerighteq B$ and $B \trianglerighteq C$).

In the worst case, the size of the lattice is $O(|L_{\mathcal{N}}|^2)$ and the time complexity of the algorithm is $O(|L_{\mathcal{N}}|^4)$, as proved in Appendix 3.A. Furthermore, optimization techniques can make the partial order compact, before constructing the lattice out of it; for example, any pairs of ℓ_{src} and ℓ_{snk} coincide in the partial order when one of them is only in relation with the other one, not any other levels. Figure 23b depicts the minimal source-sink lattice for the nontransitive policy in question; observe how A_{sink} and C_{source} are collapsed.

We demonstrate the NTNI-to-TNI tranpilation defined for a source-sink lattice, in comparison with the power-lattice encoding, by replacing $\{\ell\}$ with ℓ_{src} and $C(\ell)$ with ℓ_{snk} in the labeling function and program transformation. In Appendix 3.A, we formally introduce the transpilation using a source-sink lattice. We make use of the program canonicalization for batch-job programs and define the transitive encoding of a nontransitive policy based on a given source-sink lattice (Definition 15). We prove that any nontransitive policy on a program can be reduced to a corresponding transitive policy on a seman-

tically equivalent program (Theorem 9). For the enforcement mechanism, we prove that the presented flow-sensitive type system, while a source-sink lattice is in place, is sound and more permissive than the nontransitive type system (Theorems 10 and 11). Moreover, our results can be generalized to programs with intermediate inputs and outputs, where the program transformation algorithm replaces the level of input and output commands to ℓ_{src} and ℓ_{snk}, respectively (Algorithm 3 and Theorem 12). We also prove that the flow-sensitive type system for programs with I/O is compatible with a source-sink lattice (Theorem 13).

7 Related work

Our starting point is the special-purpose notions Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) by Lu and Zhang [17]. Our work demonstrates how to cast NTNI as classical noninterference on a lattice and how to improve the precision of NTT by classical flow-sensitive analysis.

Nontransitive noninterference is not to be confused by intransitive noninterference. Intransitive noninterference was introduced by Rushby [25] and explored by, amongst others, Roscoe and Goldsmith [23], Mantel and Sands [18], and Ron van der Meyden [30]. Intransitive noninterference is intended to address the *where* dimension of declassification [27]. The typical scenario for intransitive noninterference is ensuring that sensitive data is passed through a trusted encryption module before it is released. For example, security labels might be low, $encrypt$, and $high$, ordered by $high \rightarrow encrypt \rightarrow low$ while $high \nrightarrow low$. Like nontransitive policies, intransitive policies do not assume transitive policies. However, there is a fundamental difference between nontransitive and intransitive policies: intransitive noninterference allows low information to be (indirectly) dependent on $high$. In the encryption module scenario, this means that changes in the (high) plaintext may reflect in the changes in the (low) ciphertext. In contrast, nontransitive policy $A \trianglerighteq B$ and $B \trianglerighteq C$ guarantees that there are no information dependencies from A to C whatsoever.

Further approaches to declassification introduce decentralized hierarchies and dynamic policies. Myers and Liskov's DLM [19] is based on transitive policies that encode ownership in the labels. The goal is to allow declassification only if it is allowed by the owner(s) of the data. DC labels [28] by Stefan et al. models a setting of mutual distrust without relying on a centralized principal hierarchy. DC labels incorporate formulas over principals, modeling can-flow-to relation by logical implication. FLAM [2] by Arden et al. explores robust authorization to mitigate delegation loopholes in policies

like DLM. Jia and Zdancewic [15] encode security types using authorization logic in a programming language for access control. Their encoding does not assume transitivity and it needs to be encoded as explicit delegations. Swamy et al. [29] and Broberg et al. [6] explore the effects of dynamic policy updates on the transitivity of flows. Broberg et al. call a flow *time-transitive* if information leaks from A to C via B even if no flows from A to C are allowed at any given time. This can happen when the policy of allowing flows from A to B is dynamically updated to allow flows from B to C. Time-transitivity is not in the scope of our work because our policies are static.

Rajani and Garg [22] explore the granularity of policies for information flow control. They show that fine-grained type systems that track the propagation of values are as expressive as coarse-grained type systems that track the propagation of context. Vassena et al. [31] expand the study to the dynamic setting. Xiang and Chong [33] use opaque labeled values in their study of dynamic coarse-grained information flow control for Java-like languages. However, in both cases, the considered policies are transitive. An interesting avenue for future work is to explore whether these approaches can be integrated with ours to be able to handle nontransitive policies.

Our proof-of-concept implementation of the flow-sensitive analysis for Java draws on Hammer and Snelting's JOANA [10, 11]. Note that our reduction results are general, enabling the use of other practical flow-sensitive analyses like Pidgin [16] by Johnson et al. for tracking nontransitive policies.

8 Conclusion

In order to support module-level coarse-grained information-flow policies, Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) have been suggested recently as a new security condition and enforcement. In contrast to Denning's classical lattice model, NTNI and NTT assume no transitivity of the underlying flow relation. NTNI and NTT, in the form they were proposed, are nonstandard, requiring the development of nonstandard semantic machinery to reason about NTNI and the development of nonstandard enforcement techniques to track NTT.

This paper demonstrates that despite the different aims and intuitions of nontransitive policies compared to classical transitive policies, nontransitive noninterference can in fact be reduced to classical transitive noninterference.

On the security characterization side, we show that NTNI corresponds to classical noninterference on a lattice that records source-to-sink relations derived from nontransitive policies. On the enforcement side, we devise a lightweight program transformation that enables us to leverage standard

flow-sensitive information-flow analyses to enforce nontransitive policies. Further, we improve the permissiveness over the nonstandard NTT enforcement while retaining the soundness. We show that our security characterization and enforcement results naturally generalize to a language with intermediate input and outputs. An immediate practical benefit of our work is the implication that there is no need for dedicated design and implementation for the enforcement of nontransitive policies for practical programming languages. Instead, we can leverage state-of-the-art flow-sensitive information-flow tools, which we demonstrate by utilizing JOANA to enforce nontransitive policies for Java programs.

Acknowledgments Thanks are due to Yi Lu and Chenyi Zhang for inspiring this line of work and for the interesting discussions. This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and the Danish Council for Independent Research for the Natural Sciences (DFF/FNU, project 6108-00363).

Bibliography

[1] M. M. Ahmadpanah, A. Askarov, and A. Sabelfeld. Nontransitive Policies Transpiled - Supplementary Materials. `https://www.cse.chalmers.se/research/group/security/ntni`, 2021.

[2] O. Arden, J. Liu, and A. C. Myers. Flow-limited authorization. In *CSF*, 2015.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.

[4] K. Bertet, M. Morvan, and L. Nourine. Lazy completion of a partial order to the smallest lattice. In *Second Int. Symp. on Knowledge Retrieval, Use and Storage for Efficiency*, 1997.

[5] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *CCS*, 2009.

[6] N. Broberg, B. van Delft, and D. Sands. The anatomy and facets of dynamic policies. In *CSF*, 2015.

[7] S. Dahlgaard, M. B. T. Knudsen, and M. Stöckel. Finding even cycles faster via capped k-walks. In *STOC*, 2017.

[8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.

[9] B. Ganter and S. O. Kuznetsov. Stepwise construction of the dedekind-macneille completion (research note). In *ICCS*, volume 1453, 1998.

[10] 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.

[11] C. Hammer and G. Snelting. JOANA: Java Object-sensitive ANAlysis. `https://pp.ipd.kit.edu/projects/joana/`, 2020.

[12] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Oper. Syst. Rev.*, 22(4), 1988.

[13] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*. IOS Press, 2012.

[14] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL*, 2006.

[15] L. Jia and S. Zdancewic. Encoding information flow in Aura. In *PLAS*, 2009.

[16] A. Johnson, L. Waye, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. In *PLDI*, 2015.

[17] Y. Lu and C. Zhang. Nontransitive security types for coarse-grained information flow control. In *CSF*, 2020.

[18] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, 2004.

[19] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 2000.

[20] L. Nourine and O. Raynaud. A fast incremental algorithm for building lattices. *J. Exp. Theor. Artif. Intell.*, 14(2-3), 2002.

[21] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. SPOON: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exp.*, 46(9), 2016.

[22] V. Rajani and D. Garg. Types for information flow control: Labeling granularity and semantic models. In *CSF*, 2018.

[23] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *CSFW*, 1999.

[24] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[25] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory Menlo Park, 1992.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[27] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comp. Sec.*, 2009.

[28] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec*, 2011.

[29] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW*, 2006.

[30] R. van der Meyden. What, indeed, is intransitive noninterference? *J. Comput. Secur.*, 2015.

[31] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan. From fine- to coarse-grained dynamic information flow control and back. In *POPL*, 2019.

[32] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comp. Sec.*, 1996.

[33] J. Xiang and S. Chong. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *S&P*, 2021.

[34] R. Yuster and U. Zwick. Finding even cycles even faster. *SIAM J. Discret. Math.*, 10(2), 1997.

Appendix

3.A Source-sink encoding

We define the source-sink lattice encoding of a nontransitive policy to a transitive policy for canonical programs as follows.

Definition 15 (*Transitive Encoding of Nontransitive Policies*)**.** Given a nontransitive policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}} \rangle$ and a program c, a corresponding transitive policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$ on the canonical version of the program is $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} | \ell \in L_{\mathcal{N}}\} \cup \{\top, \bot\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell \trianglerighteq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ ($\trianglerighteq$ is reflexive) such that $\langle L_{\mathcal{T}}, \sqsubseteq \rangle$ constitutes a lattice, and

$$\forall x \in Var_c.\, \Gamma_{\mathcal{N}}(x) = \ell \implies \begin{cases} \Gamma_{\mathcal{T}}(x) & = & \ell_{src} \\ \Gamma_{\mathcal{T}}(x_{temp}) & = & \top \\ \Gamma_{\mathcal{T}}(x_{sink}) & = & \ell_{snk} \end{cases}.$$

As stated in Definition 15, the initial and final values of an ℓ-observable variable x of the given program are ℓ_{src}- and ℓ_{snk}-observable in the canonical version, respectively. Also, only the top-level observer can see final values of internal *temp* variables, thus makes them $\top$-observable. The next lemma demonstrates that for any canonical program satisfying a nontransitive policy, the program also complies with a corresponding transitive policy and vice versa.

Lemma 5 (*From $NTNI_{TI}$ to TNI_{TI} for Canonical Programs*)**.** *Any canonical program $Canonical(c)$ is secure with respect to a nontransitive security policy $\mathcal{N}$ where $\forall x \in Var_c. \Gamma_{\mathcal{N}}(x_{temp}) = \Gamma_{\mathcal{N}}(x_{sink}) = \Gamma_{\mathcal{N}}(x)$ if and only if the canonical program is secure according to a corresponding transitive security policy $\mathcal{T}$. We write $\forall c. \forall \mathcal{N}. \exists \mathcal{T}.\ NTNI_{TI}(\mathcal{N}, Canonical(c)) \iff TNI_{TI}(\mathcal{T}, Canonical(c))$.*

Therefore, we prove that any nontransitive policy on a given program can be modeled as a transitive policy on the canonical version of the program.

Expression Evaluation

$$\frac{}{\langle v, M\rangle \Downarrow v} \quad \text{(IO-Value)}$$

$$\frac{}{\langle x, M\rangle \Downarrow M(x)} \quad \text{(IO-Read)}$$

$$\frac{\langle e_1, M\rangle \Downarrow v_1 \qquad \langle e_2, M\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, M\rangle \Downarrow v_1 \oplus v_2} \quad \text{(IO-Operation)}$$

Command Evaluation

$$\frac{}{\langle \mathit{skip}, M, I, O\rangle \to \langle \mathit{stop}, M, I, O\rangle} \quad \text{(IO-Skip)}$$

$$\frac{\langle e, M\rangle \Downarrow v \qquad M' = M[x \mapsto v]}{\langle x := e, M, I, O\rangle \to \langle \mathit{stop}, M', I, O\rangle} \quad \text{(IO-Write)}$$

$$\frac{c = \mathit{if}\ e\ \mathit{then}\ c_{\mathit{true}}\ \mathit{else}\ c_{\mathit{false}} \qquad \langle e, M\rangle \Downarrow b}{\langle c, M, I, O\rangle \to \langle c_b, M, I, O\rangle} \quad \text{(IO-If)}$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{\mathit{body}} \qquad \langle e, M\rangle \Downarrow \mathit{true}}{\langle c, M, I, O\rangle \to \langle c_{\mathit{body}}; c, M, I, O\rangle} \quad \text{(IO-While-T)}$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{\mathit{body}} \qquad \langle e, M\rangle \Downarrow \mathit{false}}{\langle c, M, I, O\rangle \to \langle \mathit{stop}, M, I, O\rangle} \quad \text{(IO-While-F)}$$

$$\frac{\begin{array}{c} c = \mathit{input}(x, \ell) \qquad I(\ell) = v.\sigma \\ I' = I[\ell \mapsto \sigma] \qquad M' = M[x \mapsto v] \end{array}}{\langle c, M, I, O\rangle \to \langle \mathit{stop}, M', I', O\rangle} \quad \text{(IO-Input)}$$

$$\frac{\begin{array}{c} c = \mathit{output}(x, \ell) \\ M(x) = v \qquad O' = O.\, v_\ell \end{array}}{\langle c, M, I, O\rangle \to \langle \mathit{stop}, M, I, O'\rangle} \quad \text{(IO-Output)}$$

$$\frac{\langle c_1, M, I, O\rangle \to \langle c_1', M', I', O'\rangle}{\langle c_1; c_2, M, I, O\rangle \to \langle c_1'; c_2, M', I', O'\rangle} \quad \text{(IO-Seq-I)}$$

$$\frac{}{\langle \mathit{stop}; c, M, I, O\rangle \to \langle c, M, I, O\rangle} \quad \text{(IO-Seq-II)}$$

Figure 24: Language semantics with I/O.

Theorem 9 (*From $NTNI_{TI}$ to TNI_{TI}*). *For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$, there exist a semantically equivalent (modulo canonicaliztion) program c' and a transitive se-*

$$\frac{}{\Gamma \vdash v : \bot} \quad \text{(IO-TT-Value)}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \text{(IO-TT-Read)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \sqcup t_2} \quad \text{(IO-TT-Operation)}$$

$$\frac{}{pc \vdash \Gamma\{skip\}\Gamma} \quad \text{(IO-TT-Skip)}$$

$$\frac{\Gamma \vdash e : t}{pc \vdash \Gamma\{x := e\}\Gamma[x \mapsto pc \sqcup t]} \quad \text{(IO-TT-Write)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \\ pc \sqcup t \vdash \Gamma\{c_{true}\}\Gamma' \\ pc \sqcup t \vdash \Gamma\{c_{false}\}\Gamma'\end{array}}{pc \vdash \Gamma\{\mathit{if}\ e\ \mathit{then}\ c_{true}\ \mathit{else}\ c_{false}\}\Gamma'} \quad \text{(IO-TT-If)}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{c_{body}\}\Gamma}{pc \vdash \Gamma\{\mathit{while}\ e\ \mathit{do}\ c_{body}\}\Gamma} \quad \text{(IO-TT-While)}$$

$$\frac{pc \vdash \Gamma\{c_1\}\Gamma' \qquad pc \vdash \Gamma'\{c_2\}\Gamma''}{pc \vdash \Gamma\{c_1; c_2\}\Gamma''} \quad \text{(IO-TT-Seq)}$$

$$\frac{pc \sqsubseteq \ell}{pc \vdash \Gamma\{input(x,\ell)\}\Gamma[x \mapsto \ell]} \quad \text{(IO-TT-Input)}$$

$$\frac{pc \sqcup \Gamma(x) \sqsubseteq \ell}{pc \vdash \Gamma\{output(x,\ell)\}\Gamma} \quad \text{(IO-TT-Output)}$$

$$\frac{\begin{array}{c}pc_1 \vdash \Gamma_1\{c\}\Gamma_1' \\ pc_2 \sqsubseteq pc_1 \qquad \Gamma_2 \sqsubseteq \Gamma_1 \qquad \Gamma_1' \sqsubseteq \Gamma_2'\end{array}}{pc_2 \vdash \Gamma_2\{c\}\Gamma_2'} \quad \text{(IO-TT-Sub)}$$

Figure 25: Flow-sensitive typing rules with I/O.

curity policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$*, as specified in Definition 15, such that* $NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c')$*. Formally,*

$$\forall \mathcal{N}. \forall c. \exists \mathcal{T}. \exists c'. c \simeq_C c' \wedge NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c').$$

The next theorem states that the flow-sensitive type system is sound; in other words, if the type system accepts a canonical program, then the pro-

gram satisfies the transitive noninterference, and consequently, the original program complies with the nontransitive policy.

Theorem 10 (*Soundness of Flow-Sensitive Transitive Type System*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\{Canonical(c)\}\Gamma' \implies TNI_{TI}(\mathcal{T}, Canonical(c)).$$

The next theorem shows if a program is secure under the nontransitive type system, the flow-sensitive type system accepts the canonical version of the program as well.

Theorem 11 (*Flow-Sensitive Type System Covers Nontransitive Type System*)**.**

$$\mathcal{P}, \Gamma_1, pc \vdash c : t \implies pc \vdash \Gamma_2\{Canonical(c)\}\Gamma_3,$$

where $\forall x \in Var_c. \Gamma_3(x_{temp}) \sqsubseteq \bigsqcup_{\ell \in \Gamma_1(x)} \ell_{src} \wedge \mathcal{P}(x) = \ell \implies \Gamma_2(x) = \Gamma_3(x) = \ell_{src} \wedge \Gamma_2(x_{temp}) = \top \wedge \Gamma_2(x_{sink}) = \Gamma_3(x_{sink}) = \ell_{snk}$.

We also introduce the transpilation for programs with intermediate input/outputs. Similar to the batch-job style, we establish a source-sink lattice out of nontransitive labels, i.e., $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} | \ell \in L_{\mathcal{N}}\} \cup \{\top, \bot\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell \trianglerighteq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ ($\trianglerighteq$ is reflexive) such that $\langle L_{\mathcal{T}}, \sqsubseteq \rangle$ is a lattice. In the program transformation algorithm, only the levels of input and output commands are modified because the notion of progress-insensitive noninterference only focuses on the relation between program inputs and outputs.

Program transformation As explained in Algorithm 3, we label sources and sinks of information at a security level $\ell \in L_{\mathcal{N}}$ as ℓ_{src} and ℓ_{snk}, respectively. More precisely, we replace $input(x, \ell)$ commands with $input(x, \ell_{src})$, and also $output(x, \ell)$ commands with $output(x, \ell_{snk})$ in the program.

Algorithm 3: Transformation algorithm for programs with I/O.

Input : Program c
Output: Program $Transform(c)$
foreach $x \in Var_c$ **do**
  $c[input(x, \ell) \mapsto input(x, \ell_{src})]$
  $c[output(x, \ell) \mapsto output(x, \ell_{snk})]$
end
$Transform(c) := c$
return *Transform(c)*

Obviously, the transformed version of a given program preserves the meaning and termination behavior of the original program, yet it changes the channel of output values. The input and output values at the level ℓ can be found on the input channel with label ℓ_{src} and the output channel labeled as ℓ_{snk} in the canonical version of the given program. The next lemma shows the semantic relation between a given program and the transformed one.

Lemma 6 (*Semantic Equivalence Modulo Transformation*)**.** *For any program c, the semantic equivalence $\simeq_T$ between the programs c and $Transform(c)$ holds where $c \simeq_T c' \triangleq \forall M. \forall I. \exists I'. \Big(\forall \ell. I(\ell) = I'(\ell_{src})\Big) \wedge \langle c, M, I, \varnothing\rangle \rightsquigarrow O \wedge \langle c', M, I', \varnothing\rangle \rightsquigarrow O' \wedge O' = O\,[v_\ell \mapsto v_{\ell_{snk}}]$.*

Then, we prove a nontransitive policy on a given program (with intermediate inputs/outputs) can be reduced to a transitive policy on the transformed version of the program.

Theorem 12 (*From $NTNI_{PI}$ to TNI_{PI}*)**.** *For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$, there exist a semantically equivalent (modulo transformation) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ where $c' = Transform(c)$, $\langle L_{\mathcal{T}}, \sqsubseteq\rangle$ is a corresponding source-sink lattice and $\forall x \in Var_c. \ell = \Gamma_{\mathcal{N}}(x) \implies \Gamma_{\mathcal{T}}(x) = \ell_{src}$ such that $NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c')$. Formally,*

$$\forall \mathcal{N}. \forall c. \exists \mathcal{T}. \exists c'. c \simeq_T c' \wedge NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c').$$

Theorem 13 (*Soundness of Flow-Sensitive Type System for Programs with I/O*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\{Transform(c)\}\Gamma' \implies TNI_{PI}(\mathcal{T}, Transform(c)).$$

Proof of complexity of source-sink lattice encoding. We know that source levels are incomparable in the source-sink partial order, the same for sink levels. Thus, if there is not a quadruple of levels, two sources and two sinks, such that source levels are in relation with both of the sinks, then adding a top and a bottom element yields the smallest lattice. To do so, we detect cycles of length four in the undirected graph of the partial order. In the worst case, it takes $\binom{|L_{\mathcal{N}}|}{2}.O(|L_{\mathcal{N}}|^2) = O(|L_{\mathcal{N}}|^4)$ for the graph that has $2.|L_{\mathcal{N}}|$ nodes; $O(|L_{\mathcal{N}}|^2)$ for finding each cycle [7, 34], and $\binom{|L_{\mathcal{N}}|}{2}$ cycles exist at most. For each cycle, we add one intermediary level to the partial order, as the unique least upper (resp. greatest lower) bound of the source (resp. sink) levels. Hence, in the worst case, the resulting lattice adds $\frac{|L_{\mathcal{N}}|^2}{2} + 2$ more levels to the partial order, thus $O(|L_{\mathcal{N}}|^2)$ is the size of the lattice. It is also proven that the Dedekind-MacNeille completion takes $O(r^2)$ where r is the number of elements in the lattice [4, 9, 20], thus $O(|L_{\mathcal{N}}|^4)$. □

3.B Case studies

Alice-Bob-Charlie

```
public class Alice {
  private int data = 13;
  private Bob b;
  public Alice(){
    b = new Bob();
  }
  public static void main(String[] args){
    Alice a = new Alice();
    a.operation();
  }
  private void operation(){
    b.receive(data);
    b.good();
    b.bad();
  }
}
```

```
public class Bob {
  private int data1 = 0, data2 = 42;
  private Charlie c;
  public Bob(){
    c = new Charlie();
  }
  public void receive(int x){
    data1 = x;
  }
  public void good(){
    c.receive(data2);
  }
  public void bad(){
    c.receive(data1);
  }
}
```

```
public class Charlie {
  private int data;
  public Charlie(){ }
  public void receive(int x){
    data = x;
  }
}
```

Confused deputy

```
public class Library {
  private int someValue = 5;
  private int printValue = 0;
  public Library(){ }
  public void process(int src){
    printValue = src;
  }
  public int retrieve(int key){
    return someValue;
  }
}
```

```
public class Service {
  private int logFile = 0;
  private Library library;
  public Service(){
    library = new Library();
  }
  public void addLog(int x, int y){
    logFile += x + y ;
  }
  public void print(int data){
    library.process(data);
  }
  public int query(int key){
    return library.retrieve(key);
  }
}
```

```
public class Downloaded_Code {
  private int data = 7, key = 4, result;
  private Service service;
  public Downloaded_Code(){
    service = new Service();
  }
  public static void main(String[] args){
    Downloaded_Code dc = new Downloaded_Code();
    dc.operation();
  }
  private void operation(){
    service.addLog(data, key);
    service.print(data);
    result = service.query(key);
  }
}
```

Bank logger

```
public class Bank {
  private int id = 20, balance = 100;
  public Bank(){ }
  public int getBalance(int x){
    if (x == id)
    return balance;
    return 0;
  }
}
```

```
public class Logger {
  private static int logFile;
  public Logger(){ }
  public void append(int x){
    logFile += x;
  }
}
```

```
public class BankLog {
  private int userId = 20, balance;
  private Bank b;
  private Logger l;
  public BankLog(){
    b = new Bank();
    l = new Logger();
  }
  public static void main(String[] args){
    BankLog bl = new BankLog();
    bl.operation();
  }
  private void operation(){
    balance = b.getBalance(userId);
    if (balance > 0)
    l.append(userId);
  }
}
```

Low-High

```
public class Bob {
  private int secret = 100, data;
  public Bob(){ }
  public void receive(int x){
```

```
5       data = x;
6     }
7     public int getSecret(){
8       return secret;
9     }
10  }
```

```
1   public class Alice {
2     private int data = 10;
3     private Bob bob;
4     public Alice(){
5       bob = new Bob();
6     }
7     public static void main(String[] args){
8       Alice a = new Alice();
9       a.sendDataToBob();
10    }
11    public void sendDataToBob(){
12      bob.receive(data);
13      if (bob.getSecret() > data)
14      data++;
15    }
16  }
```

3.C Proofs

Proof of Theorem 1. It is straightforward because NTNI is a generalization of TNI where the policy defines all possible flows explicitly. Hence by considering the transitive and reflexive closure ($\sqsubseteq^*$) of the transitive relation ($\sqsubseteq$) as the nontransitive one, the theorem holds.

1. Let $L_{\mathcal{N}} = L_{\mathcal{T}}, \trianglerighteq = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$. Then, $C(\ell) = \{\ell' | \ell' \trianglerighteq \ell\} = \{\ell' | \ell' \sqsubseteq^* \ell\}$, and according to the definitions 1 and 3, $\forall \ell. \Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \iff M_1 \overset{\ell}{=}_{\mathcal{T}} M_2\Big)$.

2. Considering Definitions 3 and 4,
$\Big(\forall \ell \in L_{\mathcal{N}}. \forall M_1, M_2. \Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \wedge \langle c, M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle c, M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle\Big) \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'\Big) \iff$
$\Big(\forall \ell \in L_{\mathcal{N}}. \forall M_1, M_2. \Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \wedge \langle c, M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge$

$\langle c, M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle\Big) \implies M_1' \overset{C(\ell)}{=}_{\mathcal{N}} M_2'\Big)$, thus the theorem holds. □

Proof of Lemma 1. The transformed program c' is partitioned into three sections such that $c' = init_{c'}; orig_{c'}; final_{c'}$: (1) initial assignments for *temp* variables ($init_{c'}$), (2) the original program that variables are renamed to *temp* variables ($orig_{c'}$), and (3) final assignments for *sink* variables ($final_{c'}$).

1. The init section only sets the values of x_{temp} variables and each assignment is in the form of $x_{temp} := x$ for all $x \in Var_c$. We also know that $\forall x \in Var_c. x_{sink} \notin FV(init_{c'})$. Using the rule (Write) of the semantics by the number of elements in Var_c, we can conclude that the init section always terminates and $\forall M. \exists! M'. \langle init_{c'}, M\rangle \rightarrow^{|Var_c|} \langle \text{stop}, M'\rangle \wedge \forall x \in Var_c. M'(x_{temp}) = M'(x) = M(x) \wedge M'(x_{sink}) = M(x_{sink})$.

2. The program c and the $orig_{c'}$ section are identical up to α-renaming of variables $x \in Var_c$ with x_{temp}, and $\forall x \in Var_c. x \notin FV(orig_{c'}) \wedge x_{sink} \notin FV(orig_{c'})$. Thus, we write $\forall M_1, M_2. \forall x \in Var_c. M_1(x) = M_2(x) = M_2(x_{temp}) \implies \forall n \in \mathbb{N}. \langle c, M_1\rangle \rightarrow^n \langle c_1, M_1'\rangle \wedge \langle orig_{c'}, M_2\rangle \rightarrow^n \langle c_2, M_2'\rangle \wedge M_1'(x) = M_2'(x_{temp}) \wedge M_2'(x) = M_2(x) = M_1(x) \wedge M_2(x_{sink}) = M_2'(x_{sink})$.

3. The final section includes assignments from the value of x_{temp} variables to x_{sink} variables where assignments are in the form of $x_{sink} := x_{temp}$ for all $x \in Var_c$. We also know that $\forall x \in Var_c. x \notin FV(final_{c'})$. Similar to the init section, by applying the rule (Write) by the number of elements in Var_c, we can write $\forall M. \exists! M'. \langle final_{c'}, M\rangle \rightarrow^{|Var_c|} \langle \text{stop}, M'\rangle \wedge \forall x \in Var_c. M'(x_{sink}) = M'(x_{temp}) = M(x_{temp}) \wedge M'(x) = M(x)$.

4. If we use the semantic rule (Seq-I) for the sequence of these three sections and follow the aforementioned statements, we can conclude that Lemma 1 holds. □

Proof of Lemma 2. Using Lemma 1, we can establish a correspondence between the two security definitions. We have $\langle c, M\rangle \rightarrow^* \langle \text{stop}, M'\rangle \iff \langle Canonical(c), M\rangle \rightarrow^* \langle \text{stop}, M''\rangle$, which means the termination behavior stays the same. Then given that $\forall x \in Var_c. M'(x) = M''(x_{temp}) = M''(x_{sink}) \wedge M(x) = M''(x)$, the lemma is proven. □

Proof of Lemma 3. For simplicity, we write $c' = Canonical(c)$. We know that $\forall x. \Big(P(x) \iff Q(x)\Big) \implies \Big(\forall x. P(x) \iff \forall x. Q(x)\Big)$. So to prove the

lemma, we show the correctness of the following statement:
$\forall M_1, M_2.\langle c', M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle c', M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle \implies \Big(\forall \ell \in L_{\mathcal{N}}.\Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'\Big) \iff \forall \ell' \in L_{\mathcal{T}}.\Big(M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2 \implies M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'\Big)\Big)$.
If the execution of the program c' for (at least) one of the two arbitrary memories M_1 and M_2 does not terminate, then the premise in both security definitions does not hold, thus the lemma holds. Assuming the program is terminating for both memories, we prove the statement as follows:

1. Left to right:

   (a) Let $I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} | M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2\}$ be the set of levels in $L_{\mathcal{N}}$ that the two memories are indistinguishable for the set of labels can flow to them. Then, we have $I_{\mathcal{N}} \in L_{\mathcal{T}} \wedge I_{\mathcal{T}} = \{\ell' \in L_{\mathcal{T}} | M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2\} = \{\ell' \in L_{\mathcal{T}} | \ell \in I_{\mathcal{N}} \wedge \ell' \in \wp(C(\ell))\} = \wp(I_{\mathcal{N}})$ based on Definition 1.

   (b) Using Lemma 1, we can conclude that $\forall \ell \in I_{\mathcal{N}}.\forall x \in Var_c.\Gamma_{\mathcal{N}}(x) = \ell \implies \Big(\exists x_{sink} \in Var_{c'}.\Gamma_{\mathcal{T}}(x_{sink}) = C(\ell) \wedge M_1'(x_{sink}) = M_2'(x_{sink})\Big) \wedge \Big(\exists x \in Var_{c'}.\Gamma_{\mathcal{T}}(x) = \{\ell\} \wedge M_1'(x) = M_2'(x)\Big) \wedge \Big(\exists x_{temp} \in Var_{c'}.\Gamma_{\mathcal{T}}(x_{temp}) = L_{\mathcal{N}} \wedge M_1'(x_{temp}) = M_2'(x_{temp})\Big) \wedge \ell \in I_{\mathcal{T}} \wedge C(\ell) \in I_{\mathcal{T}}$.

   (c) Therefore, $\forall \ell \in I_{\mathcal{N}}.M_1' \overset{\ell}{=}_{\mathcal{N}} M_2' \iff \forall \ell' \in I_{\mathcal{T}}.M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'$. Hence, $\forall \ell \in L_{\mathcal{N}}.\Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'\Big) \implies \forall \ell' \in L_{\mathcal{T}}.\Big(M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2 \implies M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'\Big)$.

2. Right to left:

   (a) Let $I_{\mathcal{T}} = \{\ell' \in L_{\mathcal{T}} | M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2\}$ and $I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} | M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2\} = \{\ell \in L_{\mathcal{N}} | C(\ell) \in I_{\mathcal{T}}\}$.

   (b) According to Lemma 1, we have $\forall \ell' \in I_{\mathcal{T}}.\exists \ell \in L_{\mathcal{N}}.\Big(\ell' = \{\ell\} \implies \wp(C(\ell)) \subseteq I_{\mathcal{T}} \wedge \forall x \in Var_c.\Gamma_{\mathcal{N}}(x) = \ell \implies \Big(\exists x_{sink} \in Var_{c'}.\Gamma_{\mathcal{T}}(x_{sink}) = C(\ell) \wedge M_1'(x_{sink}) = M_2'(x_{sink})\Big) \wedge \Big(\exists x \in Var_{c'}.\Gamma_{\mathcal{T}}(x) = \ell' \wedge M_1'(x) = M_2'(x)\Big) \wedge \Big(\exists x_{temp} \in Var_{c'}.\Gamma_{\mathcal{T}}(x_{temp}) = L_{\mathcal{N}} \wedge M_1'(x_{temp}) = M_2'(x_{temp})\Big)\Big)$.

(c) Thus, $\forall \ell' \in I_{\mathcal{T}}.M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2' \iff \forall \ell \in I_{\mathcal{N}}.M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'$. Hence, $\forall \ell' \in L_{\mathcal{T}}.\Big(M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2 \implies M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'\Big) \implies \forall \ell \in L_{\mathcal{N}}.\Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'\Big)$. □

Proof of Theorem 2. By using Lemma 2 and Lemma 3. □

Proof of Theorem 3. To show soundness of the type system, we prove the following statement: $pc \vdash \Gamma\{c'\}\Gamma' \implies \Big(\forall \ell \in L_{\mathcal{T}}.\forall M_1, M_2.\Big(M_1 \overset{\ell}{=}_{\Gamma,\mathcal{T}} M_2 \wedge \langle c', M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle c', M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle\Big) \implies M_1' \overset{\ell}{=}_{\Gamma',\mathcal{T}} M_2'\Big) \wedge \forall x \in Var_{sink}.\Gamma'(x) = \Gamma(x)$, where $c' = Canonical(c)$ and $M_1 \overset{\ell}{=}_{\Gamma,\mathcal{T}} M_2 \iff \forall x \in Var_{c'}.\Gamma(x) \sqsubseteq \ell \implies M_1(x) = M_2(x)$. The first part of the statement denotes the definition of security in the flow-sensitive style and the second part of the statement ensures the flow-insensitivity of *sink* variables.

The first three rules determine the security level of expression e, which is the join of security levels associated with free variables of the expression.

By induction on the typing derivation and the structure of c', we have $\forall M.\forall x \in Var_{c'}.\Big(\langle c', M\rangle \rightarrow^* \langle \text{stop}, M'\rangle \wedge pc \vdash \Gamma\{c'\}\Gamma' \wedge pc \not\sqsubseteq \Gamma'(x)\Big) \implies M(x) = M'(x)$, where $pc \not\sqsubseteq \Gamma'(x)$ implies that no assignment to x occurs in c'. Note that in the assignment to sink variables (rule TT-Write-II), the memory gets updated in a secure way since $pc \sqcup \Gamma(x') \sqsubseteq \Gamma(x) \implies pc \sqsubseteq \Gamma(x)$.

It can also be easily proven by induction on the typing derivation that $pc \vdash \Gamma\{c'\}\Gamma' \wedge pc' \sqsubseteq pc \implies pc' \vdash \Gamma\{c'\}\Gamma'$.

By induction on the typing derivation and the structure of c', we show that $pc \vdash \Gamma\{c'\}\Gamma' \implies \forall \ell \in L_{\mathcal{T}}.\forall M_1, M_2.\Big(M_1 \overset{\ell}{=}_{\Gamma,\mathcal{T}} M_2 \wedge \langle c', M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle c', M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle\Big) \implies M_1' \overset{\ell}{=}_{\Gamma',\mathcal{T}} M_2'$. We discuss the cases as follows:

- Case (TT-Skip): We directly can write $pc \vdash \Gamma\{skip\}\Gamma \implies \forall \ell \in L_{\mathcal{T}}.\forall M_1, M_2.\Big(M_1 \overset{\ell}{=}_{\Gamma,\mathcal{T}} M_2 \wedge \langle skip, M_1\rangle \rightarrow^* \langle \text{stop}, M_1\rangle \wedge \langle skip, M_2\rangle \rightarrow^* \langle \text{stop}, M_2\rangle\Big) \implies M_1 \overset{\ell}{=}_{\Gamma,\mathcal{T}} M_2$.
- Case (TT-Write-I): The conclusion part is $pc \vdash \Gamma\{x := e\}\Gamma[x \mapsto pc \sqcup t]$, thus Γ and Γ' only might differ in x; and similarly for M_1 and M_2. The statement holds for this case because $pc \sqsubseteq \Gamma'(x) = pc \sqcup t$.
- Case (TT-Write-II): The condition $pc \sqcup \Gamma(x') \sqsubseteq \Gamma(x)$ checks if the assignment is permitted with regard to the transitive policy; it captures implicit (pc) and explicit ($\Gamma(x')$) flows to the variable x. Thus, we have $pc \vdash \Gamma\{x := x'\}\Gamma \implies \forall \ell \in L_{\mathcal{T}}.\forall M_1, M_2.\Big(M_1 \overset{\ell}{=}_{\Gamma,\mathcal{T}} M_2 \wedge$

$\langle x := x', M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle x := x', M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle\big) \implies M_1' \overset{\ell}{=}_{\Gamma,\mathcal{T}} M_2'$.

- Case (TT-If): Based on the induction hypothesis, $pc \sqcup t \vdash \Gamma\{c_b\}\Gamma' \implies TNI_{TI}(\mathcal{T}, c_b)$ for $b = true, false$. Since $pc \sqsubseteq pc \sqcup t$, the statement holds for this case.
- Case (TT-While): Based on the induction hypothesis, we have $pc \sqcup t \vdash \Gamma\{c_{body}\}\Gamma \implies TNI_{TI}(\mathcal{T}, c_{body})$, and $pc \sqsubseteq pc \sqcup t$, thus $pc \vdash \Gamma\{c\}\Gamma \implies TNI_{TI}(\mathcal{T}, c)$ for $c = while\ e\ do\ c_{body}$.
- Case (TT-Seq): Using the induction hypothesis, we have $pc \vdash \Gamma\{c_1\}\Gamma' \implies TNI_{TI}(\mathcal{T}, c_1) \wedge pc \vdash \Gamma'\{c_2\}\Gamma'' \implies TNI_{TI}(\mathcal{T}, c_2)$. Therefore, $pc \vdash \Gamma\{c_1; c_2\}\Gamma'' \implies TNI_{TI}(\mathcal{T}, c_1; c_2)$.
- Case (TT-Sub): Based on the induction hypothesis, $pc_1 \vdash \Gamma_1\{c\}\Gamma_1' \implies TNI_{TI}(\mathcal{T}, c)$. Considering the conditions $pc_2 \sqsubseteq pc_1 \wedge \Gamma_2 \sqsubseteq \Gamma_1 \wedge \Gamma_1' \sqsubseteq \Gamma_2'$, we can conclude $pc_2 \vdash \Gamma_2\{c\}\Gamma_2' \implies TNI_{TI}(\mathcal{T}, c)$.

We also prove the second part which requires the levels of *sink* variables remain unmodified through the program. There is no typing rule that updates the level of *sink* variables of the program, and the subsumption rule (rule TT-Sub) obviously guarantees the property. Therefore, by induction on the typing derivation, we have $\forall x \in Var_{sink}. \Gamma'(x) = \Gamma(x)$. □

Proof of Theorem 4. By induction on the derivation of expressions, we prove the type for expression e is the union of the security levels (i.e., the collected information flows) of free variables of the expression, formally $\Gamma \vdash e : t \implies t = \bigcup_{x \in FV(e)} \Gamma(x)$:

- Case (Value): We label values as empty set since they are visible for all levels and no free variable exists.
- Case (NT-Read): The type of variable x (i.e., $\Gamma(x)$) is the set of labels that might affect the value of the variable x in the program. It must capture all the possible flows to the variable, including the label of itself.
- Case (NT-Operation): Based on the induction hypothesis, it is easy to conclude that $t_1 \cup t_2 = \bigcup_{x \in FV(e_1 \oplus e_2)} \Gamma(x)$.
- Case (NT-Sub-I): The subtyping rule for expressions shows adding more security labels to the type of e keeps the expression well-typed.

By induction on the typing derivation and the structure of c, we prove the theorem as follows:

- Case (NT-Skip): It is easy to see that $\mathcal{P}, \Gamma, pc \vdash skip : t \implies NTNI_{TI}(\mathcal{N}, skip)$ for any $\mathcal{N}$.
- Case (NT-Write): This rule checks the explicit and implicit flows to the variable x have been collected in $\Gamma(x)$ and permitted by $\trianglerighteq$ rela-

tion. The type t is the union set of $\Gamma(x)$ (all collected information flows) and the type of e (the explicit flows). Considering pc (implicit flows) of the assignment, the premise investigates the presence of all labels in $t \cup pc$ in the collected flows to the variable x ($\Gamma(x)$), and guarantees that those are permitted according to the nontransitive flow. Hence, $\forall M_1, M_2. M_1 \overset{C(t \cup pc)}{=}_{\mathcal{N}} M_2 \wedge \langle x := e, M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle x := e, M_2\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \implies M_1' \overset{t \cup pc}{=}_{\mathcal{N}} M_2'$. Thus, $NTNI_{TI}(\mathcal{N}, x := e)$ holds.

- Case (NT-If): Based on the subtyping rule, we write $\mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{true} : t_2 \implies \mathcal{P}, \Gamma, pc \vdash c_{true} : t_2$, and similarly for c_{false}. Aggregating the labels in t_1 and t_2 and using the induction hypothesis prove the theorem statement for *if* commands.
- Case (NT-While) Similar to the previous case, if c_{body} is well-typed under $pc \cup t_1$, according to the induction hypothesis, this case is also proved.
- Case (NT-Seq): Using the induction hypothesis, $c_1; c_2$ has type $t_1 \cup t_2$ and $NTNI(\mathcal{N}, c_1; c_2)$ holds.
- Case (NT-Sub-II): The induction hypothesis shows $NTNI(\mathcal{N}, c)$ holds if c is well-typed, for example, has type t_1 under pc_1. If we extend the type with more security labels under a smaller pc, the command c remains well-typed and satisfies $NTNI(\mathcal{N}, c)$. □

Proof of Theorem 5. First, we start with demonstrating that $\mathcal{P}, \Gamma_1, pc \vdash c : t \implies \mathcal{P}', \Gamma_1', pc \vdash c' : t$, where $c' = Canonical(c)$ and we extend the typing context Γ_1 to Γ_1' and the labeling function $\mathcal{P}$ to $\mathcal{P}'$ by adding temp and sink variables with the same mappings for any variable x of the program, i.e., $\forall x \in Var_c. \mathcal{P}'(x) = \mathcal{P}'(x_{temp}) = \mathcal{P}'(x_{sink}) = \mathcal{P}(x) \wedge \Gamma_1'(x) = \Gamma_1'(x_{temp}) = \Gamma_1'(x_{sink}) = \Gamma_1(x)$.

As discussed in Lemma 1, the program is partitioned in three parts: $c' = init_{c'}; orig_{c'}; final_{c'}$. By induction on the derivation of $init_{c'}$ and using the two rules (NT-Write) and (NT-Seq), we have $\mathcal{P}', \Gamma_1', pc \vdash init_{c'} : t$ because statements are assignments of the form $x_{temp} := x$ and $\Gamma_1'(x) = \Gamma_1'(x_{temp})$. Also, since $\mathcal{P}, \Gamma_1, pc \vdash c : t$ holds, then $\forall \ell \in \Gamma_1(x) \trianglerighteq \mathcal{P}(x)$, and thus $\forall \ell \in \Gamma_1'(x_{temp}). \ell \trianglerighteq \mathcal{P}'(x_{temp})$.

We know that c and $orig(c')$ are identical up to α-renaming of variables $x \in Var_c$ with x_{temp}. Therefore, $\mathcal{P}, \Gamma_1, pc \vdash c : t \implies \mathcal{P}', \Gamma_1', pc \vdash c' : t$ because $\Gamma_1(x) = \Gamma_1'(x_{temp})$, $\mathcal{P}(x) = \mathcal{P}(x_{temp})$, and $x, x_{sink} \notin FV(c')$.

At the final section, statements are the form of $x_{sink} := x_{temp}$. Similar to the init section, because $\Gamma_1'(x_{temp}) = \Gamma_1'(x_{sink})$ and $\forall \ell \in \Gamma_1'(x_{sink}). \ell \trianglerighteq \mathcal{P}'(x_{sink})$, we can write $\mathcal{P}', \Gamma_1', pc \vdash final_{c'} : t$. Applying the rule (NT-Seq) two times, we

conclude $\mathcal{P}', \Gamma_1', pc \vdash init_{c'}; orig_{c'}; final_{c'} : t$.

Then, we prove $\mathcal{P}', \Gamma_1', pc \vdash c' : t \implies pc \vdash \Gamma_2\{c'\}\Gamma_3$ where $c' = Canonical(c)$. Remember that in the transitive type system $L_{\mathcal{T}} = \wp(L_{\mathcal{N}})$, $\sqsubseteq = \subseteq$, and $\sqcup = \cup$. To connect the typing contexts together meaningfully, the following constraints must be considered $\forall x \in Var_c$:

- $\Gamma_3(x_{temp}) \sqsubseteq \Gamma_1'(x_{temp})$: The final type of x_{temp} contains the set of labels in the last assignment that flow to the variable in the program c', due to flow-sensitivity of the transitive type system, while $\Gamma_1'(x_{temp})$ is the predicted set of *all* information flows to the variable x_{temp}.
- $\Gamma_2(x) = \{\mathcal{P}(x)\}, \Gamma_2(x_{temp}) = L_{\mathcal{N}}, \Gamma_2(x_{sink}) = C(\mathcal{P}(x))$: The conditions are based on the labeling function presented in Definition 5 to adjust the nontransitive mapping to the transitive one.
- $\Gamma_3(x) = \Gamma_2(x), \Gamma_3(x_{sink}) = \Gamma_2(x_{sink})$: As shown in Figure 9, if the program is well-typed, the types for variables remain untouched except for Var_{temp}.

There is a one-to-one correspondence between typing rules for expressions, which yields the union set of $\Gamma(x)$ for free variables $FV(e)$ as the type of the expression e. Thus, $\Gamma_1' \vdash e : t \implies \Gamma_2 \vdash e : t'$.

By induction on the nontransitive typing derivation $\mathcal{P}', \Gamma_1', pc \vdash c' : t$ and the structure of c':

- Case (NT-Skip): Based on the rule (TT-Skip), $pc \vdash \Gamma_2\{c'\}\Gamma_2$ holds.
- Case (NT-Write): We separate this case for two subcases according to the variable on the left side of the assignment:
  - If $x \in Var_{temp}$, since $\Gamma_1' \vdash e : t \implies \Gamma_2 \vdash e : t'$, based on the rule (TT-Write-I), we write $pc \vdash \Gamma_2\{c'\}\Gamma_2[x \mapsto pc \sqcup t']$.
  - If $x \in Var_{sink}$, we know that $e = x_{temp}$ is the only case in program c' at the $final_{c'}$ section. Because $\Gamma_3(x_{temp}) \subseteq \Gamma_1'(x_{temp})$ and $\forall \ell \in \Gamma_1'(x_{temp}) \cup pc.\, \ell \in \Gamma_1'(x_{sink}) \wedge \ell \trianglerighteq \mathcal{P}'(x_{sink}) \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq C(\mathcal{P}'(x_{sink})) \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq \Gamma_3(x_{sink})$. Hence, based on the rule (TT-Write-II), $pc \vdash \Gamma_3\{x := e\}\Gamma_3$.
- Case (NT-If): Using the induction hypothesis and $\Gamma_1' \vdash e : t \implies \Gamma_2 \vdash e : t'$, the statement $pc \vdash \Gamma_2\{c'\}\Gamma_3$ holds for this case with respect to the rule (TT-If).
- Case (NT-While): Similar to the case (NT-If), and according to the rule (TT-While).
- Case (NT-Seq): Using the induction hypothesis, $pc \vdash \Gamma_2\{c_1\}\Gamma_3$ and $pc \vdash \Gamma_3\{c_2\}\Gamma_4$, then $pc \vdash \Gamma_2\{c_1; c_2\}\Gamma_4$ by using the rule (TT-Seq).
- Case (NT-Sub-II): Using the induction hypothesis, we write $pc_1 \vdash \Gamma_2\{c\}\Gamma_2'$. Since $pc_2 \sqsubseteq pc_1$, $\Gamma_3 \sqsubseteq \Gamma_2$, $\Gamma_2' \sqsubseteq \Gamma_3'$ and in combination with the rule (NT-Sub-I), $pc_2 \vdash \Gamma_3\{c\}\Gamma_3'$ holds. □

Proof of Theorem 6. Simliar to the proof of Theorem 1, by considering the transitive and reflexive closure ($\sqsubseteq^*$) of the transitive relation ($\sqsubseteq$) as the nontransitive one, the theorem holds.

1. Let $L_{\mathcal{N}} = L_{\mathcal{T}}, \trianglerighteq = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$. Then, $C(\ell) = \{\ell' | \ell' \trianglerighteq \ell\} = \{\ell' | \ell' \sqsubseteq^* \ell\}$, and according to the definitions 8 and 12, $\forall \ell. \left(I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \iff I_1 \overset{\ell}{=}_{\mathcal{T}} I_2\right)$, and based on the defintions 7 and 11, $\forall \ell. \left(O_1 \overset{C(\ell)}{=}_{\mathcal{N}} O_2 \iff O_1 \overset{\ell}{=}_{\mathcal{T}} O_2\right)$.

2. Considering Definitions 11 and 14,
$\Big(\forall \ell \in L_{\mathcal{N}}. \forall M. \forall I_1, I_2. \left(I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \ \wedge \ \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1\right) \implies$
$\exists O_2. \langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \quad \wedge \quad O_1 \overset{\ell}{=}_{\mathcal{N}} O_2\Big) \iff$
$\Big(\forall \ell \in L_{\mathcal{N}}. \forall M. \forall I_1, I_2. \left(I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \ \wedge \ \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1\right) \implies$
$\exists O_2. \langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \wedge O_1 \overset{C(\ell)}{=}_{\mathcal{N}} O_2\Big)$, thus the theorem holds. □

Proof of Lemma 4. It is clear that the transformation only modifies the labels in the input and output commands of the given program, thus the behavior of the rest of the program stays unaffected. The changes in the labels of the input commands can be formulated as $\forall \ell. I(\ell) = I'(\{\ell\})$, where I is the input for the program c and I' is the input for the program c'.

By induction on the semantic rules shown in Figure 24, it is proven that c' progresses the same as c with the difference that outputs are sent to the channel $C(\ell)$ in lieu of ℓ. Therefore, we formulate it for the two outputs O and O' of programs c and c' respectively as $O' = O\,[v_\ell \mapsto v_{C(\ell)}]$, which means the only difference between the output sequences O and O' are the labels of output values; ones with the label ℓ in O are recorded at the same index in O' with the label $C(\ell)$. □

Proof of Theorem 7. Let $c' = \mathit{Transform}(c)$, $L_{\mathcal{T}} = \wp(L_{\mathcal{N}})$, $\sqsubseteq = \subseteq$ and $\forall x \in \mathit{Var}_c. \Gamma_{\mathcal{T}}(x) = \{\Gamma_{\mathcal{N}}(x)\}$.

1. We have $\forall \ell \in L_{\mathcal{N}}. \forall I_1, I_2. I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \iff \forall \ell' \in L_{\mathcal{T}}. \forall I_1', I_2'. I_1' \overset{\ell'}{=}_{\mathcal{T}} I_2'$ because of Definitions 8 and 12. Based on Lemma 4, we also know $\forall \ell \in L_{\mathcal{N}}. I(\ell) = I'(\{\ell\})$.

2. According to Definitions 10 and 14, and the semantic relation presented in Lemma 4, the statement $\forall M. \Big(\forall \ell \in L_{\mathcal{N}}. \forall I_1, I_2. \left(I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \ \wedge\right.$

$\langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1\big) \implies \exists O_2.\langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \wedge O_1 \overset{\ell}{=}_{\mathcal{N}} O_2\Big) \iff$
$\Big(\forall \ell' \in L_{\mathcal{T}}.\forall I_1', I_2'.\big(I_1' \overset{\ell'}{=}_{\mathcal{T}} I_2' \quad \wedge \quad \langle c', M, I_1', \varnothing\rangle \rightsquigarrow O_1'\big) \implies$
$\exists O_2'.\langle c', M, I_2', \varnothing\rangle \rightsquigarrow O_2' \wedge O_1' \overset{\ell'}{=}_{\mathcal{T}} O_2'\Big)$ holds if $O_1 \overset{\ell}{=}_{\mathcal{N}} O_2 \iff$
$O_1' \overset{\ell'}{=}_{\mathcal{T}} O_2'$.

3. As stated in Lemma 4, we conclude $O_1' = O_1[v_\ell \mapsto v_{C(\ell)}] \wedge O_2' = O_2[v_\ell \mapsto v_{C(\ell)}]$. Hence, $O_1 \overset{\ell}{=}_{\mathcal{N}} O_2 \iff O_1' \overset{\ell'}{=}_{\mathcal{T}} O_2'$ and consequently, the theorem holds. □

Proof of Theorem 8. We prove the following statement by induction on the typing derivation and the structure of $c' = \mathit{Transform}(c)$: $pc \vdash \Gamma\{c'\}\Gamma' \implies \forall \ell \in L_{\mathcal{T}}.\forall M.\forall I_1, I_2. I_1 \overset{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1 \implies \exists O_2.\langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \wedge O_1 \overset{\ell}{=}_{\mathcal{T}} O_2$.

The first three rules calculate the security level for expression e, by joining the security levels of its free variables.

The commands that update the security level of a variable are assignment (rules IO-TT-Write) and input (rule IO-TT-Input). Therefore, by induction on the typing derivation and the structure of c', we can write $\forall I.\forall M.\forall x \in \mathit{Var}_{c'}.\Big(\langle c', M, I, \varnothing\rangle \rightarrow^* \langle c'', M', I', O\rangle \wedge pc \vdash \Gamma\{c'\}\Gamma' \wedge pc \not\sqsubseteq \Gamma'(x)\Big) \implies M(x) = M'(x)$, where $pc \not\sqsubseteq \Gamma'(x)$ implies that no input or assignment to x occurs in c'. Note that for input commands (rule TT-Write-II), the memory gets updated in a secure way since $pc \sqsubseteq \Gamma'(x)$.

It can be easily proven by induction on the typing derivation that $pc \vdash \Gamma\{c\}\Gamma' \wedge pc' \sqsubseteq pc \implies pc' \vdash \Gamma\{c\}\Gamma'$.

Next, we investigate each case as follows:

- Case (IO-TT-Skip): It is easy to see that $pc \vdash \Gamma\{\mathit{skip}\}\Gamma \implies \forall \ell \in L_{\mathcal{T}}.\forall M.\forall I_1, I_2. I_1 \overset{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle \mathit{skip}, M, I_1, O\rangle \rightsquigarrow O \implies \langle \mathit{skip}, M, I_2, O\rangle \rightsquigarrow O \wedge O \overset{\ell}{=}_{\mathcal{T}} O$.
- Case (IO-TT-Write): For this case, we can write $pc \vdash \Gamma\{x := e\}\Gamma' \implies \forall \ell \in L_{\mathcal{T}}.\forall M.\forall I_1, I_2. I_1 \overset{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle x := e, M, I_1, O\rangle \rightsquigarrow O \implies \langle x := e, M, I_2, O\rangle \rightsquigarrow O \wedge O \overset{\ell}{=}_{\mathcal{T}} O$. Note that the security label of the variable after the execution of the command carries both implicit (pc) and explicit (t) dependencies.
- Case (IO-TT-If): Based on the induction hypothesis, $pc \sqcup t \vdash \Gamma\{c_b\}\Gamma' \implies \mathit{TNI}_{PI}(\mathcal{T}, c_b)$ for $b = \mathit{true}, \mathit{false}$. Since $pc \sqsubseteq pc \sqcup t$, the statement holds for this case.

- Case (IO-TT-While): Based on the induction hypothesis, we have $pc \sqcup t \vdash \Gamma\{c_{body}\}\Gamma \implies TNI_{PI}(\mathcal{T}, c_{body})$, and $pc \sqsubseteq pc \sqcup t$, thus $pc \vdash \Gamma\{c\}\Gamma \implies TNI_{PI}(\mathcal{T}, c)$ for $c = while\ e\ do\ c_{body}$.
- Case (IO-TT-Seq): Using the induction hypothesis, we have $pc \vdash \Gamma\{c_1\}\Gamma' \implies TNI_{PI}(\mathcal{T}, c_1) \wedge pc \vdash \Gamma'\{c_2\}\Gamma'' \implies TNI_{PI}(\mathcal{T}, c_2)$. Therefore, $pc \vdash \Gamma\{c_1; c_2\}\Gamma'' \implies TNI_{PI}(\mathcal{T}, c_1; c_2)$.
- Case (IO-TT-Input): Taking the condition $pc \sqsubseteq \ell$ into account, the type system only accepts input commands in the same context as the label ℓ or lower. Leaving the premise empty makes the type system unsound, due to not considering implicit flow (pc) to inputs from the level ℓ. Hence, $pc \vdash \Gamma\{input(x, \ell')\}\Gamma' \implies \forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2. I_1 \overset{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle input(x, \ell'), M, I_1, O\rangle \rightsquigarrow O \implies \langle input(x, \ell'), M, I_2, O\rangle \rightsquigarrow O \wedge O \overset{\ell}{=}_{\mathcal{T}} O$.
- Case (IO-TT-Output): The condition $pc \sqcup \Gamma(x) \sqsubseteq \ell$ controls if the output is permitted with regard to the transitive policy; the premise monitors implicit flow (pc) and explicit flow ($\Gamma(x)$) to the output channel at the level ℓ. Thus, we have $pc \vdash \Gamma\{output(x, \ell')\}\Gamma \implies \forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2. I_1 \overset{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle output(x, \ell'), M, I_1, O\rangle \rightsquigarrow O_1 \implies \langle output(x, \ell'), M, I_2, O\rangle \rightsquigarrow O_2 \wedge O_1 \overset{\ell}{=}_{\mathcal{T}} O_2$ since $O_1 = O_2 = O.M(x)_{\ell'}$.
- Case (IO-TT-Sub): $pc_1 \vdash \Gamma_1\{c\}\Gamma_1' \implies TNI_{PI}(\mathcal{T}, c)$. Considering the conditions $pc_2 \sqsubseteq pc_1 \wedge \Gamma_2 \sqsubseteq \Gamma_1 \wedge \Gamma_1' \sqsubseteq \Gamma_2'$, we can conclude $pc_2 \vdash \Gamma_2\{c\}\Gamma_2' \implies TNI_{PI}(\mathcal{T}, c)$. □

Proof of Lemma 5. For simplicity, we write $c' = Canonical(c)$. We know that $\forall x.\big(P(x) \iff Q(x)\big) \implies \big(\forall x.P(x) \iff \forall x.Q(x)\big)$. So to prove the lemma, we show the correctness of the following statement:
$\forall M_1, M_2. \langle c', M_1\rangle \rightarrow^* \langle \text{stop}, M_1'\rangle \wedge \langle c', M_2\rangle \rightarrow^* \langle \text{stop}, M_2'\rangle \implies$
$\Big(\forall \ell \in L_{\mathcal{N}}.\big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'\big) \iff \forall \ell' \in L_{\mathcal{T}}.\big(M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2 \implies M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'\big)\Big)$.
If the execution of the program c' for (at least) one of the two arbitrary memories M_1 and M_2 does not terminate, then the premise in both security definitions does not hold, thus the lemma holds. Assuming the program is terminating for both memories, we prove the statement as follows:

1. Left to right:

   (a) Let $I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} | M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2\}$ be the set of levels in $L_{\mathcal{N}}$ that the two memories are indistinguishable for the set of labels can

flow to them. Then, we have $I_{\mathcal{T}} = \{\ell' \in L_{\mathcal{T}} | M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2\} = \{\ell_{snk}, \ell_{src} \in L_{\mathcal{T}} | \ell \in I_{\mathcal{N}}\}$. Based on Definition 1, $M_1 \overset{\ell_{snk}}{=}_{\mathcal{T}} M_2 \implies M_1 \overset{\ell_{src}}{=}_{\mathcal{T}} M_2$.

(b) Using Lemma 1, we can conclude that $\forall \ell \in I_{\mathcal{N}}. \forall x \in Var_c. \Gamma_{\mathcal{N}}(x) = \ell \implies \Big(\exists x_{sink} \in Var_{c'}. \Gamma_{\mathcal{T}}(x_{sink}) = \ell_{snk} \wedge M_1'(x_{sink}) = M_2'(x_{sink})\Big) \wedge \Big(\exists x \in Var_{c'}. \Gamma_{\mathcal{T}}(x) = \ell_{src} \wedge M_1'(x) = M_2'(x)\Big) \wedge \Big(\exists x_{temp} \in Var_{c'}. \Gamma_{\mathcal{T}}(x_{temp}) = \top \wedge M_1'(x_{temp}) = M_2'(x_{temp})\Big) \wedge \ell_{src}, \ell_{snk} \in I_{\mathcal{T}}$.

(c) Therefore, $\forall \ell \in I_{\mathcal{N}}. M_1' \overset{\ell}{=}_{\mathcal{N}} M_2' \iff \forall \ell' \in I_{\mathcal{T}}. M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'$. Hence, $\forall \ell \in L_{\mathcal{N}}. \Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'\Big) \implies \forall \ell' \in L_{\mathcal{T}}. \Big(M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2 \implies M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'\Big)$.

2. Right to left:

(a) Let $I_{\mathcal{T}} = \{\ell' \in L_{\mathcal{T}} | M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2\}$ and $I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} | M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2\} = \{\ell \in L_{\mathcal{N}} | \ell_{snk} \in I_{\mathcal{T}}\}$.

(b) According to Lemma 1, we have $\forall \ell' \in I_{\mathcal{T}}. \exists \ell \in L_{\mathcal{N}}. \Big(\ell_{snk}, \ell_{src} \in I_{\mathcal{T}} \wedge \forall x \in Var_c. \Gamma_{\mathcal{N}}(x) = \ell \implies \Big(\exists x_{sink} \in Var_{c'}. \Gamma_{\mathcal{T}}(x_{sink}) = \ell_{snk} \wedge M_1'(x_{sink}) = M_2'(x_{sink})\Big) \wedge \Big(\exists x \in Var_{c'}. \Gamma_{\mathcal{T}}(x) = \ell_{src} \wedge M_1'(x) = M_2'(x)\Big) \wedge \Big(\exists x_{temp} \in Var_{c'}. \Gamma_{\mathcal{T}}(x_{temp}) = \top \wedge M_1'(x_{temp}) = M_2'(x_{temp})\Big)\Big)$.

(c) Thus, $\forall \ell' \in I_{\mathcal{T}}. M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2' \iff \forall \ell \in I_{\mathcal{N}}. M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'$. Hence, $\forall \ell' \in L_{\mathcal{T}}. \Big(M_1 \overset{\ell'}{=}_{\mathcal{T}} M_2 \implies M_1' \overset{\ell'}{=}_{\mathcal{T}} M_2'\Big) \implies \forall \ell \in L_{\mathcal{N}}. \Big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'\Big)$. □

Proof of Theorem 9. By using Lemma 2 and Lemma 5. □

Proof of Theorem 10. Similar to the proof of Theorem 3. □

Proof of Theorem 11. We start with showing that $\mathcal{P}, \Gamma_1, pc \vdash c : t \implies \mathcal{P}', \Gamma_1', pc \vdash c' : t$, where $c' = Canonical(c)$ and we extend the typing context Γ_1 to Γ_1' and the labeling function $\mathcal{P}$ to $\mathcal{P}'$ by adding temp and sink variables with the same mappings for any variable x of

the program, i.e., $\forall x \in Var_c. \mathcal{P}'(x) = \mathcal{P}'(x_{temp}) = \mathcal{P}'(x_{sink}) = \mathcal{P}(x) \land \Gamma_1'(x) = \Gamma_1'(x_{temp}) = \Gamma_1'(x_{sink}) = \Gamma_1(x)$.

As discussed in Lemma 1, the program is partitioned in three parts: $c' = init_{c'}; orig_{c'}; final_{c'}$. By induction on the derivation of $init_{c'}$ and using the two rules (NT-Write) and (NT-Seq), we have $\mathcal{P}', \Gamma_1', pc \vdash init_{c'} : t$ because statements are assignments of the form $x_{temp} := x$ and $\Gamma_1'(x) = \Gamma_1'(x_{temp})$. Also, since $\mathcal{P}, \Gamma_1, pc \vdash c : t$ holds, then $\forall \ell \in \Gamma_1(x) \trianglerighteq \mathcal{P}(x)$, and thus $\forall \ell \in \Gamma_1'(x_{temp}). \ell \trianglerighteq \mathcal{P}'(x_{temp})$.

We know that c and $orig(c')$ are identical up to α-renaming of variables $x \in Var_c$ with x_{temp}. Therefore, $\mathcal{P}, \Gamma_1, pc \vdash c : t \implies \mathcal{P}', \Gamma_1', pc \vdash c' : t$ because $\Gamma_1(x) = \Gamma_1'(x_{temp})$, $\mathcal{P}(x) = \mathcal{P}(x_{temp})$, and $x, x_{sink} \notin FV(c')$.

At the final section, statements are the form of $x_{sink} := x_{temp}$. Similar to the init section, because $\Gamma_1'(x_{temp}) = \Gamma_1'(x_{sink})$ and $\forall \ell \in \Gamma_1'(x_{sink}). \ell \trianglerighteq \mathcal{P}'(x_{sink})$, we can write $\mathcal{P}', \Gamma_1', pc \vdash final_{c'} : t$. Applying the rule (NT-Seq) two times, we conclude $\mathcal{P}', \Gamma_1', pc \vdash init_{c'}; orig_{c'}; final_{c'} : t$.

Then, we prove $\mathcal{P}', \Gamma_1', pc \vdash c' : t \implies pc \vdash \Gamma_2\{c'\}\Gamma_3$ where $c' = Canonical(c)$. Remember that in the transitive type system $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} | \ell \in L_{\mathcal{N}}\} \cup \{\top, \bot\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell \trianglerighteq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ such that $\langle L_{\mathcal{T}}, \sqsubseteq \rangle$ is a lattice. To connect the typing contexts together meaningfully, the following constraints must be considered $\forall x \in Var_c$:

- $\Gamma_3(x_{temp}) \sqsubseteq \bigsqcup_{\ell \in \Gamma_1(x)} \ell_{src}$: The final type of x_{temp} is the join of the set of source labels in the last assignment that flow to the variable in the program c', due to flow-sensitivity of the transitive type system, while $\Gamma_1'(x_{temp})$ is the predicted set of *all* information flows to the variable x_{temp}. Thus, $\Gamma_3(x_{temp})$ should be lower than or equal to the join of corresponding source labels of $\Gamma_1'(x_{temp}) = \bigsqcup_{\ell \in \Gamma_1(x)} \ell_{src}$.
- $\mathcal{P}(x) = \ell \implies \Gamma_2(x) = \ell_{src}, \Gamma_2(x_{temp}) = \top, \Gamma_2(x_{sink}) = \ell_{snk}$: The conditions are based on the labeling function presented in Definition 15 to adjust the nontransitive mapping to the transitive one.
- $\Gamma_3(x) = \Gamma_2(x), \Gamma_3(x_{sink}) = \Gamma_2(x_{sink})$: As shown in Figure 9, if the program is well-typed, the types for variables remain untouched except for Var_{temp}.

There is a one-to-one correspondence between typing rules for expressions, which yields the join of $\Gamma(x)$ for free variables $FV(e)$ as the type of the expression e. Thus, $\Gamma_1' \vdash e : t \implies \Gamma_2 \vdash e : t'$.

By induction on the nontransitive typing derivation $\mathcal{P}', \Gamma_1', pc \vdash c' : t$ and the structure of c':

- Case (NT-Skip): Based on the rule (TT-Skip), $pc \vdash \Gamma_2\{c'\}\Gamma_2$ holds.
- Case (NT-Write): We separate this case for two subcases according to

the variable on the left side of the assignment:
  - If $x \in Var_{temp}$, since $\Gamma_1' \vdash e : t \implies \Gamma_2 \vdash e : t'$, based on the rule (TT-Write-I), we write $pc \vdash \Gamma_2\{c'\}\Gamma_2[x \mapsto pc \sqcup t']$.
  - If $x \in Var_{sink}$, we know that $e = x_{temp}$ is the only case in program c' at the $final_{c'}$ section. Because if $\mathcal{P}'(x_{sink}) = \ell'$, then $\forall \ell \in \Gamma_1'(x_{temp}) \cup pc. \ell \in \Gamma_1'(x_{sink}) \wedge \ell \trianglerighteq \ell' \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq \ell'_{snk} \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq \Gamma_3(x_{sink})$. Hence, based on the rule (TT-Write-II), $pc \vdash \Gamma_3\{x := e\}\Gamma_3$.
- Case (NT-If): Using the induction hypothesis and $\Gamma_1' \vdash e : t \implies \Gamma_2 \vdash e : t'$, the statement $pc \vdash \Gamma_2\{c'\}\Gamma_3$ holds for this case with respect to the rule (TT-If).
- Case (NT-While): Similar to the case (NT-If), and according to the rule (TT-While).
- Case (NT-Seq): Using the induction hypothesis, $pc \vdash \Gamma_2\{c_1\}\Gamma_3$ and $pc \vdash \Gamma_3\{c_2\}\Gamma_4$, then $pc \vdash \Gamma_2\{c_1; c_2\}\Gamma_4$ by using the rule (TT-Seq).
- Case (NT-Sub-II): Using the induction hypothesis, we write $pc_1 \vdash \Gamma_2\{c\}\Gamma_2'$. Since $pc_2 \sqsubseteq pc_1$, $\Gamma_3 \sqsubseteq \Gamma_2$, $\Gamma_2' \sqsubseteq \Gamma_3'$ and in combination with the rule (NT-Sub-I), $pc_2 \vdash \Gamma_3\{c\}\Gamma_3'$ holds. □

Proof of Lemma 6. Clearly, the transformation only modifies the labels in the input and output commands of the given program, thus the behavior of the rest of the program stays unaffected. The changes in the labels of the input commands can be formulated as $\forall \ell. I(\ell) = I'(\ell_{src})$, where I is the input for the program c and I' is the input for the program c'.

By induction on the semantic rules shown in Figure 24, it is proven that c' progresses the same as c with the difference that outputs are sent to the channel ℓ_{snk} instead of ℓ. Therefore, we formulate it for the two outputs O and O' of programs c and c' respectively as $O' = O[v_\ell \mapsto v_{\ell_{snk}}]$. Thus the only difference between the output sequences O and O' are the labels of output values; ones with the label ℓ in O are recorded at the same index in O' with the label ℓ_{snk}. □

Proof of Theorem 12. Let $c' = Transform(c)$, $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} | \ell \in L_{\mathcal{N}}\} \cup \{\top, \bot\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell \trianglerighteq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ ($\trianglerighteq$ is reflexive) such that $\langle L_{\mathcal{T}}, \sqsubseteq \rangle$ is a lattice, and $\forall x \in Var_c. \Gamma_{\mathcal{N}}(x) = \ell \implies \Gamma_{\mathcal{T}}(x) = \ell_{src}$.

1. We have $\forall \ell \in L_{\mathcal{N}}. \forall I_1, I_2. I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \iff \forall \ell' \in L_{\mathcal{T}}. \forall I_1', I_2'. I_1' \overset{\ell'}{=}_{\mathcal{T}} I_2'$ because of Definitions 8 and 12. Based on Lemma 6, we also know $\forall \ell \in L_{\mathcal{N}}. I(\ell) = I'(\ell_{src})$.

2. According to Definitions 10 and 14, and the semantic relation presented in Lemma 6, the

statement $\forall M.\Big(\forall \ell \in L_{\mathcal{N}}.\forall I_1, I_2.\big(I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \quad \wedge \quad \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1\big) \Longrightarrow \exists O_2.\langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \wedge O_1 \overset{\ell}{=}_{\mathcal{N}} O_2\Big) \Longleftrightarrow \Big(\forall \ell' \in L_{\mathcal{T}}.\forall I_1', I_2'.\big(I_1' \overset{\ell'}{=}_{\mathcal{T}} I_2' \quad \wedge \quad \langle c', M, I_1', \varnothing\rangle \rightsquigarrow O_1'\big) \Longrightarrow \exists O_2'.\langle c', M, I_2', \varnothing\rangle \rightsquigarrow O_2' \wedge O_1' \overset{\ell'}{=}_{\mathcal{T}} O_2'\Big)$ holds if $O_1 \overset{\ell}{=}_{\mathcal{N}} O_2 \Longleftrightarrow O_1' \overset{\ell'}{=}_{\mathcal{T}} O_2'$.

3. As stated in Lemma 6, we conclude $O_1' = O_1[v_\ell \mapsto v_{\ell_{snk}}] \wedge O_2' = O_2[v_\ell \mapsto v_{\ell_{snk}}]$. Hence, $O_1 \overset{\ell}{=}_{\mathcal{N}} O_2 \Longleftrightarrow O_1' \overset{\ell'}{=}_{\mathcal{T}} O_2'$ and consequently, the theorem holds. □