Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities

Z. BERKAY CELIK, Penn State University
EARLENCE FERNANDES, University of Washington
ERIC PAULEY, GANG TAN, and PATRICK MCDANIEL, Penn State University

Recent advances in Internet of Things (IoT) have enabled myriad domains such as smart homes, personal monitoring devices, and enhanced manufacturing. IoT is now pervasive—new applications are being used in nearly every conceivable environment, which leads to the adoption of device-based interaction and automation. However, IoT has also raised issues about the security and privacy of these digitally augmented spaces. Program analysis is crucial in identifying those issues, yet the application and scope of program analysis in IoT remains largely unexplored by the technical community. In this article, we study privacy and security issues in IoT that require program-analysis techniques with an emphasis on identified attacks against these systems and defenses implemented so far. Based on a study of five IoT programming platforms, we identify the key insights that result from research efforts in both the program analysis and security communities and relate the efficacy of program-analysis techniques to security and privacy issues. We conclude by studying recent IoT analysis systems and exploring their implementations. Through these explorations, we highlight key challenges and opportunities in calibrating for the environments in which IoT systems will be used.

CCS Concepts: • **Security and privacy → Software and application security**; • **Software and its engineering → Automated static analysis**; **Dynamic analysis**;

Additional Key Words and Phrases: IoT security and privacy, IoT programming platforms, program analysis

ACM Reference format:
Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2019. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Comput. Surv.* 52, 4, Article 74 (August 2019), 30 pages.
https://doi.org/10.1145/3333501

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. Earlence Fernandes is supported by the University of Washington Tech Policy Lab and the MacArthur Foundation.
Authors' addresses: Z. B. Celik, Department of Computer Science, Purdue University, West Lafayette, Indiana, 47907; email: zcelik@purdue.edu; E. Fernandes, Department of Computer Science, University of Wisconsin-Madison, Madison, Wisconsin, 53706; email: earlence@cs.wisc.edu; E. Pauley, G. Tan, and P. McDaniel, Department of Computer Science and Engineering, Pennsylvania State University (Penn State), State College, PA, 16802; emails: eap5377@psu.edu, {gtan, mcdaniel}@cse.psu.edu.

0360-0300/2019/08-ART74 $15.00
https://doi.org/10.1145/3333501

1 INTRODUCTION

The introduction of IoT devices into public and private spaces has changed the way we live. For example, home applications that integrate smart locks, thermostats, switches, surveillance systems, and appliances allow users to monitor and interact with their living spaces from anywhere. While industry and users alike have embraced IoT, concerns have been raised about the security and privacy of digitally augmented spaces [39, 49, 85]. 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 [20]. In addition, 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 [19].

Driven by consumer concerns, one of the central criticisms of IoT is that existing platforms lack the essential tools and services to analyze security and privacy. Such criticisms have not gone unnoticed. Recent technical community efforts have proposed a range of tools to identify sensitive data leaks in IoT apps [19, 40], while others have focused on improving IoT safety and security [20, 54, 97, 102]. Works in this area use program-analysis techniques to design and build algorithms that identify vulnerabilities and dangerous behavior within a targeted IoT programming platform. These works motivate our work to study security and privacy issues in IoT that are solved by program-analysis techniques.

While thematically similar to program analysis in mobile apps and other domains, from our study of five major IoT programming platforms (Samsung's SmartThings, Apple's HomeKit, OpenHAB, Amazon AWS IoT, and Android Things), we have found that IoT programming platforms present unique characteristics and challenges in program analysis when compared to other platforms [19]. First, in the case of Android, a well-defined intermediate representation (IR) is available, and analysis can directly analyze IR code. However, IoT programming platforms are diverse, and each uses its own programming language. Second, IoT integrates physical processes with digital connectivity through a diverse set of devices, each of which has a different set of internal device states (e.g., door locked/unlocked); thus, identifying security and privacy issues through these physical states is quite subtle. For example, an adversary can break into a home by changing the thermostat temperature value that causes the windows to open once the temperature reaches a threshold value [21, 30]. Last, each IoT programming platform has its own idiosyncrasies that can pose challenges to program analysis. For instance, the SmartThings platform allows apps to perform call by reflection and make web-service requests; each of these features makes program analysis more difficult and requires special treatment. Due to these domain-specific challenges, ensuring the safety, security, and privacy of IoT systems is not a trivial endeavor.

In this work, we present security and privacy issues in IoT that motivate program analysis techniques. We contrast program analysis in IoT with other domains, demonstrating key differences that complicate analyses. We first study five IoT programming platforms to gain insights into the structure of their apps. We then present IoT-specific issues that require program-analysis techniques within an IoT app or multiple-apps colocated in an environment. We focus on areas that prior research has addressed and others that remain open problems. Last, we demonstrate a number of IoT program idiosyncrasies that require special treatment and present several general precision requirements for IoT code analysis by providing examples from IoT apps. We conclude by studying a representative set of recent IoT analysis systems from literature. Our study serves as a guideline for researchers and provides insights into the design and implementation of IoT program analysis for security and privacy. In this work, we explore the following:

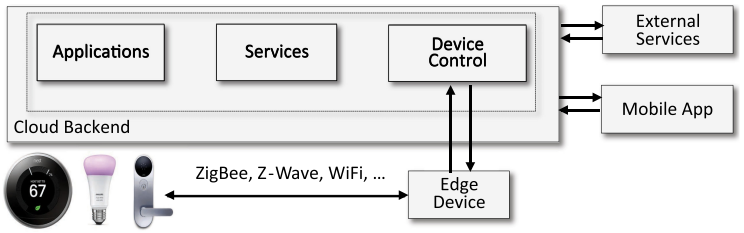

Fig. 1. An example architecture of edge-based IoT system.

- We conduct a study of five major IoT programming platforms to understand their program structures. We map their program structures to a sensor-computation-actuator idiom that includes the common building blocks of IoT apps.
- We present IoT-specific issues, IoT program idiosyncrasies, and general precision requirements for IoT app analysis with examples from 230 SmartThings IoT apps. We discuss the problems that the research community has already started to study and the areas that need attention. In highlighting open issues, we draw insights and motivate future work.
- We study six IoT analysis systems from literature for security and privacy that incorporate program-analysis techniques. We measure their ability to analyze IoT apps and evaluate their approaches to IoT-specific issues. We note that we limit our analysis to publications that use program analysis for IoT security and privacy and were published at a major venue.

Scope. This work is at the intersection of three domains: IoT programming platforms, program analysis, and security and privacy. IoT programming platforms provide a software stack to develop applications that monitor and control devices. Program analysis includes the techniques used for analyzing the behavior of an IoT app or multiple-apps in an environment. Security and privacy cover the objective of the program-analysis techniques to identify potential security and privacy issues. We begin below by giving an overview of IoT systems and program structures of IoT platforms.

2 BACKGROUND

We start with an overview of how IoT systems structure their design (Section 2.1). We then present recent research on IoT security and privacy (Section 2.2). As IoT is a diverse domain, we focus on consumer IoT, which has the largest number of applications and the most significant market [95].

2.1 An Overview of IoT System Architectures

IoT systems integrate physical processes with digital connectivity. These systems are used to achieve simple tasks such as motion-activated light switches as well as complex tasks such as controlling the traffic lights in a smart city. Regardless of their purpose and complexity, IoT systems often structure their architecture from bottom to top with (1) devices, (2) connectivity protocols, and (3) IoT programming platforms (see Figure 1). These systems often use an edge device as a centralized gateway that connects devices in a physical environment, use a cloud backend to synchronize device states, and provide interfaces for remote control and monitoring of devices.

Devices are equipped with embedded sensors and actuators that interact with a physical environment. Sensors collect physical states and send events to other devices, the hub, or the cloud. These events are processed and used to actuate the devices. For example, a presence sensor detects a presence event and communicates with a switch (actuator) that turns on the lights. We note that

a mobile phone or even a coffee machine can be a sensor as long as it can gather information about its environment. Protocols are used to establish communication between heterogeneous devices and network endpoints. These protocols are selected according to the requirements of the environment, such as low power or non-lossy connection. For instance, the Bluetooth Low Energy (BLE) protocol is used for short-range communication and is extremely energy-efficient.

IoT programming platforms deliver app-specific services by managing devices and their interactions. They also enable crucial functions such as data collection, control, and interoperability. In recent years, several IoT programming platforms have emerged in a wide range of domains: Apple's HomeKit [8], OpenHAB [43], Samsung's SmartThings [2] for smart home, Android Sensor API [31], Google Fit for wearables [33], ThingWorx [51] for aerospace, Eclipse Kura [32] for general-purpose solutions, and FarmBeats [99] for agriculture. These platforms offer web-based environments and tools that enable developers to write applications used to create custom automations. Applications use a diverse set of languages and execute in a variety of environments (e.g., the cloud or a local hub). Further, in some IoT platforms, applications are written in a Domain Specific Language (DSL) [43] and applications run in a sandbox for performance and security purposes [90].

2.2 IoT Security and Privacy

The growth of IoT devices has had profound impacts in settings such as the automotive industry [57], aviation [27], smart homes [39], medical wearables [108], agriculture [99], and smart cities [111]. The broad adoption of IoT among consumers and industry also raises concerns about security and privacy [19, 61, 75], with some arguing for more rigorous standards and regulation surrounding its use [35]. Failures in IoT environments could lead to privacy violations (e.g., compromised baby monitors [103]), or safety and health consequences such as vehicle crashes and monetary theft [100], failed IoT pacemakers [94], and pipeline explosions [52].

In response to the security and privacy threats in IoT, most attempts to date aim to improve perimeter defenses that harden the IoT infrastructure against attacks using firewalls [58], intrusion detection systems [112], access control policies [48], and software patches [64]. Other efforts have explored vulnerability analysis within specific IoT devices and IoT programming platforms. Oluwafemi et al. [74] investigated the security risks in smart lights controlled by compromised automation systems, and Ho et al. [49] studied the vulnerabilities of smart locks. Fernandes et al. discovered design flaws in permission control of the SmartThings IoT platform [39], and Xu et al. [107] surveyed the security problems in IoT hardware design. 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. Past analysis of IoT devices and environments have also focused on securing an IoT app through source-code analysis. Most previous studies rely on techniques designed for mobile phone security [11, 26, 36, 45, 81, 114]. For instance, some systems infer an app's context to enforce permissions based on that context through run-time prompts [54] or asking users for authorization through an interface [97].

There are also several recent surveys on IoT security and privacy, which differ in scope and focus from this work. These surveys 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 [5]. Roman et al. performed a study on reported IoT attacks and defenses [83]. Others focused on security analysis of IoT architectures [113], available security solutions [55], and privacy threats [1, 115]. However, this work studies the space of IoT application security and privacy research through program-analysis techniques. Those seeking a survey of IoT more broadly can look to many recent papers covering this rapidly developing area [41, 49, 74, 80, 89, 107, 110].

Table 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	✓	✓	✓ [82]	✓ [44]	Groovy
OpenHAB	Hub	Hub	✓	◓•	✓ [76]	✓ [77]	Xtend-based DSL
Apple's HomeKit	Hub	Hub	✓	✓	n/a+	n/a	Swift/Objective C
Android Things	Cloud	Cloud	✓	✓	✓ [10]	n/a	Java
Amazon AWS IoT	Both	Cloud	✓	✓	n/a	n/a	SQL-like, (Java, Python, 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.

3 IOT PROGRAMMING PLATFORMS

IoT platforms provide a software stack used to develop apps that monitor and control IoT devices. In 2018, there are hundreds of IoT platforms in the marketplace [95]. 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 (Section 3.1). Table 1 summarizes our study. Our survey 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 app (Section 3.2).

3.1 Overview of IoT Programming Platforms

Samsung's SmartThings consists of a hub, apps, and the cloud back-end [20, 46]. 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 back-end 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 (see Listing 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.

OpenHAB is an open-source automation platform built in the Eclipse IDE [43]. 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). 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

```
/* 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 of an app */
def updated() {...}
def initialize() {...}
def installed() {
    subscribe(device, "device event", handler)
}
def handler() {
// Computation and actuators.
}
```

Listing 1. SmartThings IoT application structure.

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

Listing 2. OpenHAB IoT rule structure.

similar to the Xtend language [34]. Users can install OpenHAB apps by placing them in the rules folder of their installation directories or by downloading from the Eclipse IoT marketplace [77].

Apple's HomeKit is a development kit that manages and controls compatible smart devices [8]. The HMHomeManager class describes a set of homes (locations). An HNHome class defines each house and each room within that set. Each room may 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. Last, 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 programs 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 (see Listing 3). Users can install HomeKit apps using the Home mobile application provided by Apple [9].

Amazon Web Services (AWS) IoT provides communication between smart devices and the AWS Cloud [96]. 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

```
/* 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) */
}
```

Listing 3. Apple HomeKit IoT application 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. */
      }
    ]
}
```

Listing 4. AWS IoT rule structure.

```
public class ClassName extends Activity{
    protected void onCreate(...) {
      // Detect events and 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
    }
}
```

Listing 5. Android Things IoT application structure.

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. Furthermore, AWS IoT supports SQL-like rules, which are used for filtering messages sent to AWS IoT Core and transfers them to other devices or an AWS cloud service (see Listing 4). A rule can use data from many devices and perform a set of actions at the same time.

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 Google Play services. Android Things uses the same lower layers of the stack as Android. For the app framework, the Things Support Library is 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 5). The apps then monitor device state changes through listeners. When a device event happens, a callback is triggered to implement app functionality.

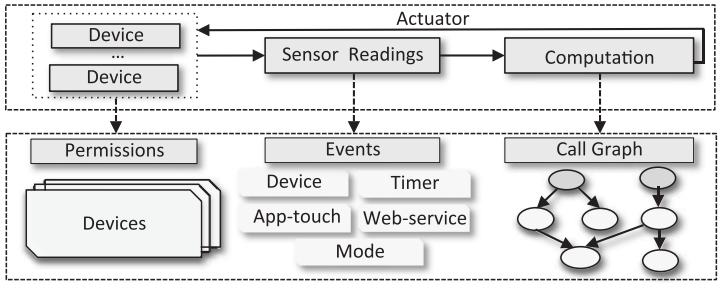

Fig. 2. Mapping IoT application structures to the sensor-computation-actuator idiom.

3.2 Generalizing IoT Application Structure

A broad investigation of dominant IoT platforms shows that IoT systems structure their apps' design around the *sensor-computation-actuator* idiom regardless of their purpose and complexity [19, 20]. Therefore, the source code of an IoT app can be translated to a platform-agnostic structure with three types of common building blocks as shown in Figure 2: (1) *Permissions* grant access to devices and user inputs used in the app to implement the app functionality; (2) *Events* reflect the association between sensor readings and actuators: when a sensor reading is triggered, a device is actuated; and (3) *Call graphs* represent the relationship between main methods and call-sites in the app.

Permissions are granted when an app is installed or updated. This is where various types of devices and user inputs are described and granted access. Apps can only interact with devices for which they have been given permission. Devices have capabilities of *actuators* and *sensor readings*. Actuators represent the actions that a device can do and sensor readings represent the state information of devices. Actuators and sensor readings are not one-to-one. While a device may support many sensor readings, it may have a limited number of actuators, e.g., a door may have opening, opened, closing, and closed sensor readings, but has only open and close actuators.

Events connect particular sensor readings and handler methods. That is, when an event through a sensor reading is triggered by a device, an associated event handler of an app is invoked. Event handlers may actuate changes in the state of the devices. For instance, when a motion sensor reports a motion-active event, an app may invoke an event handler to actuate a light switch from off to on. We found that events are not limited to device events; while different IoT platforms name these differently, we call them *abstract events* and classify them into four different groups [19][1]: (1) *Timer events*: event-handlers are scheduled to take actions within a particular time or at predefined 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) *App touch events*: for example, some action can be performed when the user taps on a button in an app; (3) *External events*: IoT programming platforms may allow an app to be accessible over the web; this enables external entities (e.g., If This Then That (IFTTT) [50]) to make requests to the app and get information about or control end devices; (4) what actions get generated may also depend on *mode events*, which are behavior filters that are used to 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.

[1]During the time we wrote the article, some platforms started supporting additional abstract events. One such example is OpenHAB's system events, which are triggered when a system boots up or shuts down. We refer readers to platform documentation for a complete set of events a platform supports.

An IoT app does not have a main method (i.e., entry point) due to its event-driven program structure. Apps implicitly define entry points by subscribing events through event handler methods. An app may have multiple entry points by subscribing to multiple events. Additionally, apps often call other functions in event handlers to implement logic, send messages, or log device events to a database. A call graph is used to represent this control-flow relationship between a particular event handler and other functions the event handler invokes.

4 PROGRAM ANALYSIS OF IOT APPLICATIONS

Program-analysis techniques operate on IoT app source code to achieve a variety of goals, such as understanding apps' security. In this section, we begin by identifying common program analysis goals to understand security and privacy threats (Section 4.1), followed by a description of the type of program analysis techniques (Section 4.2). In the next section, we classify the program analysis issues into three groups and discuss each of them (Section 5).

4.1 Goals of Analyses

We first discuss several common goals of performing program analysis on IoT apps. Many of these goals remain open problems; thus, understanding the goals can guide future work.

Sensitive Data Leaks. IoT devices have access to data that can be intensely private, e.g., the door is locked or unlocked, and users are at present home or away [19]. IoT platforms ensure only coarse-grained access controls to sensitive information and provide limited controls over how that information is used. 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 any other entity.

Abuse Prevention. IoT apps necessarily have access to functions that if abused would put the user's safety and security at risk, e.g., unlock doors when the user is not at home [49] or create unsafe or damaging conditions by turning on a smart oven [29]. Therefore, it is crucial to prevent IoT apps from abusing device capabilities by ensuring those apps operate devices according to a set of security, safety, and functional properties [20].

Permission Misuse. The permission model of an IoT platform defines an app's access to sensitive actions such as device state changes. However, IoT apps may misuse permission models. This can happen for two main reasons: First, a permission model may be coarse-grained and conflate permissions of devices; for example, an app granting the permission to a door lock grants access to both door-lock and door-unlock actions, even though the app may only need the privilege of locking the door [63]. Second, an app may trick users to acquire unneeded and dangerous device permissions; for example, a smoke-alarm app may request the permission of a security camera to disable it, even though the app does not need the permission to function [97].

Data Provenance. As IoT apps perform increasingly diverse activities, attacks and misconfigurations require investigation. To address this, provenance systems use program instrumentation that aims to collect IoT app information to construct complete and accurate app behavior. After that, they aggregate that information into a data structure such as provenance graphs for forensics and system diagnosis. For instance, a provenance system designed for IoT apps may provide complete history of device actions and events, which can be used to identify the cause of an attack [12, 102].

4.2 Type of Program Analysis

Previously covered issues can be addressed through static or dynamic code analysis, and in some cases, issues are related to both. For example, path sensitivity is not an issue in dynamic analysis, since they follow execution paths; they instead suffer from coverage problems. We split these

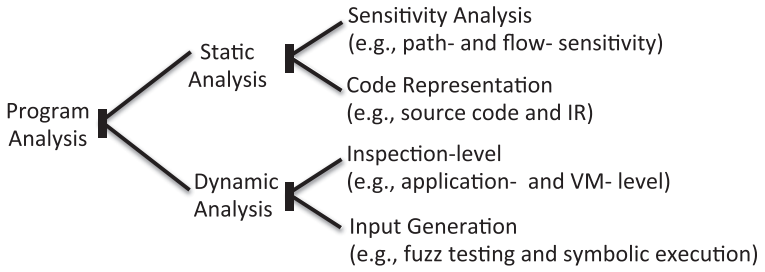

Fig. 3. Categorization of issues based on program analysis type.

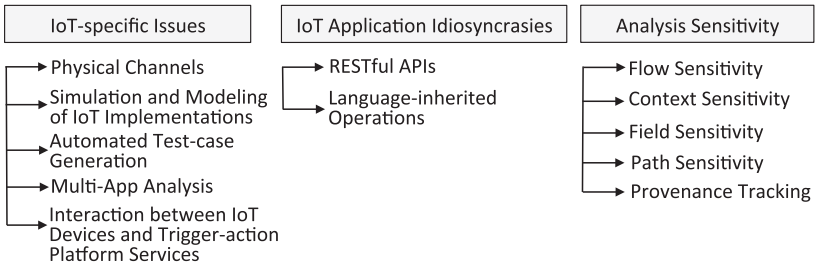

Fig. 4. Categorization of issues in IoT program analysis discussed in Section 5.

issues based on the analysis type as shown in Figure 3. In static analysis, the source code of an app is analyzed without running it, and in dynamic analysis, the code is run, possibly under-instrumented conditions, to see if there are likely problems [4]. Static analysis benefits from analyzing the complete source code whereas, in dynamic analysis, only a portion of the code is executed; thus, analysis results are limited to observed executions. Furthermore, static analysis may lead to over-approximations by generalizing all possible behaviors of a program, risking false positives [37]. For instance, an analysis tool detects a sensitive data leak through a piece of code in an IoT app that is not executable at run-time. A dynamic analysis may under-approximate because the execution inputs of a program are often incomplete; thus the analysis may produce false negatives. For example, an analysis tool may miss vulnerabilities or malicious behaviors at run-time.

5 ANALYSIS OF IOT PROGRAMS

We split the analysis characteristics and challenges of IoT apps into three groups as shown in Figure 4. In this section, we begin by introducing issues and challenges in IoT program analysis (Section 5.1). We then detail IoT-specific analysis issues (Section 5.2) and IoT application idiosyncrasies (Section 5.3). Last, we present general precision requirements for IoT code analysis (Section 5.4).

5.1 Issues and Challenges in IoT Program Analysis

Program analysis has been applied, either statically or dynamically, to many different settings such as mobile apps. From our study of five IoT platforms, we found that IoT platforms possess a few unique characteristics and issues when compared to other platforms.

First, in the case of Android, a well-defined Intermediate representation (IR) is available, and analysis can directly analyze IR code. For instance, popular analysis frameworks including Soot [59] and WALA [66] that have been used to analyze Android app source code provide libraries to convert Dalvik bytecode to the Jimple IR [14] to construct call graphs [65] and to perform inter-procedural dataflow analysis via graph reachability [16]. However, IoT programming platforms

Table 2. Description of Official and Third-party SmartThings IoT Apps Used in our Discussion

	Number of Apps		Unique Device Types		Avg/Max LOC	
App functionality	**Official**	**Third-party**	**Official**	**Third-party**	**Official**	**Third-party**
Convenience	80	26	49	37	244/2,633	247/1,360
Security and Safety	19	10				
Personal Care	10	0				
Home Automation	48	24				
Entertainment	10	0				
Smart Transport	1	2				

†We determined an app's functionality by checking definition blocks in its source code.

are diverse, and each uses its own programming language. Therefore, the analysis must capture the event-driven nature of IoT apps and perform analysis on it.

Second, IoT apps control physical hardware peripherals and drivers. Consequently, IoT apps have qualitatively different vulnerabilities resulting from handling physical processes such as temperature, smoke, motion, humidity, water leak, and luminance. For instance, an adversary might misuse the capability of an IoT device through physical channels to achieve a damaging effect. To illustrate, we consider an app that grants permissions to a smart light, which supports color and intensity capabilities. The app may strobe light at a frequency and change colors to various shades, which could trigger seizures in users who have photosensitive epilepsy [84].

Third, IoT apps may interact with each other when they are co-located in an environment. The interaction between apps, among others, may happen when a device action executed in an app's event handler is used as an event to trigger another app's event handler [20]. For instance, two apps interact with each other when the "switch off" action of an app is used as a "switch turned-off" event in another app. The interactions among apps may lead to undesirable device states causing security and safety violations and exposing users to risks such as a locked door when there is a fire.

Fourth, trigger-action platforms such as IFTTT [50], Zapier [106], and Microsoft Flow [78] are increasingly used to bridge the divide between physical (e.g., IoT devices) and digital (e.g., e-mail services, social media platforms) processes. These platforms allow users to use rules that connect the events and actions of IoT devices with the events and actions of digital services. For example, a user may use a rule that posts a Tweet when she turns on the light in the living room, 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 [21, 93]; for example, an IoT app that subscribes to the switch "turn-on" event interacts with a trigger-action platform rule that "turns on" the switch when the user is tagged in a photo on Facebook.

Last, each IoT platform has its own idiosyncrasies that can pose challenges to program analysis. For instance, SmartThings IoT apps written in the Groovy programming language that allows apps to perform call by reflection and allows web-service apps; each of these features makes the analysis more complicated and requires special treatment.

Example Code Blocks. During our discussion, we will provide example code blocks obtained from our analysis of 230 SmartThings apps [19]. We primarily reference SmartThings, because a large number of open-source market apps are available, and it has a detailed, publicly available documentation that helps validate our findings [90]. In late 2017, we obtained 168 official (vetted) apps from the SmartThings GitHub repository [82] and 62 community-contributed third-party (non-vetted) apps from the SmartThings community forum [44] (see Table 2).[2] These apps

[2]The apps are available at our IoTBench test-suite repository [69].

```
/* An app leaks information by changing light intensity */
/* Similar logic can be used to strobe the light */
/* The app may subscribe to the presence sensor's "not-present" state to know that users are not home */

subscribe(motion, "motion.inactive", motionInactiveHandler)
def motionInactiveHandler(evt) {
    runIn(60 * minutes, checkMotionStatus)
}
def checkMotionStatus(evt) {
    if (evt.value == "inactive") { // motion inactive
      // setting intensity of the switch 0
      myLight.setLevel(0)
      changeIntensity()
    }
}
def changeIntensity() {
    def value = myLight.currentState("level")
    // misuse light functionality
    if (value<=20) {
      state.bool=true
      myLight.setLevel(value+20) }
    if (value>20 && value<80 && state.bool) {
      myLight.setLevel(value+20) }
    if (value>=80) {
      state.bool=false
      myLight.setLevel(value-20) }
    if (value>20 && value<80 && !state.bool) {
      myLight.setLevel(value-20) }
    // change light intensity every 3 seconds
    runIn(60*0.05,changeIntensity)
}
```

Listing 6. An example code block that leaks sensitive information through physical channels.

were selected to include various IoT devices and contexts that encompass diverse real-life use cases.

5.2 IoT-specific Analysis Issues

IoT apps possess unique characteristics and challenges in terms of program analysis when compared to other platform apps. In this section, we enumerate five challenges that are mainly due to the capabilities provided by IoT platforms to the apps.

Physical Channels. IoT devices integrate physical processes into digital connectivity. Misuse of physical processes allows an app to deviate from a device's intended functionality to achieve an unexpected effect. We give three examples of physical processes that lead to security and privacy issues: (1) data leaks through side channels, (2) health-related risk through device functionality misuse, and (3) safety issues through indirect physical interactions.

We demonstrate the first two examples with an app that grants access to a light device. The light has the capability to change color, hue, saturation, and intensity level. The first example is an app that creates a side channel by changing the light intensity to notify an adversary or another app when the households are sleeping or not at home [19, 54] (see Listing 6). The second example is an app that flashes the lights by adjusting the light intensity and changes the light color at regular intervals. This process creates visual stimuli that can trigger seizures in people who suffer from photosensitive epilepsy [84]. Similar health-related risks can be inflicted on users through other physical processes such as temperature and sound. To address the misuse of physical processes in these examples, one solution would be to construct a set of templates that define insecure and unsafe device states for side channels and health-related risks. For instance, a template says that an app must not change the volume of a music player above a threshold to prevent hearing loss

and tinnitus. The analysis then tracks the device states either at install time or run-time to ensure that an app does not cause the volume state to exceed a threshold or create spikes.

In the third example, an adversary controls a physical process to control some other devices indirectly. For instance, an adversary increases the room's temperature by turning on the heater to activate an app that opens the window when the room temperature exceeds a threshold value [20, 30]. This process would allow a burglar to break into homes via windows by controlling the room's temperature. To address indirect access to devices, an app may add additional path conditions to guard device actions based on the app's context. Turning to our example, the window would be open when the temperature value is above a threshold and with some additional conditions such as when the user is at home and when the time is between sunrise and sunset.

Simulation and Modeling of IoT Programs. A collection of many IoT devices forms a complex system that requires simulators to execute and analyze them accurately. In contrast to traditional modeling and simulation frameworks, simulation of large-scale and heterogeneous IoT environments requires capturing the state of many devices and the interdependence between events, actions, and computational logic [56]. The research community and industry have recently explored the requirements for modeling and simulation of IoT implementations [3, 28, 47, 62]. For instance, IoT-lab provides an infrastructure for testing heterogeneous IoT devices [3], and IoTify enables IoT application development by simulating virtual devices in the cloud [56]. However, to our knowledge, current IoT simulation tools that researchers often use (e.g., SmartThings web-based IoT simulator [104]) have insufficient support for diverse devices and events, which prevents the simulation of apps that have various functionality.

Another noteworthy point is that physical processes of devices including temperature, illuminance, power consumption, and humidity are often hard to replicate in a simulated environment. Similar to simulating cyber-physical systems and other physical process-driven systems, IoT analysis tools must consider the evolution of the state of an IoT system over time. This requirement motivates the need for an IoT simulation environment that executes IoT apps by means of a discrete-event simulation engine through continuous-time solvers and state machine-based modeling [62].

Automated Test-case Generation. Dynamic analysis of IoT apps requires input data for execution of the apps [86]. In IoT apps, inputs are the events that trigger the apps (i.e., entry points of an app) and user and device inputs. This introduces a challenge of automating systematic and scalable input generation for IoT apps that control a diverse set of devices with a wide range of internal states. For instance, devices such as a thermostat and power meter may have a discrete (e.g., integer-valued) or continuous attributes that would lead to a large input space—generating an input for every possible value in such cases would result in a large number of test cases.

Similar to other computing platforms, fuzzing and symbolic execution can be used to increase code-coverage for an automated test-case generation. Fuzzing executes the app with random input data, and symbolic execution uses symbolic inputs to perform path-based exploration [17]. For instance, tools for Android, such as Google's Android Monkey [38], generate random test case inputs of user events and system-level events. As another example, IoTFuzzer uses a dynamic analysis to identify IoT app content and mutates that content to detect memory corruptions of IoT devices [22]. To improve test input generation, contemporary approaches use heuristics that guide input generation to cover app source code intelligently, avoid redundant test paths, and enable multi-objective automated testing [18, 24, 67, 79, 101]. Yet, to our knowledge, tools that automate test input data and event generation to execute IoT apps are largely non-existent. This motivates future work to improve test-case generation techniques as applied to IoT.

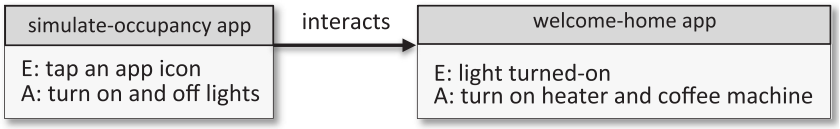

Fig. 5. An example of interacting IoT apps. `simulate-occupancy` app interacts with `welcome-home` app through the light turn-on event. (E is for Event, and A is for Action.)

Multi-app Analysis. Multi-app analysis that targets IoT environments studies the joint behavior of the apps, whereas individual app analysis considers each app in isolation. In a multi-app analysis, apps interact through a common device or abstract events [20, 21, 71]. More specifically, we found that 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 switch 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 motion is detected, one app turns on a switch while another app turns off the switch. We found that apps also interact through *modes*, which are behavior filters that automate device actions. For instance, an app that changes the "away" mode to the "home" mode when a user arrives home interacts with an app that uses the "mode change" event to activate the security alarm.

The interactions among devices may cause security, safety, and privacy risks even though individual apps are safe in operation [20, 23, 30, 71]. To illustrate, we consider `simulate-occupancy` app co-resident with `welcome-home` app (see Figure 5). `Simulate-occupancy` turns on and turns off the light switch to simulate occupancy when the user is not home. `Welcome-home` brews coffee and turns on the heater when the light is turned on. `Simulate-occupancy` interacts with `welcome-home` through the "light-on" event. However, unexpected behavior may happen when these apps interact with each other. In the example, the analysis reveals a safety violation when the user is not home: the heater and coffee machine are turned on when the bedroom light is turned on because the "light-on" is used as an event in `welcome-home` app. To prevent undesired and unsafe states through interactions, an analysis requires finding the interactions among apps, developing policies for undesired device states, checking that the app conforms to those properties when interacting with other apps, and blocking the states causing the policy violations.

Interaction between IoT Devices and Trigger-action Platform Services. Trigger-action platforms such as IFTTT [50], Zapier [106], and Apiant [109] allow users to connect services together. Services include 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 their SmartThings account. Services communicate with each other using REST APIs over HTTP [42]. 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. For instance, as of May of 2018, IFTTT has the largest market share [68]; it provides users with 500 services, 158 of which are IoT services. IFTTT enables IoT applications such as fitness trackers and other wearables, hobbyist projects, and connected homes. DO rules act as virtual buttons, which can trigger a set of actions; for example, a DO rule may turn on a smart switch when

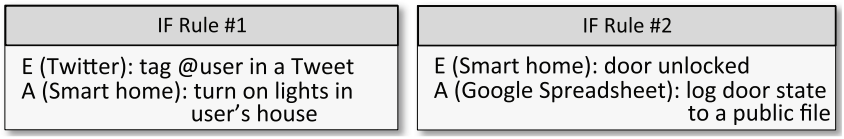

Fig. 6. Example IF rules in a trigger-action platform. The left IF rule causes an integrity violation, and the right one violates user privacy. (E is for Event, and A is for Action.)

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.

Similar to multi-app analysis, the interaction between IoT devices and trigger-action platform services may cause security and privacy issues [15, 21, 93]. Figure 6 shows two examples of IF rules that connect a smart home to Twitter and Google Spreadsheet services. `IF rule #1` turns on lights when a user is tagged in a Twitter post. `IF rule #2` logs the door state to a public spreadsheet when the door is unlocked. In the first example, an integrity violation occurs, because the untrusted event (Tweet post) changes the state of a trusted action (light on). In the second example, a confidentiality violation occurs, because the sensitive information (door unlocked) is made publicly available. Another point worth noting is that these services may also create interactions between IoT apps and services—the actions and events of services and IoT apps can be linked together, similar to the case of interaction between multiple apps.

Analyses targeting trigger-action platforms require information flow analysis that considers the security and privacy of the environment. More specifically, an analysis may extract the events and actions of the trigger-action rules and label them with the integrity and confidentiality labels. For instance, unlocking a door might be labeled with trusted, and saving a device state to a public file might be labeled confidential. We found that this process is not a trivial endeavor, because trigger-action rules are strings and a rule's event and actions often do not match with the capabilities defined in an IoT programming platform. For instance, Santa detector IFTTT rule's definition [7] says that "Ho ho ho! Receive a notification when Santa arrives to deliver you some Merry Christmas joy (and presents)." Determining the actions and events and labeling them may need user help or advanced natural language processing techniques.

5.3 IoT Application Idiosyncrasies

Each IoT platform has its own idiosyncrasies based on how they structure the apps and the programming languages they use. These idiosyncrasies require special treatment for analysis precision. In this subsection, we give a couple of example idiosyncrasies.

RESTful APIs. RESTful APIs allow external entities to access smart devices and manage those devices. For instance, an app can set the cooling point of a climate control system when the temperature value obtained from a weather forecasting service is above a threshold. These apps declare mappings that relate endpoints, HTTP operations, and callback methods. The SmartThings platform names these apps web-service apps [87], other platforms provide similar functionality through APIs that enables communicating with the external services. For instance, AWS IoT Core allows both companion and server apps to access connected devices through RESTful APIs [96]. Listing 7 shows a code snippet of a SmartThings web-service app. The `/switches` endpoint handles an HTTP GET request and returns the state information of configured switches by calling the

```
1  /* An example use of Restful APIs */
2  mappings {
3    path("/switches") {
4      action: [GET: "listSwitches"] }
5    path("/switches/:command") {
6      action: [PUT: "updateSwitches"] }
7  }
8  def listSwitches() {
9      switches.each {
10       resp << [name: it.displayName, value:
11                 it.currentValue("switch")] }
12     return resp
13 }
```

Listing 7. Sample code blocks for RESTful APIs.

`listSwitches()` method; the `/switches/:command` endpoint handles a PUT request by invoking the `updateSwitches()` method to turn on or off the switches. In our analysis of SmartThings apps, we found 23 official and 6 third-party web-service apps. These APIs might be used to transmit sensitive data to external services or receive undesired device commands from external services [19]. Turning to our example app, if an adversary compromises the forecast server and sends fake temperature values to the app, she can turn on many high-power devices to cause outages [91].

Language-inherited Operations. Analysis techniques need to address the challenges, which programming languages of IoT platforms pose, for analysis precision. In the following discussion, we will provide two examples from IoT apps developed with the Groovy language on the SmartThings platform: closures and call by reflection. We found in our corpus 37 official and 9 third-party SmartThings apps use closures; and 9 official apps and 1 third-party app use call by reflection. Closures are often used in SmartThings apps to loop through a list of devices and perform computation on each device. Listing 8 (lines 1–7) shows an example code block in which a closure is used to iterate through the `currSwitches` object to identify switches that are on. For analysis precision, tools need to analyze the structure of closures and inspect expressions within the closures, for example, to see how taints should be propagated in taint tracking [19].

Call by reflection is used to invoke a method by passing its name as a string. For instance, a method `foo()` can be invoked by declaring a string `name="foo"` requested from an external server through the `httpGet()` interface and thereafter called by reflection through `$name` (see Listing 8, lines 8–18). In another example, a developer defines a string conditioned on the state of a presence sensor and passes the string as an argument to a function call (see Listing 8, lines 19–36). To handle reflective calls, an analysis's call graph construction may add all methods in an app as possible call targets as a safe over-approximation [19]. For the example in Listing 8, an analysis may include both `foo()` and `bar()` methods into the targets of the call by reflection in the call graph of an app. Furthermore, an analysis may use string analysis to identify possible values of strings and refine the target sets of reflective calls.

5.4 Analysis Sensitivities

IoT app analysis can benefit from a more precise program analysis, such as context-sensitive analysis. We next present sensitivities an IoT source-code analysis might need for precision and motivate them through code examples. Although these examples are from SmartThings apps, the sensitivity issues are valid for all IoT programming platform apps, as many IoT platforms rely on general-purpose programming languages.

Flow Sensitivity. Flow sensitivity considers the order of execution in a program analysis [72]. Specifically, a flow-sensitive analysis accounts for variables whose contents change during

```
/* A code block of an app using closures */
def eventHandler(evt) {
    def currSwitches = switches.currentSwitch
    def onSwitches = currSwitches.findAll {
        switchVal -> switchVal == "on" ? true : false
    }
}
/* Reflection example 1 */
def getMethod() {
  httpGet("http://url") { resp ->
      if (resp.status == 200) {
          name = resp.data.toString()
      }
  }
  "$name"() // call by reflection
}
def foo() {...}
def bar() {...}
/* Reflection example 2 */
subscribe(presenceSensor, "present", presenceChanged)
subscribe(presenceSensor, "not present", presenceChanged)
def presenceChanged(evt) {
    def s
    if (evt.value == "not present") {
      s = "offDevices"
    } else {
      s = "onDevices"
    }
  performAction(s)
}
def performAction(String f) {
     $f() // call by reflection
}
def onDevices() { // turns on switches }
def offDevices() { // turns off switches }
def otherFunction() { // leak data or misuse device states }
```

Listing 8. Sample code blocks for language-inherited operations.

```
energy = powerMeter.currentValue
energy = developer_threshold
message = "energy consumption is $energy"
sendSMS(message, "attackerPhone")
```

Listing 9. An example code block for flow sensitivity.

program execution. In contrast, in a flow-insensitive analysis, a variable includes one qualifier abstracting the values that the variable gets during the entire program execution. In Listing 9, an example IoT app is presented. A flow-sensitive analysis would not flag it to have a data leak, because the message variable has a final value defined by the developer regardless of the sensitive value the power meter has (`powerMeter.currentValue`). However, a flow-insensitive analysis would flag it to leak sensitive data, because it determines that the current value of the power meter can be leaked when the ordering of assignments is not taken into account.

Context Sensitivity. Context-sensitive analyses span multiple procedures, considering a target function block within the context of the code calling it [88]. Specifically, if call-site contexts are used, only execution paths that are feasible by matching calls and returns are considered during analysis. In Listing 10, an analysis using depth-one call-site context sensitivity distinguishes the two call sites of `take_action` on lines 3 and 5. This means that the analysis analyzes `take_action` separately through arguments of "present" and "not_present" for those two call sites.

```
def presenceHandler(evt) {
    if (evt.value == "present") {
        take_actions("present")
    } else {
        take_actions("not present")
    }
}

def take_actions(evt_value) {
    if (evt.value == "present") {
        door.unlock(); lights.on()
        msg = "do not disturb please"
        sendSMS(msg, "userDefinedPhone")
    }
    if (evt.value == "not present") {
        door.lock(); lights.off();
        msg = "user left, event: $evt_value"
        sendSMS(msg, "attackerPhone")
    }
}
```

Listing 10. An example for illustrating context sensitivity.

```
subscribe(theSwitch, "switch.on", turnedOnHandler)
// initialize switchCounter and presenceCounter to 0
def turnedOnHandler() {
    s_threshold = 10
    state.presenceCounter = state.presenceCounter + 1
    p_counter = state.presenceCounter
    state.switchCounter = state.switchCounter + 1
    s_counter = state.switchCounter
    if (s_counter > s_threshold) {
        // invoke device actions
    }
    if (p_counter == 1) {
        // send text message
        state.presenceCounter = 0
    }
}
```

Listing 11. An example code block for field-sensitivity.

A context-sensitive analysis infers that, for the first call, there is no data leak, since `msg` is sent to a user-defined phone; yet, for the second call, a message is sent to an attacker's phone, which leaks information. In contrast, a context-insensitive analysis considers even infeasible paths in the control flow graph and would decide that both calls leak information. We found that depth-one call-site sensitivity in 230 analyzed apps was precise. Yet, more complex IoT apps might require contexts of greater depth.

Field Sensitivity. Field-insensitive analysis treats all fields in an object as equivalent [92]. IoT apps can use objects for various purposes; for example, SmartThings provides state objects (`state` and `atomicState`) as external storage to persist data across executions. State variables are often used in conditional branches to guard state transitions. In our analysis, we found 74 official and 34 third-party apps declare state variables. Listing 11 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. A field-insensitive system would not distinguish `presenceCounter` from `switchCounter` (indeed, the field insensitive analysis would not consider fields at all). A field-sensitive analysis is required to track all fields defined in the `state` and `atomicState` objects. For example, the switch-off device

```
1  input "ther", "capability.thermostat"
2  tempMax = 0
3  tempMin = 0
4
5  if (developerSetPoint < 65) {
6      tempMin = ther.currentValue
7  }
8  if (developerSetPoint > 65) {
9      tempMax = tempMin
10 }
11 message = "thermostat heating is set to: $tempMax"
12 sendSMS(message, "attackerPhone")
```

Listing 12. An example code block for path-sensitivity.

state is guarded by the predicate `state.switchCounter>10`. Furthermore, the analysis may label state variables in predicates as "state-variables," indicating they are stored in external data storage.

Path Sensitivity. Path sensitivity requires that the predicates at conditional branches are considered in a program analysis [72]. For instance, in Listing 12, the value of the sensitive information `ther.currentValue` never flows to the `message` variable, because the assignments `tempMin = ther.currentValue` and `tempMax = tempMin` never execute together in the program execution. A path-insensitive system, however, will conservatively analyze the impossible program execution "`tempMax = 0; tempMin = 0; tempMin = ther.currentValue; tempMax = tempMin; message = "thermostat... : $tempMax"`" in which the message string contains sensitive information due to an explicit flow from the thermostat state (i.e., `ther.currentValue`). One way of achieving path sensitivity is through predicate analysis. This is to track the predicates on a particular path during analysis. Take Listing 13 as an example. There are three feasible paths in `presentHandler`: (1) `userTemp=0` and `currentValue("power")<50` as the path condition of the path that returns constant value 68; (2) `userTemp=0` and `currentValue("power")≥50` as the path condition of the path that sends a text message, turns off the switch, and returns a constant 63; (3) `userTemp!=0` as the path condition of the path that returns `userTemp`.

Provenance tracking. It is often necessary for an analysis to track sources of data; for example, whether a piece of data is hard-coded by the developer or received as a user input. When such data is used in a device action, knowing its provenance can be extremely helpful in deciding whether the action is intended, by mistake, or even malicious. In the example of Listing 13, constants 63 and 68, and `threshold` are hard-coded by the developer, and as a result `x` is computed from hard-coded data by the developer; therefore, they should be labeled as "developer-defined." In some cases, a user of an application can define some data at install time. For instance, if the threshold value were entered by a user, then `x` would receive both the label "user-defined" and "developer-defined." In our analysis of 230 SmartThings apps, we found that apps mostly propagate a developer-defined constant or a user input to places that change device attributes. Occasionally, simple arithmetic is performed; for example, a user input is stored in `y`, followed by `x=y+10`, followed by changing a device attribute using `x`.

6 STUDY OF IOT ANALYSIS SYSTEMS

This section presents a study of six recent IoT analysis systems from the literature that use program-analysis techniques for security and privacy. Table 3 gives an overview of the systems. We begin by introducing analysis techniques used in these systems (Section 6.1). The systems, excluding FlowFence, use SmartThings apps for evaluation; thus, we present a background of

```
1  input "userTemp", "number", title: "Degrees", description: "Adjust temp or default is used by this many
       degrees", required: false, defaultValue:0
2  subscribe (presenceSensor, "present", presenceHandler)
3
4  def presentHandler() {
5      def threshold = 5
6      def x = threshold + evaluate(userTemp)
7      thermostat.setHeatingPoint(x)
8  }
9
10 def evaluate() {
11     if (userTemp == 0) {
12         if (currentValue("power")<50) {
13             return 68
14         } else {
15             sendSMS(userPhone, "power usage is high")
16             lightSwitch.off() // prevent high energy use
17             return 63
18         }
19     } else {
20         return userTemp
21     }
22 }
```

Listing 13. An example code block for predicate analysis and provenance tracking.

Table 3. A Summary of Studied IoT Analysis Systems

System	Purpose	Analysis method	Supplementary tech.	Analysis type	Analysis DS	IoT platform	Input gen.	# Apps
FlowFence [40]	Data leaks	Opacified comp.	–	Dynamic	Source code	–[1]	✓•	3
Saint [19]	Data leaks	Taint analysis	–	Static	AST	ST◇	n/a*	230[2]
ContexIoT [54]	Permission misuse	Code inst.	Taint analysis	Dynamic	AST	ST	✓	283[3]
SmartAuth [97]	Permission misuse	Code inst.[5]	NLP	Static[5]	AST	ST	⊖+	180[4]
ProvThings [102]	Data provenance	Code inst.	Program slicing	Dynamic	AST	ST	✓	236[3]
Soteria [20]	Abuse prevention	Symbolic exe.	Model checking	Static	AST	ST	n/a	65[2]

[1]Evaluates three existing IoT apps on Android OS. [2]Includes both official and third-party apps. [3]App type not specified.
[4]Includes only official apps. ◇ ST refers to the SmartThings IoT platform. * n/a, not applicable for a static system.
[5]SmartAuth extracts an app's behavior through static analysis; however, it also collects run-time information to block unauthorized device actions.
• FlowFence, ContexIoT, and ProvThings employ brute-force fuzzing that randomly generates user inputs and events to execute the apps.
+ ⊖ means that we could not find enough implementation details to be conclusive.

SmartThings apps (Section 6.2). Last, we study systems with regards to the issues we have introduced (Section 6.3). In particular, we contrast analysis types and practical implementation specifics.

IoT Systems. We give an overview of six recent IoT analysis systems studied throughout.

(1) *FlowFence* enforces sensitive data flow control in IoT apps and discloses intended data flow patterns to restrict the usage of sensitive data in IoT apps [40].
(2) *Saint* is a static taint analysis tool that finds sensitive data flows in IoT apps by tracking information flow from taint sources to taint sinks [19].

(3) *ContexIoT* is a context-based permission system that infers the app context automatically and enforces permissions based on that context [54].
(4) *SmartAuth* collects device information, annotations, and descriptions from app source to generate an authorization interface [97].
(5) *ProvThings* captures system-level provenance through security-sensitive APIs and leverages it for forensic reconstruction and attack investigation [102].
(6) *Soteria* extracts a state-model from an IoT app's source code for validating whether an app or multi-app environment adheres to safety, security, and functional properties [20].

6.1 Fundamental Analysis Techniques

We give an overview of analysis techniques used in six examined IoT analysis systems. Section 6.3 studies the systems with respect to these techniques.

Taint Tracking. Taint analysis begins by identifying sensitive data at a taint source with a label that shows the type of information. Taint tracking then starts from a taint source and propagates taint when tainted data is copied and deletes taint when all traces of tainted data are removed (e.g., when some variable is loaded with a constant). The impacted data is then flagged at a taint sink (often via the Internet or messaging interface) before it is sent out of the system. Last, the impacted data is investigated with malware detection tools or by human analysts to determine whether a leak actually constitutes a violation.

Code Instrumentation. Code instrumentation adds specific code to the source code of an app to collect the app's run-time behavior [70]. The code added during instrumentation is often called instrumented code. The instrumented code executes as part of the program's normal behavior, but it collects information necessary for some analysis such as context identification, attack detection, and attack reconstruction. Instrumenting every instruction of an app may incur high memory and performance overhead; thus, instrumentation aims to add the minimal code necessary for analysis.

Symbolic Execution. Symbolic execution indicates that an app is executed with symbolic value as an argument [13]. Unlike concrete execution, where the path is decided by the input, in symbolic execution, the app may practice any feasible path. Symbolic execution enables reasoning about an app behavior on many different inputs, which enables to discover infeasible paths, identify bugs and vulnerabilities, and create test inputs [86].

Model Checking. Model checking is used to analyze the correctness of software concerning some formally defined program property [53]. Systems or applications are first represented as finite state machines, and the execution of the software is validated against specified specifications through a generic model checker. The specifications are written in temporal logic formulas such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) [25].

Program Slicing. Program slicing is used to compute program slices that include the program parts affecting the values at some point of interest [105]. For example, the slice of a value at a statement includes a set of statements involved in computing the value in that statement. Program slicing can be used, among others, in debugging to capture the minimal program essentials and in information flow control to restrict trusted data from interacting with untrusted data.

Opacified Computing. Opacified computing provides sandboxes in places where an app has functions that access privacy-sensitive information or device states. Under this model, developers explicitly declare intended functions and a model is constructed to enforce access to the declared functions and prevent all others [40]. To achieve this, the developers split an app into modules

that operate on functions. The sandbox accumulates information from functions and returns the results that only respect the flow policies.

6.2 Analysis of SmartThings Apps

The analysis systems, excluding FlowFence, use SmartThings apps for evaluation. We provide a brief overview of SmartThings apps and present techniques for program analysis of its apps.

SmartThings Apps are developed with dynamic, object-oriented language Groovy in a sandboxed environment [19, 39]. The sandbox limits developers to a specific subset of the Groovy language for performance and security. For instance, the sandbox bans apps from creating their own classes and threads. The cloud back-end creates software wrappers for physical devices and runs the apps. SmartThings apps are executed within the SmartThings ecosystem, either in the hub or the SmartThings cloud. Users can install SmartThings apps from the market or proprietary system through SmartThings [20]. In the former, an app is published in the official market after the developer submits the app source code for review. Official apps appear in the market after a review process [20, 46]. In the latter, organizations can develop an app and make it accessible using the Web IDE. These apps are often shared in the SmartThings official community forum and do not receive any review process [20, 44].

Program Analysis of SmartThings Apps. Performing a program analysis from the source code of an app requires, among other things, building the app's Inter-procedural Control Flow Graph (ICFG) [19]. 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 perform analysis via the help of an analysis framework such as Soot [98]. However, we found that this approach may not be feasible due to the heavy use of reflection in the bytecode generated by the Groovy compiler [19]. In particular, the Groovy compiler translates direct method calls into a call by reflection. IoT systems often analyze Abstract Syntax Tree (AST) representations of Groovy source directly. The Groovy compiler supports customizing the compilation process by supporting compiler hooks, through which one can insert extra passes into the compiler. This is similar to the modular design of the LLVM compiler [60]. Therefore, systems often use `ASTTransformation` to hook into the compiler, `GroovyClassVisitor` to obtain the entry points, and the structure of the app and `GroovyCodeVisitor` to visit method calls and expressions inside AST nodes [19, 73].

6.3 Review of IoT Systems

We review the IoT analysis systems in light of the program-analysis issues developed in Section 5. We broadly split the systems into two groups based on their goals. The first group includes FlowFence and Saint for privacy, and the second group includes ProvThings, SmartAuth, ContexIoT, and Soteria for safety and security. Our review discusses issues in IoT that have been addressed by prior work and issues that remain open problems. We summarize the characteristics of the systems in Table 4. The following sections discuss the findings of this review process.

6.3.1 Systems for Privacy. We start our analysis with FlowFence and Saint for use (and potential avenues for misuse) of sensitive information in IoT apps. The main difference between FlowFence and Saint lies in the application of the taint tracking. FlowFence, a dynamic system, enforces intended data flow patterns through Quarantined Modules (QMs) whereas Saint, a static system, tracks data flow paths from taint sources to taint sinks. In FlowFence, a developer splits the source code of an app into QMs. QMs run on sensitive data in a sandbox. When a QM accesses sensitive information, taint from data sources (e.g., a photo taken by a camera) is tracked, and the data is passed to the sandboxed QM in the form of labeled and immutable data references called opaque

Table 4. Review of IoT Analysis Systems Based on Our Discussion in Section 5

	IoT-specific issues				IoT app idiosyncrasies			Analysis sensitivity				
System	**I.1**	**I.2**	**I.3**	**I.4**	**S.1**	**S.2**	**S.3**	**P.1**	**P.2**	**P.3**	**P.4**	**P.5**
FlowFence [40]	✓	✗	✓	✓	n/a†	n/a	n/a	✓	n/a	n/a	n/a	n/a
Saint [19]	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
ContexIoT [54]	✗	✗	✗	✓	✓	✓	✓	✓	n/a	✓	n/a	✓
SmartAuth [97]	✗	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗
ProvThings [102]	✓	✗	✗	✗	✓	✓	✓	✓	n/a	✓	n/a	✓
Soteria [20]	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓

Legend					
IoT-specific issues		**IoT app idiosyncrasies‡**	**Analysis sensitivity**		
I.1 Multi-app analysis	I.2 Trigger-action platform support	S.1 RESTful APIs⋆	P.1 Flow sensitivity	P.2 Context sensitivity	
I.3 Proactive defense	I.4 Run-time prompts	S.2 Closures and other operations	P.3 Field sensitivity	P.4 Path Sensitivity	
		S.3 Calls by reflection	P.5 Provenance Tracking		

‡We split the criterion of language-inherited operations of the SmartThings platform into "closures and other operations" and "call by reflection."

* n/a means not applicable. ⋆ RESTful APIs refer to web-service apps in SmartThings platform.

handles. Opaque handles can be dereferenced in a QM and transmitted out with a trusted sink API. The data sent through a sink must satisfy a flow policy such as `<camera, http>` defined in an app's manifest file. In contrast, Saint uses data flow analysis on IoT app source code to find sensitive data flows by tracking information flow from sensitive sources to external sinks. Saint's data flow analysis uses the app's IR. The IR models the app's lifecycle, including main methods of an app, devices, user inputs, and call graphs. By leveraging this IR, Saint prunes infeasible paths via path- and context-sensitivity through a work-list-based dependency algorithm.

The other difference between FlowFence and Saint is the implicit flows. The use of QMs in FlowFence eliminates the complexity of handling the implicit flows, because non-sensitive code cannot evaluate the value of an opaque handle (return value from a QM) unless it passes to a QM. In contrast, Saint tracks implicit flow by checking 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.

FlowFence and Saint also differ in addressing IoT-specific issues. Saint addresses SmartThings idiosyncrasies through on-demand algorithms for precision. Yet, for a call by reflection, Saint adds all methods in an app as possible call targets as a safe over-approximation. This increases the number of methods to be analyzed and may lead to over-tainting. FlowFence incurs over-tainting when an app is not accurately separated into QMs. The modulation depends on how a developer structures their data flow controls and IoT-specific mechanisms such as call by reflection. For instance, a developer that does not split an app into the least privilege QMs might cause over-tainting, because the analysis does not limit QMs to the code blocks that only process the sensitive data. Another point worth mentioning is that FlowFence can track sensitive data flows in multiple IoT apps by enforcing information flow policies between the IoT apps; however, Saint detects sensitive data flows within an individual app.

Last, FlowFence's taint tracking requires platform and app developers invest significant efforts towards extending their software to support information flow control, yet Saint automates information flow tracking through backward taint analysis. Both systems require users to make security decisions. FlowFence prompts users for confirmation with taint sources and sinks that indicate how an app will use sensitive data. This may cause frequent flow-prompts to request user permission if

publisher policies do not match with the policies. In contrast, Saint presents users with a warning report at install time. The report contains the full data flow paths between taint sources and sinks, including the taint labels and taint sink information such as hostname and contact information.

6.3.2 Systems for Safety and Security. We study SmartAuth, ContexIoT, ProvThings, and Soteria systems designed for safety and security. While these systems differ in analysis precision, run-time, and scope, all systems must be responsive to program-analysis issues.

All systems perform analysis on the AST of app source code. In detail, ContexIoT and ProvThings add instrumentation code to the app source code. Using instrumented code, ContexIoT determines the app functionality under a particular context; ProvThings logs app information for attack investigation and system diagnosis. We note that even though ContexIoT and ProvThings are dynamic systems, they use static analysis to determine where to insert code for obtaining the run-time behavior of apps. Soteria is a static analysis system that extracts a state-model from an app's source code to verify security and safety properties through a model checker. Last, SmartAuth performs static analysis to generate an authorization interface for users. SmartAuth complements the static analysis with Natural Language Processing (NLP) techniques to capture the differences between an app's actual functionality and the functionality a developer defines. NLP techniques are mainly used to gather data from developer-defined device code annotations and user inputs. For example, the device location is extracted from the device code block, and the app definition is obtained from the definition block of an app's source code. However, the application of NLP techniques might preclude the precise analysis of many practical scenarios. For instance, an app may have incorrect or incomplete device annotations, and some IoT platforms (e.g., OpenHAB [43]) do not require an app definition block that can be analyzed.

Systems implement different algorithms for analysis sensitivities depending on their goals. To obtain numerical-valued device attributes through provenance collection, ContexIoT implements taint analysis to find dependencies between numerical attributes. ProvThings computes a backward slice from a numerical-valued attribute as slicing criteria, and Soteria uses dependence analysis to identify a set of possible sources that a numerical-valued attribute can take. To obtain the predicates that guard device actions, ContexIoT gathers the value of the variables on which a device attribute is control-dependent. Soteria uses forward symbolic execution to perform path exploration on source code and accumulates path conditions during exploration. Systems, excluding Soteria, do not track the sources of the values in predicates that show whether a value is defined by a user, hard-coded by the developer, or that user input is modified by the developer. We note that labeling numerical-valued attributes and components in predicates may provide the user with more information for context identification and forensic analysis. For path-sensitivity, Soteria prunes infeasible paths by collecting the predicates at conditional branches and checking whether the conjunction of those predicates is always false. For context-sensitivity, it throws away paths that do not match function calls and returns using depth-one call-site sensitivity.

Systems also differ in handling IoT-specific issues. First, ContexIoT and SmartAuth analyze IoT apps in isolation—collecting context of an individual app; ProvThings and Soteria, however, capture interactions among apps. ProvThings supports this capability by analyzing provenance graphs of multiple apps, and Soteria constructs a union state-model that represents the unified behavior of apps when they are installed together. Second, systems address SmartThings-specific idiosyncrasies of Restful APIs, closures, and call by reflection in different ways. ContexIoT, ProvThings, and Soteria implement on-demand algorithms for idiosyncrasies; yet systems differ in handling call by reflection. Soteria constructs a call graph by adding all methods as possible call targets of a reflective call and may over-approximate the safety and security violations. ProvThings and ContexIoT instrument all reflective calls and may perform more instrumentation than needed.

Last, some IoT systems require users to make decisions. ContexIoT asks for a user approval of a context through run-time prompts before an action is executed. SmartAuth eliminates this limitation by presenting an authorization interface to users at install time. ProvThings requires users to investigate provenance graphs and create policies. Soteria defines a set of safety and security properties through requirements engineering.

7 TAKEAWAYS AND CONCLUSIONS

The security and privacy of IoT is a new and emergent area. This work studies IoT application security and privacy research through program-analysis techniques. We began by surveying five major IoT programming platforms to gain insights into the structure of their apps and map their app structures into common building blocks. By studying these IoT platforms, we have distilled the key aspects of program analysis under IoT-specific analysis issues, IoT app idiosyncrasies, and analysis sensitivities. Last, we have explored IoT app analysis academic papers over the past two years that employ program-analysis techniques for security and privacy issues. Broadly speaking, most attempts to date focus on issues such as sensitive data leaks, abuse prevention, permission misuse, and provenance collection. Our study yields a natural structure for reasoning about the capacity of the IoT systems and reveals the extent to which each system identifies and mitigates safety, security, and privacy issues.

Our key findings through these explorations include: (1) The dominant IoT programming platforms structure their apps around a sensor-computation-actuator idiom; (2) a suite of analysis tools and algorithms targeted at diverse IoT platforms is at this time largely absent; (3) because IoT applications control physical processes through devices, security and privacy issues are more subtle and difficult to identify than in related fields; (4) most approaches lack multiple analysis sensitivities such as path- and context-sensitivity; (5) most approaches often do not consider security and safety problems in multi-app environments and through information flows in trigger-action platforms; (6) members of the research community often use the SmartThings platform to evaluate their tools, as numerous open-source official and third-party apps are available; and (7) IoT systems often implement algorithms on the Abstract Syntax Tree (AST) of a SmartThings app because of the constraints on Groovy language and proprietary back-end libraries.

While the research community has been effective in providing tools that identify security and privacy issues in specific IoT implementations, many areas remain open problems, and IoT program analysis needs further progress before apps are safe for broader use:

- IoT analysis systems that use program analysis techniques for security and privacy often focus on smart homes. Yet, IoT environments are diverse in type and number of connected devices. Therefore, the analysis must be responsive to the unique characteristics and constraints of each different IoT domain.
- Current IoT analysis systems could encounter the same scalability concerns seen in other 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.
- 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. 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.
- Analysis systems often do not assess the impact of approaches on the system resources. Thus, existing IoT solutions may incur high computational cost and energy consumption

that might be infeasible for real systems. For instance, an IoT analysis system may need to poll sensor data periodically to obtain device states. The polling could consume sensor battery when the intervals are too short or may limit real-time detection when the intervals are too long. Program analysis and statistical modeling techniques can be combined to create efficient methods to reduce the energy consumption of devices.

- 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 ask 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.

We envision these explorations to be a central pillar for applying program-analysis techniques to IoT and providing researchers with insights useful for future work.

ACKNOWLEDGMENTS

The authors thank Xiaolei Wang, Dongrui Zeng, and Leonardo Babun for helpful discussions about this work.

REFERENCES

[1] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and A Selcuk Uluagac. 2018. Peek-a-Boo: I see your smart home activities, even encrypted! Retrieved from: *Arxiv Preprint:1808.02741*.

[2] SmartThings Inc. 2018. Samsung SmartThings add a little smartness to your things. Retrieved from: https://www.smartthings.com/.

[3] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele et al. 2015. FIT IoT-LAB: A large-scale open experimental IoT testbed. In *Proceedings of the 2nd IEEE World Forum on Internet of Things (WF-IoT'15)*.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques*. Addison Wesley.

[5] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. 2019. SoK: Security evaluation of home-based IoT deployments. In *IEEE Symposium on Security and Privacy (SP'19)*.

[6] Android Things. 2018. Retrieved from: https://developer.android.com/things/.

[7] IFTTT Santa Detector App. 2018. Retrieved from: https://ifttt.com/applets/170037p-santa-detector.

[8] Apple's HomeKit. 2018. Retrieved from: https://www.apple.com/ios/home/.

[9] Apple's HomeKit App Market. 2018. Retrieved from: https://support.apple.com/en-us/HT204893.

[10] Android Things Official Apps. 2018. Retrieved from: https://github.com/androidthings.

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 49, 6 (2014).

[12] Leonardo Babun, Amit Kumar Sikder, Abbas Acar, and A. Selcuk Uluagac. 2018. IoTDots: A Digital Forensics Framework for Smart Environments. Retrieved from: arXiv:arXiv:1809.00745.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.* 51, 3 (2018).

[14] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN Workshop on State of the Art in Java Program Analysis*.

[15] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If this then what? Controlling flows in IoT apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'18)*.

[16] Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM International Workshop on State of the Art in Java Program Analysis*.

[17] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the International Conference on Software Engineering*.
[18] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2016. CuriousDroid: Automated user interface interaction for Android application analysis sandboxes. In *Proceedings of the International Conference on Financial Cryptography and Data Security*.
[19] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive information tracking in commodity IoT. In *Proceedings of the USENIX Security Symposium*.
[20] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT safety and security analysis. In *Proceedings of the USENIX Technical Conference (USENIX ATC'18)*.
[21] Z. Berkay Celik, Gang Tan, and Patrick McDaniel. 2019. IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*.
[22] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'18)*.
[23] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-app threats in smart homes: Categorization, detection and handling. Retrieved from: *Arxiv Preprint:1808.02125*.
[24] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet? Retrieved from: *Arxiv Preprint:1503.07217*.
[25] Edmund M. Clarke and E. Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logic of Programs*.
[26] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*.
[27] Paul Comitz and Aaron Kersch. 2016. Aviation analytics and the internet of things. In *Integrated Communications Navigation and Surveillance, 2016*.
[28] Gabriele D'Angelo, Stefano Ferretti, and Vittorio Ghini. 2016. Simulation of the internet of things. In *Proceedings of the IEEE International Conference on High Performance Computing & Simulation (HPCS'16)*.
[29] Tamara Denning, Tadayoshi Kohno, and Henry M. Levy. 2013. Computer security and the modern home. *ACM Commun.* 56, 1 (2013).
[30] Wenbo Ding and Hongxin Hu. 2018. On the safety of IoT device physical interaction control. In *Proceedings of the ACM Computer and Communications Security Conference (CCS'18)*.
[31] Android Sensor API Documentation. 2018. Retrieved from: https://developer.android.com/guide/topics/sensors/sensors_overview.html.
[32] Eclipse Kura Documentation. 2018. Retrieved from: http://eclipse.github.io/kura/.
[33] Google Fit Developer Documentation. 2018. Retrieved from: https://developers.google.com/fit/.
[34] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. 2012. Xbase: Implementing domain-specific languages for Java. In *ACM SIGPLAN Notices*, Vol. 48.
[35] Leverett Eireann, Richard Clayton, and Ross Anderson. 2017. Standardisation and certification of the internet of things. In *Proceedings of the Workshop on the Economics of Information Security (WEIS'17)*.
[36] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.* 32, 2 (2014).
[37] Michael D. Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *Proceedings of the Workshop on Dynamic Analysis*.
[38] UI/Application Exerciser. 2018. Retrieved from: https://developer.android.com/studio/test/monkey.
[39] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'16)*.
[40] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical data protection for emerging IoT application frameworks. In *Proceedings of the USENIX Security Symposium*.
[41] Earlence Fernandes, Amir Rahmati, Kevin Eykholt, and Atul Prakash. 2017. Internet of things security research: A rehash of old ideas or new intellectual challenges? *Proceedings of the IEEE Symposium on Security & Privacy* (S&P'17).
[42] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized action integrity for trigger-action IoT platforms. In *Proceedings of the Network and Distributed Systems Symposium (NDSS'18)*.
[43] OpenHAB: Open Source Automation Software for Home. 2018. Retrieved from: https://www.openhab.org/.
[44] SmartThings Community Forum for Third-party Apps. 2018. Retrieved from: https://community.smartthings.com/.

[45] B. Gu, X. Li, G. Li, A. C. Champion, Z. Chen, F. Qin, and D. Xuan. 2013. D2Taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'13)*.
[46] SmartThings Code Review Guidelines and Best Practices. 2018. Retrieved from: http://docs.smartthings.com/en/latest/code-review-guidelines.html.
[47] Son N. Han, Gyu Myoung Lee, Noel Crespi, Kyongwoo Heo, Nguyen Van Luong, Mihaela Brut, and Patrick Gatellier. 2014. Dpwsim: A simulation toolkit for IoT applications using devices profile for web services. In *Proceedings of the IEEE World Forum on Internet of Things (WF-IoT'14)*.
[48] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking access control and authentication for the home internet of things (IoT). In *Proceedings of the USENIX Security Symposium*.
[49] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. 2016. Smart locks: Lessons for securing commodity Internet of Things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*.
[50] IFTTT (if this then that). 2018. Retrieved from: https://ifttt.com/.
[51] PTC Industrial IoT. 2018. Retrieved from: https://www.ptc.com/en/about.
[52] Alex Jablokow. 2015. How the IoT helps keep oil and gas pipelines safe, PTC. Accessed on Feb. 15, 2019 from https://www.ptc.com/en/product-lifecycle-report/how-the-iot-helps-keep-oil-and-gas-pipelines-safe.
[53] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Comput. Surv.* 41, 4 (2009).
[54] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. 2017. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Proceedings of the Network and Distributed Systems Symposium (NDSS'17)*.
[55] Qi Jing, Athanasios V. Vasilakos, Jiafu Wan, Jingwei Lu, and Dechao Qiu. 2014. Security of the Internet of Things: Perspectives and challenges. *Wireless Netw.* 20, 8 (2014).
[56] Gabor Kecskemeti, Giuliano Casale, Devki Nandan Jha, Justin Lyon, and Rajiv Ranjan. 2017. Modelling and simulation challenges in internet of things. *IEEE Cloud Comput.* 4, 1 (2017).
[57] Richard Kirk. 2015. Cars of the future: The internet of things in the automotive industry. *Netw. Sec.* 2015, 9 (2015).
[58] Sylvain Kubler, Kary Främling, and Andrea Buda. 2015. A standardized approach to deal with firewall and mobility policies in the IoT. *Pervas. Mob. Comput.* 20 (2015). https://www.sciencedirect.com/science/article/pii/S1574119214001588.
[59] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot Framework for Java program analysis: A retrospective. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop*.
[60] Chris Lattner. 2012. *LLVM Compiler Infrastructure Project*. The architecture of open source applications PTC. Accessed on Feb. 15, 2019 from https://www.aosabook.org/en/llvm.html.
[61] Maria Lazarte. 2016. Are we safe in the Internet of Things? *International Organization for Standardization* (September 2016). Retrieved from: https://www.iso.org/news/2016/09/Ref2113.html.
[62] Edward A. Lee, Mehrdad Niknami, Thierry S. Nouidui, and Michael Wetter. 2015. Modeling and simulating cyber-physical systems using CyPhySim. In *Proceedings of the International Conference on Embedded Software*.
[63] Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. 2017. FACT: Functionality-centric access control system for IoT programming frameworks. In *Proceedings of the Symposium on Access Control Models and Technologies*.
[64] Oded Leiba, Yechiav Yitzchak, Ron Bitton, Asaf Nadler, and Asaf Shabtai. 2018. Incentivized delivery network of IoT software updates based on trustless proof-of-distribution. Retrieved from: *Arxiv Preprint:1805.04282*.
[65] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *Proceedings of the International Conference on Compiler Construction*. Springer.
[66] Watson Android libraries for Android application analysis. 2018. Retrieved from: https://github.com/wala/WALA.
[67] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*.
[68] IFTTT Platform Size Metrics. 2018. Retrieved from: https://platform.ifttt.com/pricing.
[69] IoTBench A micro-benchmark suite to assess the effectiveness of tools designed for IoT apps. 2018. Retrieved from: https://github.com/IoTBench.
[70] Nicholas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation*. Technical Report. University of Cambridge, Computer Laboratory.
[71] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IoTSan: Fortifying the safety of IoT systems. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT'18)*.
[72] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2015. *Principles of Program Analysis*. Springer.

[73] GroovyCodeVisitor An Implementation of the Groovy Visitor Patterns. 2018. Retrieved from: http://docs.groovy-lang.org/docs.
[74] Temitope Oluwafemi, Tadayoshi Kohno, Sidhant Gupta, and Shwetak Patel. 2013. Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security. In *Proceedings of the USENIX LASER Workshop*.
[75] Mike Orcutt. 2016. Security experts warn congress that the internet of things could kill people. *MIT Technol. Rev.* (2016). Accessed on Feb. 15, 2019 from https://www.technologyreview.com/s/603015/security-experts-warn-congress-that-the-internet-of-things-could-kill-people.
[76] OpenHAB IoT App Market (Eclipse Market Place). 2018. Retrieved from: https://github.com/openhab/openhab1-addons/wiki/Samples-Rules.
[77] OpenHAB IoT App Market (Eclipse Market Place). 2018. Retrieved from: http://docs.openhab.org/eclipseiotmarket.
[78] Microsoft Flow Automate processes and tasks. 2018. Retrieved from: https://flow.microsoft.com/.
[79] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic security analysis of smartphone applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*.
[80] Partha Pratim Ray. 2016. A survey of IoT cloud platforms. *Fut. Comput. Inform. J.* 1, 1–2 (2016), 35–46.
[81] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife et al. 2016. *droid: Assessment and evaluation of Android application analysis tools. *ACM Comput. Surv.* 49, 3 (2016).
[82] SmartThings Official App Repository. 2018. Retrieved from: https://github.com/SmartThingsCommunity.
[83] Rodrigo Roman, Jianying Zhou, and Javier Lopez. 2013. On the features and challenges of security and privacy in distributed Internet of Things. *Comput. Netw.* 57, 10 (2013).
[84] E. Ronen and A. Shamir. 2016. Extended functionality attacks on IoT devices: The case of smart lights. In *Proceedings of the IEEE European Symposium on Security and Privacy (Euro S&P'16)*.
[85] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. 2017. IoT goes nuclear: Creating a ZigBee chain reaction. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'17)*.
[86] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'10)*.
[87] SmartThings Web service App Overview. 2017. Retrieved from: http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/overview.html.
[88] M. Sharir and A. Pnueli. 1981. *Two Approaches to Inter-procedural Dataflow Analysis*. Computer Science Department, New York University.
[89] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. 2015. Network-level security and privacy control for smart-home IoT devices. In *Proceedings of the International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob'15)*.
[90] SmartThings Official Developer Documentation. 2018. Retrieved from: http://docs.smartthings.com.
[91] Saleh Soltan, Prateek Mittal, and H. Vincent Poor. 2018. BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In *Proceedings of the USENIX Security Symposium*.
[92] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*. Springer, 196–232.
[93] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *Proceedings of the International Conference on World Wide Web*.
[94] Harriet Taylor. 2016. How the internet of things could be fatal. Retrieved from: *CNBC* (March 2016). https://www.cnbc.com/2016/03/04/how-the-internet-of-things-could-be-fatal.html.
[95] IoT Platform Comparison: How the 450 providers stack up. 2018. Retrieved from: https://iot-analytics.com/iot-platform-comparison-how-providers-stack-up/.
[96] The Internet of Things with AWS. 2018. Retrieved from: https://aws.amazon.com/iot/.
[97] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. 2017. SmartAuth: User-centered authorization for the internet of things. In *Proceedings of the USENIX Security Symposium*.
[98] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot: A Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*. IBM Press, 13 pages. http://dl.acm.org/citation.cfm?id=781995.782008.
[99] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N. Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. 2017. FarmBeats: An IoT platform for data-driven agriculture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*.
[100] G. Veerendra. 2016. *Hacking Internet of Things (IoT): A Case Study on DTH Vulnerabilities*. Technical Report. SecPod.

[101] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. 2014. A5: Automated analysis of adversarial Android applications. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*.

[102] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and logging in the internet of things. In *Proceedings of the Network and Distributed Systems Symposium (NDSS'18)*.

[103] Olivia Waxman. 2014. Stranger hacks into baby monitor and screams at child. *Time Magazine* (April 2014).

[104] SmartThings web-based simulator for testing SmartThings apps with virtual devices. 2018. Retrieved from: https://goo.gl/rfTB7e.

[105] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*. IEEE Press, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[106] Zapier Automate Workflows. 2018. Retrieved from: https://zapier.com/.

[107] Teng Xu, James B. Wendt, and Miodrag Potkonjak. 2014. Security of IoT systems: Design challenges and opportunities. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 417–423.

[108] Geng Yang, Li Xie, Matti Mäntysalo, Xiaolin Zhou, Zhibo Pang, Li Da Xu, Sharon Kao-Walter, Qiang Chen, and Li-Rong Zheng. 2014. A health-IoT platform based on the integration of intelligent packaging, unobtrusive bio-sensor, and intelligent medicine box. *IEEE Trans. Industr. Inform.* 10, 4 (2014).

[109] Apiant Connect your apps automate your business. 2018. Retrieved from: https://apiant.com/.

[110] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet of Things. In *Proceedings of the ACM Workshop on Hot Topics in Networks*.

[111] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. 2014. Internet of Things for smart cities. *IEEE Int. Things J.* 1, 1 (2014), 22–32.

[112] Bruno Bogaz Zarpelão, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, and Sean Carlisto de Alvarenga. 2017. A survey of intrusion detection in Internet of Things. *J. Netw. Comput. Appl.* 84 (2017).

[113] Nan Zhang, Soteris Demetriou, Xianghang Mi, Wenrui Diao, Kan Yuan, Peiyuan Zong, Feng Qian, XiaoFeng Wang, Kai Chen, Yuan Tian et al. 2017. Understanding IoT security through the data crystal ball: Where we are now and where we are going to be. Retrieved from: *Arxiv Preprint:1703.09809*.

[114] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Op. Syst. Rev.* 45, 1 (2011).

[115] Jan Henrik Ziegeldorf, Oscar Garcia Morchon, and Klaus Wehrle. 2014. Privacy in the Internet of Things: Threats and challenges. *Sec. Commun. Netw.* (2014).

Received November 2018; revised May 2019; accepted May 2019