Sensitive Information Tracking in Commodity IoT

Z. Berkay Celik, *The Pennsylvania State University;* Leonardo Babun, Amit Kumar Sikder, and Hidayet Aksu, *Florida International University;* Gang Tan and Patrick McDaniel, *The Pennsylvania State University;* A. Selcuk Uluagac, *Florida International University*

https://www.usenix.org/conference/usenixsecurity18/presentation/celik

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

978-1-939133-04-5

Sensitive Information Tracking in Commodity IoT

Z. Berkay Celik*[1], Leonardo Babun*[2], Amit K. Sikder[2], Hidayet Aksu[2],
Gang Tan[1], Patrick McDaniel[1], and A. Selcuk Uluagac[2]

[1] Systems and Internet Infrastructure Security Lab
Department of CSE, The Pennsylvania State University
{zbc102,gtan,mcdaniel}@cse.psu.edu
[2] Cyber-Physical Systems Security Lab
Department of ECE, Florida International University
{lbabu002,asikd003,haksu,suluagac}@fiu.edu

Abstract

Broadly defined as the Internet of Things (IoT), the growth of commodity devices that integrate physical processes with digital connectivity has had profound effects on society–smart homes, personal monitoring devices, enhanced manufacturing and other IoT applications have changed the way we live, play, and work. Yet extant IoT platforms provide few means of evaluating the use (and potential avenues for misuse) of sensitive information. Thus, consumers and organizations have little information to assess the security and privacy risks these devices present. In this paper, we present SAINT, a static taint analysis tool for IoT applications. SAINT operates in three phases; (a) translation of platform-specific IoT source code into an intermediate representation (IR), (b) identifying sensitive sources and sinks, and (c) performing static analysis to identify sensitive data flows. We evaluate SAINT on 230 SmartThings market apps and find 138 (60%) include sensitive data flows. In addition, we demonstrate SAINT on IOTBENCH, a novel open-source test suite containing 19 apps with 27 unique data leaks. Through this effort, we introduce a rigorously grounded framework for evaluating the use of sensitive information in IoT apps—and therein provide developers, markets, and consumers a means of identifying potential threats to security and privacy.

1 Introduction

The introduction of IoT devices into public and private spaces has changed the way we live. For example, home applications supporting smart locks, smart thermostats, smart switches, smart surveillance systems, and Internet-connected appliances change the way we monitor and interact with our living spaces. Here mobile phones become movable control panels for managing the environment that supports entertainment, cooking, and even sleeping. Such devices enable our living space to be more autonomous, adaptive, efficient, and convenient. However, IoT has also raised concerns about the privacy of these digitally augmented spaces [33, 10, 21, 17, 6]. These networked devices have access to data that can be intensely private, e.g., when you sleep, what your door lock pin code is, what you watch on TV or other media, and who and when others are in the house. Moreover, the state of the devices themselves represents potentially sensitive information.

Because IoT apps are exposed to a myriad of sensitive data from sensors and devices connected to the hub, one of the chief criticisms of modern IoT systems is that the existing commercial frameworks lack basic tools and services for analyzing what they do with that information–i.e., application privacy [47, 27, 46]. SmartThings [34], OpenHAB [31], Apple's HomeKit [1] provide guidelines and policies for regulating security [39, 30, 3], and related markets provide a degree of internal (hand) vetting of the applications prior to distribution [36, 4]. However, tools for evaluating privacy risks in IoT implementations is at this time largely non-existent. What is needed is a suite of analysis tools and techniques targeted to IoT platforms that can identify privacy concerns in IoT apps. This work seeks to explore formally grounded methods and tools for characterizing the use of sensitive data, and identifying the sensitive data flows in IoT implementations.

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

---

*contributed equally.

identify IoT-specific events/actions and asynchronously executing events, as well as platform-specific challenges such as call by reflection and the use of state variables. SAINT uses the IR to perform efficient static analysis that tracks information flow from sensitive sources to sinks.

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

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

We make the following contributions:

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

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

2 Problem Scope and Attacker Model

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

We focus on home automation platforms, which have the largest number of applications and consumer products [19]. Currently, SAINT is designed to analyze Smart-Things IoT apps written in the Groovy programming language. We evaluate the SmartThings platform for two reasons. First, it supports the largest number of devices (142) among all IoT platforms and provides apps of various functionalities [41]. Second, it has a detailed, publicly available documentation that helps validate our findings [40]. As we will detail in Sec. 4.1, SAINT exploits the highly-structured nature of the IoT programming platforms and extracts an abstract intermediate representation from the source code of an IoT app. This would allow the algorithms developed in SAINT to be easily integrated into other programming platforms written in different programming or domain-specific languages.

Attacker Model. SAINT detects sensitive data flows from taint sources to taint sinks caused by carelessness or malicious intent. We consider an attacker who provides a user with a malicious app that is used to leak sensitive information with or without permissions granted by the user. First, the granted permissions may violate user privacy by deviating from the functionality claimed by the app. Second, permissions granted by an IoT programming platform may also be used to leak information; for instance, permissions to access the hub id or the manufacturer name are often granted by default to develop device-specific solutions. We assume attackers cannot bypass the security measures of an IoT platform, nor can they exploit side channels [35]. For instance, an app that changes the light intensity to leak the information about whether anyone is at home is out of the scope of this work.

3 Background of IoT Platforms

We present background of the SmartThings IoT platform [40] to gain insights into the structure of its apps. We also discuss two other popular IoT platforms: openHAB [31] and Apple's HomeKit [1]. Our discussion is

based on a survey, which was performed by reviewing the platforms' official documentation, running their example IoT apps, and analyzing their app construction logic. We then present the challenges of information flow tracking in IoT apps. Lastly, we define each potential type of taint sources, the mechanisms for taint propagation, and taint sinks by studying their API documentation.

3.1 Overview of IoT Platforms

SmartThings is a proprietary platform developed by Samsung. The platform includes three components: a hub, apps, and the cloud backend [36]. The hub controls the communication between connected devices, the cloud backend, and mobile apps. Apps are developed with Groovy (a dynamic, object-oriented language) in a Kohsuke sandboxed environment [10]. 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 backend creates software wrappers for physical devices and runs the apps.

The permission system in SmartThings allows a developer to specify devices and user inputs required for an app at install time. User inputs are used to implement the app logic. For instance, a user input is used to set the heating point of a thermostat. Devices in SmartThings have capabilities (i.e., permissions). Capabilities are composed of *actions* and *events*. Actions represent how to control or actuate devices and events represent the state information of devices. Actions and events are not one to one. While a device may support many events, it may have limited actions. Apps are event-driven. They subscribe to device events or other pre-defined events such as clicking an icon; when an event is activated, the corresponding event handler is invoked to take actions.

Users can install SmartThings apps in two different ways using a smartphone companion app called Smart-Things Mobile. First, users may download apps through the official app market. Second, users may install third-party apps through the Web IDE on a proprietary cloud backend. Publishing an app in the official market requires the developer to submit the source code of the app for review. Official apps appear in the market after the completion of a review process that takes around two months to finish [36]. Users can also develop or install the source code of a third-party app and make it accessible to only themselves using the Web IDE. These apps do not require any review process and are often shared in the Smart-Things community forum [37]. Compared to other competing platforms, SmartThings supports more devices and has a growing number of official and third-party apps.

OpenHAB is a vendor- and technology-agnostic open-source automation platform built in the Eclipse IDE [31]. It includes various devices specifically designed for home automation. OpenHAB is open source and provides flexible and customizable device integration and applications

(referred to as rules) to build automated tasks. Similar to the SmartThings platform, the rules are implemented through three triggers to react to the changes in the environment. Event-based triggers listen to commands from devices; timing-based triggers respond to specific times (e.g., midnight); system-based triggers run with certain system events such as system start and shutdown. The rules are written in a Domain Specific Language (DSL) based on the Xbase language, which is similar to the Xtend language with some missing features [8]. Users can install OpenHAB apps by placing them in rules folder of their installations and from Eclipse IoT marketplace [29]. **Apple's HomeKit** is a development kit that manages and controls compatible smart devices [1]. The interaction between users and devices occurs through Siri and HomeKit apps. Similar to SmartThing and OpenHAB, each device has capabilities that represent what a device can do. Actions are defined to send commands to specific devices and triggers can be defined to execute actions based on location, device, and time events. Developers write scripts to specify a set of actions, triggers, and optional conditions to control HomeKit-compatible devices. Developing applications in HomeKit can either be written in Swift or Objective C. Users can install HomeKit apps using the Home mobile app provided by Apple [2].

3.2 Information Tracking in IoT Apps

Information flow tracking either statically or dynamically is a well-studied technique, which has been applied to many different settings such as mobile apps. From our study of the three IoT platforms, we found that IoT platforms possess a few unique characteristics and challenges in terms of tracking information flow when compared to other platforms. First, in the case of Android, it has a well-defined IR, and analysis can directly analyze IR code. However, IoT programming platforms are diverse, and each uses its own programming language. We propose a novel IR that captures the event-driven nature of IoT apps; it has the potential to accommodate many IoT platforms (Sec. 4.1). Second, while all taint tracking systems have to be configured with a set of taint sources and sinks, identifying taint sources and sinks in IoT apps is quite subtle, since they access a diverse set of devices, each of which has a different set of internal states. We describe common taint sources and sinks in IoT platforms to understand why they pose privacy risks (Sec. 3.3). Lastly, each IoT platform has its idiosyncrasies that can pose challenges to taint tracking. For instance, the SmartThings platform allows apps to perform call by reflection and allows webservice apps; each of these features makes taint tracking more difficult and requires special treatment (Sec. 4.2).

3.3 IoT Application Structure

From our studying of the three IoT platforms, we found that their apps share a common structure and common types of taint sources and sinks. In this subsection, we
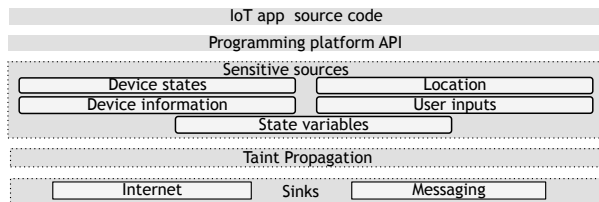
Figure 1: SAINT's source and sink categorization in IoT apps.

describe these common taint sources and taint sinks to understand why they pose privacy risks and how sensitive information gets propagated in their app structure (see Fig. 1). We present the taint sources and sinks of the SmartThings platform in Appendix C.

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

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

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

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

4) User Inputs. IoT apps often require user inputs either to manage the app logic or to control devices. In a simple example, a temperature value needs to be entered by a user at install time to set the heating point of a thermostat. User inputs are also often used to form predicates that control device actions; for instance, an app may turn off the switch of a device at a particular time entered by the user. Lastly, users may enter contact information to enable

notifications through messaging services when specific events occur. We mark such inputs as sensitive since they contain personally identifiable data and may be used to profile user behavior. We will discuss more about the semantics of user inputs in Sec. 6.

5) State Variables. IoT apps do not store data about their previous executions. To retrieve data across executions, platforms allow apps to persist data to some proprietary external storage and retrieve this data in later executions. For instance, a SmartThing app may persist a "counter" that keeps track of how many times a door is unlocked; during every execution of the app, the counter is retrieved from external storage and incremented when a door is unlocked. We call such persistent data app *state variables*. As we detail in Sec. 4.2.2, state variables store sensitive data and needs to be tracked during taint propagation.

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

During the execution of event handlers, it is necessary to track how sensitive information propagates in an app's logic. To obtain precision in taint propagation, we start from event handlers to propagate taint when tainted data is copied or used in computation, and we delete taint when all traces of tainted data are removed (e.g., when some variable is loaded with a constant). We will detail event handlers and SAINT's taint propagation logic in Sec. 4.

Taint Sinks. Our initial analysis also uses two taint sinks (although adding more later is a straightforward exercise).

1) Internet. IoT apps may send sensitive data to external services or may act as web services through which external entities acquire sensitive information. For the first kind, HTTP interfaces may be used to send out information. For instance, an app may connect to a weather forecasting service (e.g., www.weather.com) and send out its location information to get the local weather. For the second kind, a web-service IoT app may expose a URL that allows external entities to make requests to the app. For instance, a request from a remote server may be used to get the room temperature value. We will detail how SAINT tracks taint of web-service apps in Sec. 4.2.2.

2) Messaging Services. IoT apps use messaging APIs to deliver push notifications to mobile-app users and to send SMS messages to designated recipients when specific events occur. We consider all messaging service interfaces taint sinks–naturally, as they exfiltrate data by design.
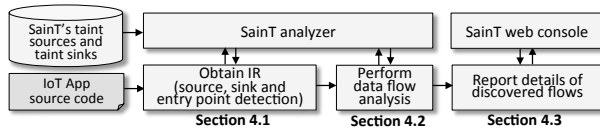
Figure 2: Overview of SAINT architecture.

4 SAINT

We present SAINT, a static taint analysis tool designed and implemented for SmartThings apps. Fig. 2 shows the overview of SAINT architecture. We implement the SAINT analyzer that extracts an intermediate representation (IR) from the source code of an IoT app. The IR is used to construct an app's entry points, event handlers, and call graphs (Sec. 4.1). Using these, SAINT models the lifecycle of an app and performs static taint analysis (Sec. 4.2). Finally, based on static taint analysis, it reports sensitive data flows from sources to sinks; for each data flow, the type of the sensitive information, as well as information about sinks, are reported (Sec. 4.3).

4.1 From Source Code to IR

The first step toward modeling the app lifecycle is to extract an IR from an app's source code. We exploit the highly-structured nature of IoT programming platforms based on our analysis in Sec. 3. We found that IoT systems are generally structured similarly regardless of their purpose and complexity. The dominant IoT platforms structure their app's design around the *sensor-computation-actuator* idioms. Therefore, we translate the source code of an IoT app into an IR by exploiting this structure.

SAINT builds the IR from a framework-agnostic component model, which is comprised of the building blocks of IoT apps, shown in Fig. 3. A broad investigation of existing IoT environments showed three types of common building blocks: (1) *Permissions* grant capabilities to devices used in an app; (2) *Events/Actions* reflect the association between events and actions (when an event is triggered, an associated action is performed); and (3) *Call graphs* represent the relationship between entry points and functions in an app. The IR has several benefits. First, it allows us to precisely model the app lifecycle as described above. Second, it is used to abstract away parts of the code that are not relevant to property analysis, e.g., definition blocks that specify app meta-data or logging code. Third, it allows us to have effective taint tracking, e.g., by associating permissions with the corresponding taint tags and by knowing what methods are entry points.

We use a sample app presented in Fig. 4 to illustrate the use of the IR. When a user arrives at home, the app unlocks the front door and turns on the lights. When she leaves, it turns off the lights, locks the front door, and sends to a security service a short message that she is away based on the time window specified by her.
Permissions. Permissions are granted when a user installs or updates an app. This is where various types of devices
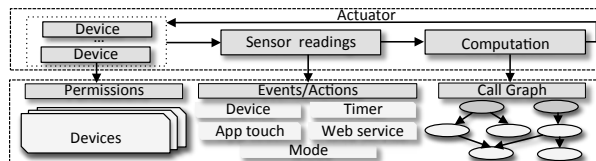

Figure 3: Components of the Intermediate Representation (IR).

```
 1: // Permissions block
 2: input (p, presenceSensor, type:device)
 3: input (s, switch, type:device)
 4: input (d, door, type:device)
 5: input (fromTime, time, type:user_defined)
 6: input (toTime, time, type:user_defined)
 7: input (c, contact, type:user_defined)
 8: // Events/Actions block
 9: subscribe(p, "present", h1)
10: subscribe(p, "not present", h2)
11: // Entry point
12: h1(){
13:       x()
14: }
15: // Entry point
16: h2(){
17:       s.off()
18:       d.lock()
19:       def between= y()
20:       if (between){
21:          z()
22:       }
23: }
24: x(){
25:       s.on()
26:       d.unlock()
27: }
28: y(){
29:       return timeOfDayIsBetween(fromTime, toTime,
30:                      new Date(), location.timeZone)
31: }
32: z(){
33:       sendSms(c, "...")
34: }
```

Figure 4: The IR of a sample app constructed by SAINT from the app's source code to demonstrate the precise modelling of the app's lifecycle. (Appendix A presents its source code.)

and user inputs are described and granted access. The permissions are read-only, and app logic is implemented using the permissions. The SAINT analyzer analyzes the source code of an app and extracts permissions for all devices and user inputs. Turning to the IR example in Fig. 4, the permission block (Lines 1-7) defines: (1) the devices: a presence sensor, a switch, and a door; and (2) user inputs: security-service "contact" information for sending notification messages, and "fromTime" and "toTime" values that are used to determine whether notification messages should be sent. For each permission, the IR declares a triple following keyword "input". For devices, the first two entries map device identifiers to their platform-specific device names in order to determine the interfaces that a device may access. For instance, an app that grants access to a switch may use `theswitchState` object to access its "on" or "off" state. For a user input, the line in the IR contains the string name that stores the user input and its type. The next entry labels the input with a taint tag showing the type of information such as the user-defined tag. As noted in Sec. 3.3, we consider user inputs sensitive.

We also include in the permission block a set of common interfaces designed for all apps that may leak sensitive data. For instance, `location.currentMode` gives the location mode either set to "home" or "away". We

assign each sensitive value to its label based on taint tags defined in Sec. 3.3. In this way, we obtain a complete list of sensitive interfaces an app may access.

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

We also found that events are not limited to device events, and can be generated in many other ways: (1) *Timer events*; event handlers are scheduled to take actions within a particular time or at pre-defined times (e.g., an event handler is invoked to take actions after a given number of minutes has elapsed or at specific times such as sunset); (2) *Web service events*; IoT programming platforms may allow an app to be accessible over the web. This allows external entities (e.g., If This Then That (IFTTT) [18]) to make requests to the app, and get information about or control end devices; (3) *App touch events*; for example, some action can be performed when the user clicks on a button in an app; (4) what actions get generated may also depend on *mode events*, which are behavior filters that automate device actions. For instance, an app running in "home" mode turns off the alarm and turns on the alarm when it is in the "away" mode. The SAINT analyzer analyzes all event subscriptions and finds their corresponding event handler methods; it creates a dummy main method for each entry point.

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

Call Graphs. We create a call graph for each entry point that defines an event-handler method. Turning to IR depicted in Fig. 4, we have two entry points h1() and h2()

Algorithm 1 Computing dependence from taint sinks

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

(Lines 12 and 16). h1() invokes x() to unlock the door and turn on the lights. The entry point h2() turns off the light and locks the door. It then calls method y() to check the time to decide whether to send a short message to a predefined contact via method z(). We note that the next section will detail how to construct call graphs, for example, in the case of call by reflection.

4.2 Static Taint Tracking

We start with backward taint tracking (Sec. 4.2.1). We then present algorithms to address platform- and language-specific taint-tracking challenges like state variables, call by reflection, web-service IoT apps, and Groovy-specific properties (Sec. 4.2.2). Last, we discuss the problem of implicit flows in static taint tracking (Sec. 4.2.3).

4.2.1 Backward Taint Tracking

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

In the first step, SAINT starts at the sinks of the ICFG and propagates taint backward. The reason that SAINT uses the backward approach is to reduce the processing overhead by starting from a few sinks instead of from a huge number of sensitive sources. This is confirmed by checking the ratio of sinks over sources in analyzed IoT apps (see Fig. 7 in Sec. 5 for taint source analysis and see Fig. 9 in Sec. 5 for taint sink analysis).

Algorithm 1 details the steps for computing a *dependence* relation that captures how values propagate in an app. It is a worklist-based algorithm. The worklist is initialized with identifiers that are used in the arguments of sink calls. Note that each identifier is also labeled with the node information to uniquely identify the use of an identifier because the same identifier can be used in multiple locations. The algorithm then takes an entry (n, id) from

Figure 5: Taint tracking under backward flow analysis.

the worklist and finds a definition for *id* on the ICFG; it adds identifiers on the right-hand side of the definition to the worklist; furthermore, the dependence between *id* and the right-hand side identifiers are recorded in *dep*. For ease of presentation, the algorithm treats parameter passing in a function call as inter-procedural definitions.

To illustrate, we use the code in Fig. 5 as an example. There is a sink call at place ❶. So the worklist is initialized to be ((23:phone), (23:t)); for illustration, we use line numbers instead of node information to label identifiers. Then, because of the function call at ❷, (16:temp_cel) is added to the worklist and the dependence (23:t, 16:[temp_cel]) is recorded in *dep*. With similar computation, the final output dependence relation for the example is as follows:

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

With the dependence relation computed and information about taint sources, SAINT can easily construct a set of possible data-leak paths from sources to sinks. For the example, since the threshold value thld is a user-input value (Lines 4 and 5 in Fig. 5), we get the following possible data-leak path: 5:thld to 16:temp_cel to 23:t.

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

4.2.2 SmartThings Idiosyncrasies

Our initial prototype implementation of SAINT was based on the taint tracking approach we discussed. However, SmartThings platform has a number of idiosyncrasies that

---
*Similar to how symbolic execution prunes paths via path conditions.

Listing 1: Sample code blocks for SmartThings idiosyncrasies

```
1  /* A code block of an app using a state variable */
2  def initialize() {
3      state.switchCounter = 0
4      subscribe(theswitch, "switch.on", turnedOnHandler)
5  }
6  def turnedOnHandler() {
7      state.switchCounter = state.switchCounter + 1
8      taintedVar = state.switchCounter // tainted
9  }
10 /* A code block of app using call by reflection */
11 def getMethod(){
12   httpGet("http://url"){
13     resp -> if(resp.status == 200){
14         methodName = resp.data.toString()
15       }
16     "$methodName"() //call by reflection
17 }
18 def foo() {...}
19 def bar() {...}
20 /* A code block of an example web-service app */
21 mappings {
22   path("/switches") {
23     action: [GET: "listSwitches"] }
24   path("/switches/:command") {
25     action: [PUT: "updateSwitches"] }
26 }
27 def listSwitches() {
28     switches.each {
29       resp << [name: it.displayName, value:
30           it.currentValue("switch")]} //tainted
31     return resp
32 }
33 def updateSwitches() {...}
34 /* A code block of an app using closures */
35 def someEventHandler(evt) {
36     def currSwitches = switches.currentSwitch //tainted
37     def onSwitches = currSwitches.findAll { //tainted
38       switchVal -> switchVal == "on" ? true : false
39     }
40 }
41 /* Implicit flows in an example app */
42 def batteryHandler(evt) {
43   def batLevel = event.device?.currentBattery;
44     if (batLevel < 25) {
45         switches.off()
46         def message = "battery low for device"
47         sendSMS(phone, message)
48     }
49 }
```

may cause imprecision in taint tracking. We next discuss how these issues are addressed in SAINT.

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

Call by Reflection. The Groovy language supports programming by reflection (using the GString feature) [38], which allows a method to be invoked by providing its name as a string. For example, a method foo() can be invoked by declaring a string name="foo" and thereafter called by reflection through $name; see Listing 1 (Lines 10–19) for another example. This can be exploited if an at-

```
saint-project.appspot.com

SainT Analysis Console

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

Analysis Result   Stacktrace

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

Actions   ≫Analyze SmartThings App · Reset Console · Publish This App · View Recent Apps   ☞ *IoT Test Suite* IoTBench
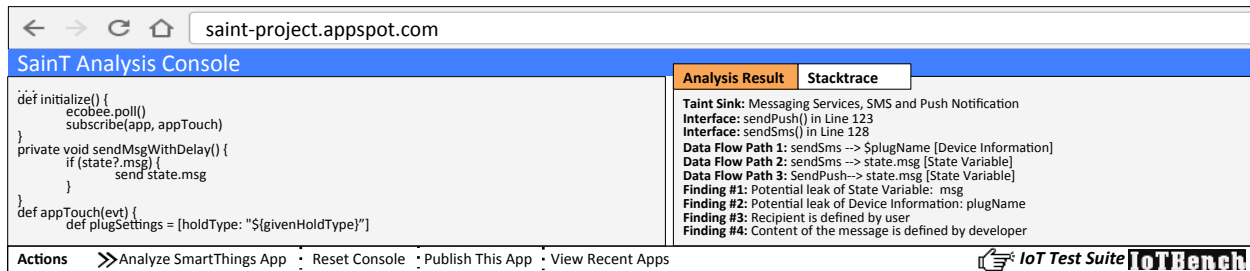
Figure 6: Our SAINT data flow analysis tool designed for IoT apps. The left region is the analysis frame, and the right region is the output of an example IoT app for a specific data flow evaluation.

tacker can control the string used in call by reflection [10], e.g., if the code has `name=httpGet(URL)` and the URL is read from an external server. While SmartThings does not recommend using reflective calls, our study found that ten apps in our corpus use this feature (see Sec. 5). To handle calls by reflection, SAINT's call graph construction adds all methods in an app as possible call targets, as a safe over-approximation. For the example in Listing 1, SAINT adds both `foo()` and `bar()` methods to the targets of the call by reflection in the call graph.

Web-service Applications. A web-service SmartThings app allows external entities to access smart devices and manage those devices. Such apps declare mappings relating *endpoints*, HTTP operations, and callback methods. Listing 1 (Lines 20–33) presents a code snippet of a real web-service app. The `/switches` endpoint handles an HTTP GET request that returns the state information of configured switches by calling the `listSwitches()` method; the `/switches/:command` endpoint handles a PUT request that invokes the `updateSwitches()` method to turn on or off the switches. The first prototype of SAINT did not flag the web-service apps for leaking sensitive data. However, our manual investigation showed that the web-service apps respond to HTTP GET, PUT, POST, and DELETE requests from external services and may leak sensitive data. To correct this, we modified the taint-tracking algorithm to analyze what call back methods are declared through the `mappings` declaration keyword [42]. Sensitive data leaked through those call back methods are then flagged by SAINT.

Closures and Groovy-Specific Operations. The Kohsuke sandbox enforced in SmartThings allows for closures and other Groovy-specific operations such as array insertions via $<<$. The SmartThings official developer guideline [40] imposes certain restrictions on these operations. For instance, closures are disallowed outside of methods. SAINT's implementation follows the guideline and imposes the same restrictions. For closures, we found that apps often loop through a list of devices and use a closure to perform computation on each device in the list. Listing 1 (Lines 34–40) shows an example in which a closure is used to iterate through the `currSwitches` object to identify those switches that

are turned on. For correct taint tracking, SAINT analyzes the structure of closures and inspects expressions in the closures to see how taints should be propagated.

4.2.3 Implicit Flows

An implicit flow occurs if the invocation of a sink interface is control dependent on a sensitive test used in a conditional branch. SAINT implements an algorithm designed to track implicit flows [23]. It checks the condition of a conditional branch and sees whether it depends on a tainted value. If so, it taints all elements in the conditional branch [26]. Listing 1 (Lines 41–49) presents an example app, in which an implicit flow happens because a `sendSMS()` call is control dependent on a test that involves sensitive data `batLevel`. We found that IoT apps often use tainted values in control flow dependencies. In our analysis, approximately two-thirds of analyzed apps implement device actions (such as unlocking a door) in branches whose tests are based on tainted values (such as a user's presence). We leave the detection of implicit flows optional in SAINT, and evaluate the impact of implicit flow tracking on false positives in Sec. 5.2.

4.3 Implementation

The IR construction from the source code of the input IoT app requires the building of the app's ICFG. SAINT's IR-building algorithm directly works on the Abstract Syntax Tree (AST) representation of Groovy code. The Groovy compiler supports customizing the compilation process by supporting compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [24]). The SAINT analyzer visits AST nodes at the compiler's semantic analysis phase where the Groovy compiler performs consistency and validity checks on the AST. Our implementation uses an `ASTTransformation` to hook into the compiler, `GroovyClassVisitor` to extract the entry points and the structure of the analyzed app, and `GroovyCodeVisitor` to extract method calls and expressions inside AST nodes [14]. This allows our implementation to use AST visitors to analyze expressions and statements, and get all necessary information to build IR.

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

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

Table 1: Applications grouped by permissions to taint sources and sinks. App functionality shows the diversity of studied apps.

SAINT's taint analysis also uses Groovy AST visitors. It extends the `ASTBrowser` class implemented in the Groovy Swing console, which allows a user to enter and run Groovy scripts [13]. The implementation hooks into the IR of an app in the console and dumps information to the `TreeNodeMaker` class; the information includes an AST node's children, parent, and all properties built at the pre-defined compilation phase. This allows us to acquire the full AST including the resolved classes, static imports, the scope of variables, method calls, and interfaces accessed in an app. SAINT then uses Groovy visitors to traverse IR's ICFG and performs taint tracking on it.

Output of SAINT. Fig. 6 presents the screenshot of SAINT's analysis result on a sample app. A warning report by SAINT contains the following information: (1) full data flow paths between taint sources and sinks, (2) the taint labels of sensitive data, and (3) taint sink information, including the hostname or URL, and contact information.

5 Application Study

This section reports our experience of applying SAINT on SmartThings apps to analyze how 230 IoT apps use privacy-sensitive data. Our study shows that approximately two-thirds of apps access a variety of sensitive sources, and 138 of them send sensitive data to taint sinks including the Internet and messaging channels. We also introduce an IoT-specific test suite called IOTBENCH [20]. The test suite includes 19 hand-crafted malicious apps that are designed to evaluate taint analysis tools such as SAINT. We next present our taint analysis results by focusing on several research questions:

RQ1 What are the potential taint sources whose data can be leaked? And, what are the potential taint sinks that can leak data? (Sec 5.1)

RQ2 What is the impact of implicit flow tracking on false positives? (Sec. 5.2)

RQ3 What is the accuracy of SAINT on IOTBENCH apps? (Sec. 5.3)

Experimental Setup. In late 2017, we obtained 168 *official* apps from the SmartThings GitHub repository [39] and 62 community-contributed *third-party* apps from the official SmartThings community forum [37]. Table 1 cate-

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

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

5.1 Data Flow Analysis

In this subsection, we report experimental results of tracking explicit "sensitive" data flows by SAINT in IoT apps (implicit flows are considered in Sec. 5.2). Table 2 summarizes data flows via Internet and messaging services reported by SAINT. It flagged 92 out of 168 official, and 46 out of 62 third-party apps have data flows from taint sources to taint sinks. We manually checked the data flows and verified that all reported ones are true positives. The manual checking process was straightforward to perform since the SmartThings apps are comparatively smaller than the apps found in other domains such as mobile phone apps. Finally, although user inputs and state variables may over-approximate sources of sensitive information, during manual checking, we made sure the reported data flows do include sensitive data.

SAINT labels each piece of flow information with the sink interface, the remote hostname, the URL if the sink

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

Table 2: Number of apps sending sensitive information through Internet and Messaging taint sinks.
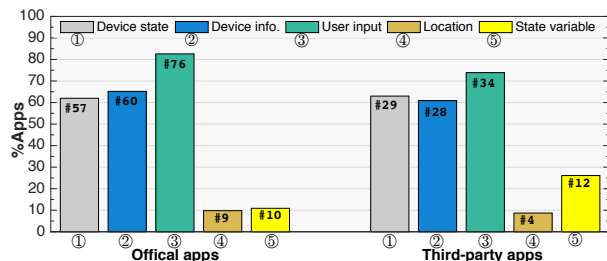

Figure 7: Percentages of apps sending sensitive data for specific kinds of taint sources. The absolute numbers of apps are also presented after the # symbol.

is the Internet, and contact information if the sink is a messaging service. In Table 2, the Internet column lists the number of apps that include only the taint source of the Internet. The Messaging column lists the number of apps that include only the taint source of some messaging service. 71.8% of the analyzed apps are configured to send an SMS message or a push notification. As shown in the table, 47.2% more apps include taint source in messaging services than the Internet. Finally, the Both column lists the number of apps (3.6% of apps) that includes a taint source through both the Internet and messaging services.

Taint Source Analysis. Fig. 7 shows the percentages of apps that have sensitive data flows of a specific kind of taint sources. To measure this, we used sensitive data's taint labels provided by SAINT, which precisely describe what sources the data comes from. More than half of the apps send user inputs, device states, and device information. Approximately, one-ninth of the apps expose location information and values in state variables. We found that 64 out of 92 official apps and 30 out of 46 third-party apps send multiple kinds of data (e.g., both device state and location information).

To better characterize the taint sources, we present the types of taint sources flagged by SAINT for apps that sends data in Table 3. There are 92 official apps that send sensitive data, marked with "O1" to "O92", and 46 third-party apps that send sensitive data, marked with "T1" to "T46". Out of 92 official apps, 28 apps (O1-O28) send one single kind of sensitive data, 16 apps (O29-O44) send two kinds of sensitive data, and the remaining 48 apps (O45-O92) send more than two and at most four kinds of sensitive data. Similar results are also identified for third-party apps. Our investigation suggests that apps at the top of the Table 3 implement simpler tasks such as managing motion-activated light switches; the apps at

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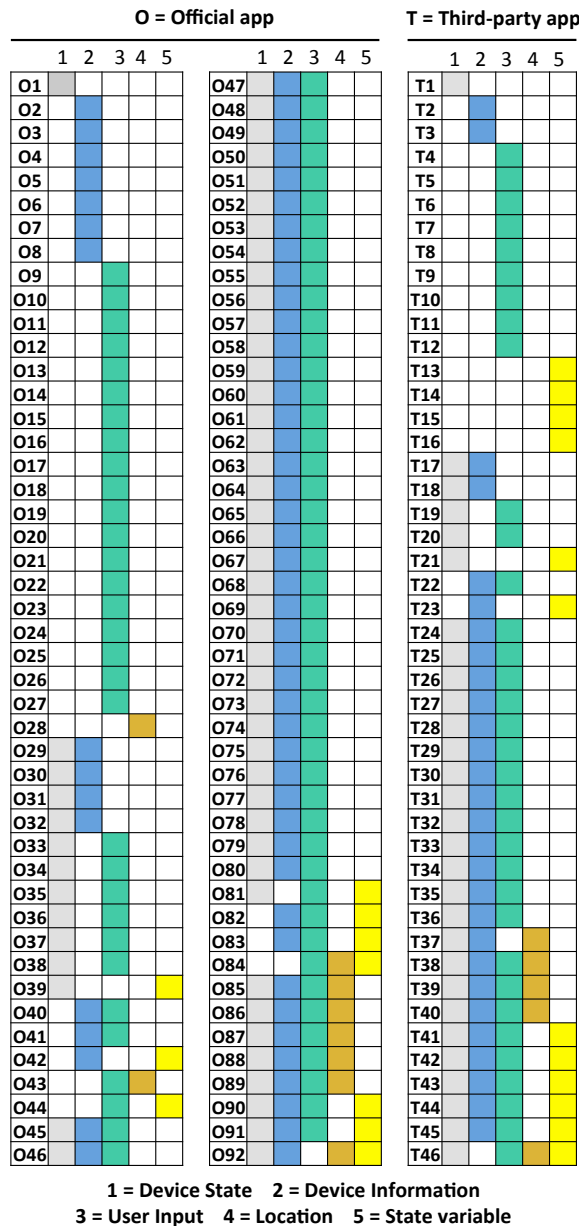3 = User Input 4 = Location 5 = State variable

Table 3: Data flow behavior of each official (O1-O92) and third-party (T1-T46) app. 43.2% of the official and 25.8% of the third-party apps do not send sensitive data (not shown).

the bottom tend to manage and control more devices to perform complex tasks such as automating many devices in a smart home. However, data flows depend on the functionality of the apps. For instance, a security and safety app managing few devices may send more types of sensitive data than an app designed for convenience that manages many devices.

In general, we found that there is no close relationship between the number of devices an app manages and the number of sensitive data flows. Fig. 8 shows the number of apps for each combination of device numbers and numbers of data flows. As an example, there are two apps that

Figure 8: The number of devices vs. the number of data flows based on taint labels in official (O) and third-party (T) apps. The numbers in the grids show the frequency of the apps.

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

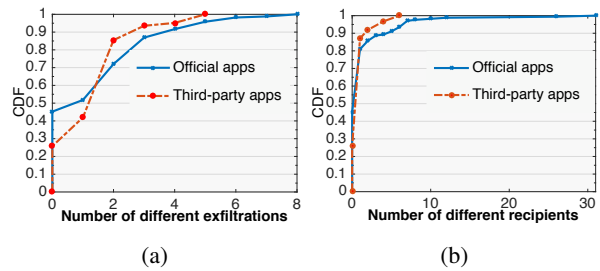|	(a)	(b)

Figure 9: Cumulative Distribution Function (CDF) of the number of different (a) sink interfaces and (b) recipients (contact information, remote hostname or URL) identified by SAINT.

manage seven devices and have four data flows. As shown in the figure, 15 official apps with a single device have three data flows, while an app with 16 devices has a single data flow. Similar results hold for third-party apps. Out of 46 third-party apps, 16 apps (T1-T16) have a single data flow, and the remaining 30 apps (T17-T46) have two to four data flows.

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

A vast majority of apps contain data flows through either a push notification or an SMS message or makes a few external requests to integrate external devices with SmartThings. Fig. 9a presents the CDF of the different sinks defined in official and third-party apps. Approximately, 90% of the official apps contain at most four, and 90% of the third-party apps contain at most three different invocations of sink interfaces (including apps that do not invoke sink interfaces). We also study the recipients at each taint sink reported in an app by SAINT. We first get the contact information for messaging, and hostname and URL for the Internet sinks. We then collect different contact addresses and URL paths to determine the recipients. Fig. 9b shows the CDF of the number of recipients defined in apps. The vast majority of apps involve a few recipients; they typically send SMS and push notifications to recipients. Approximately, 90% of the official apps have less than three sink recipients, and 90% of the third-party apps define at most two different recipients (including apps that do not implement taint sinks). A large number of recipients observed in official apps respond to external HTTP requests. For instance, a web-service app connects

to a user's devices, accesses their events and commands, and uses their state information to perform actions, and an app allows users to stream their device events to a remote server for data analysis and visualization. This leads to using a variety of taint sinks and URLs to access and manage various devices.

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

Table 4 presents the number of times a user, a developer, or an external party specifies the recipient and the content used in a data flow. The messaging rows of the table tell that, in official apps, users specify recipients 154 times, while contents are specified by users five times and 149 times by developers; for third-party apps, users define recipients 67 times, while message contents are specified by users five times, and 63 times by developers. In contrast, message contents are often hard-coded in the apps by developers. Table 4 shows a different story for Internet-sink calls. In this case, recipients and contents are often specified by developers and external services. An app in which recipients and contents of Internet-sink call are specified by external services is often a web-service app. As detailed in Sec. 4.2.2, web-service apps expose

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

Table 4: Recipient and content analysis of data flows.

endpoints and respond to requests from external services. These apps allow external services to access and manage devices. Additionally, in some apps, developers hard-code the recipients and contents of Internet communications to send information to external remote servers.

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

5.2 Implicit Flows

We repeated our experiments by turning on both explicit and implicit flows tracking. Approximately two-thirds of the apps invoke some sink interface that is control-dependent on sensitive tests. However and somewhat surprisingly, there are only six extra warnings produced when turning on implicit flows. The reason we found is that most of those sink calls already leak data through explicit flows. For example, in one app, x gets the state of a device x=currentState("device") and, when a user is present, x is sent out via an SMS message; even though there is an implicit flow (because sending the message depends on whether the user is present), there is also an explicit flow as the device information is sent out. The six extra warnings are all about sending out hard-coded strings: "Your mail has arrived!", "Your ride is here!", "No one has fed the dog", "Remember to take your medicine", "Potential intruder detected", and "Gun case has moved!". These messages contain information in themselves and are sent conditionally upon sensitive information; therefore, we believe information is indeed leaked in these

cases. We note that turning on implicit flow tracking increases the tracking overhead as more identifiers need to be tracked; however, based on the results, turning on implicit flow tracking on SmartThings IoT apps does not lead to an unmanageable number of false positives.

5.3 IoTBench

We introduce an IoT-specific test suite, IOTBENCH [20], an open repository for evaluating information leakage in IoT apps. We designed our test suite similar to those designed for mobile systems [5, 9] and the smart grid [25]; they have been widely adopted by the security community. IOTBENCH currently includes 19 hand-crafted malicious SmartThings apps that contain data leaks. Sixteen apps have a single data leak, and three have multiple data leaks; a total of 27 data leaks via either Internet and messaging service sinks. We crafted the IOTBENCH apps based on official and third-party apps. They include data leaks whose accurate identification through program analysis would require solving problems including multiple entry points, state variables, call by reflection, and field sensitivity. Each app in IOTBENCH also comes with ground truth of what data leaks are in the app; this is provided as comment blocks in the app's source code. IOTBENCH can be used to evaluate both static and dynamic taint analysis tools designed for SmartThings apps; it enables assessing a tool's accuracy and effectiveness through the ground truths included in the suite. We present three example SmartThings apps and their privacy violations in Appendix B. We made IOTBENCH publicly available:

```
https://github.com/IoTBench.
```

SAINT results on IOTBENCH. We next report the results of using SAINT on 19 IOTBENCH apps. In the discussion, we will use app IDs defined in Table 3 in Appendix B. SAINT produces false warnings for two apps that use call by reflection (Apps 6 and 7). These two apps invoke a method via a string. SAINT over-approximates the call graph by allowing the method invocation to target all methods in the app. Since one of the methods leaks the state of a door (locked or unlocked) to a malicious URL and the mode of a user (away or home) to a hard-coded phone number, SAINT produces warnings. However, it turns out that the data-leaking method would not be called by the reflective calls in those two apps. This pattern did not appear in the 230 real IoT apps we discussed earlier. SAINT did not report leaks for two apps that leak data via side channels (Apps 18 and 19). For example, in one app, a device operates in a specific pattern to leak information. As our threat model states, data leaks via side channels are out of the scope of SAINT and are not detected.

6 Limitations and Discussion

SAINT leaves detecting implicit flows optional. Even though our evaluation results on SmartThings apps show that tracking implicit flows does not lead to over-tainting

and false positives, whether this holds on apps of other IoT platforms and domains would need further investigation. Another limitation is SAINT's treatment of call by reflection. As discussed in Sec. 4, it constructs an imprecise call graph that allows a call by reflection target any method. This increases the number of methods to be analyzed and may lead to over-tainting. We plan to explore string analysis to statically identify possible values of strings and refine the target sets of calls by reflection.

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

Finally, SAINT's implementation and evaluation are purely based on the SmartThings programming platform designed for home automation. There are other IoT domains suitable for studying sensitive data flows, such as FarmBeats for agriculture [43], HealthSaaS for healthcare [16], and KaaIoT for the automobile [22]. We plan to extend SAINT's algorithms designed for SmartThings to these platforms and identify sensitive data flows.

7   Related Work

There has been an increasing amount of recent research exploring IoT security. These works centered on the security of emerging IoT programming platforms and IoT devices. For example, Fernandes et al. [10] identified design flaws in permission controls of SmartThings home apps and revealed the severe consequences of over-privileged devices. In another paper, Xu et al. [45] surveyed the security problems on IoT hardware design. Other efforts have explored vulnerability analysis within specific IoT devices [28, 17]. These works have found that apps can be easily exploited to gain unauthorized access to control devices and leak sensitive information of users and devices.

Many of previous efforts on taint analysis focus on the mobile-phone platform [9, 48, 15, 7, 5, 12]. These techniques are designed to address domain-specific challenges such as designing on-demand algorithms for context and object sensitivity. Several efforts on IoT analysis have focused on the security and correctness of IoT programs using a range of analyses. To restrict the usage of sensitive data, FlowFence [11, 32] enforces sensitive data flow control via opacified computation. ContexIoT [21] is a permission-based system that provides contextual integrity for IoT programs at runtime. ProvThings [44] captures system-level provenance through security-sensitive SmartThings APIs and leverages it for forensic reconstruction of a chain of events after an attack. In contrast, to our best knowledge, SAINT is the first system that precisely detects sensitive data flows in IoT apps by carefully identifying a complete set of taint sources and sinks, adequately modeling IoT-specific challenges, and addressing platform- and language- specific problems.

8   Conclusions

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

SAINT represents a potentially important step forward in IoT analysis, but further work is required. In future work, we will expand our analysis to support more platforms as well as refine our analysis for more complex and subtle properties. At a higher level, we will extend the kinds of analysis provided by the online systems and therein provide a suite of tools for developers and researchers to evaluate implementations and study the complex interactions between users and the IoT devices that they use to enhance their lives. Lastly, we will expand the IOTBENCH app suite. In particular, we are studying the space of privacy violations reported in academic papers, community forums, and from security reports, and will reproduce unique flow vectors in sample applications.

9   Acknowledgments

---

*SAINT is available at `http://saint-project.appspot.com`.

References

[1] APPLE'S HOMEKIT. https://www.apple.com/ios/home/. [Online; accessed 9-January-2018].

[2] APPLE'S HOMEKIT APP MARKET. https://support.apple.com/en-us/HT204893. [Online; accessed 9-January-2018].

[3] APPLE'S HOMEKIT SECURITY AND PRIVACY ON iOS. https://www.apple.com/business/docs/iOS_Security_Guide.pdf. [Online; accessed 9-January-2018].

[4] APPLE'S HOMEKIT SUBMISSION GUIDELINE. https://developer.apple.com/app-store/review/guidelines. [Online; accessed 9-January-2018].

[5] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices* (2014).

[6] CELIK, Z. B., MCDANIEL, P., AND TAN, G. Soteria: Automated IoT Safety and Security Analysis. In *USENIX ATC* (2018).

[7] CLAUSE, J., ET AL. Dytan: a Generic Dynamic Taint Analysis Framework. In *ACM Software Testing and Analysis* (2007).

[8] EFFTINGE, S., EYSHOLDT, M., KÖHNLEIN, J., ZARNEKOW, S., VON MASSOW, R., HASSELBRING, W., AND HANUS, M. Xbase: Implementing Domain-specific Languages for Java. In *ACM SIGPLAN Notices* (2012).

[9] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transaction on Computer Systems* (2014).

[10] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security Analysis of Emerging Smart Home Applications. In *IEEE Security and Privacy (SP)* (2016).

[11] FERNANDES, E., PAUPORE, J., RAHMATI, A., SIMIONATO, D., CONTI, M., AND PRAKASH, A. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security* (2016).

[12] GORDON, M. I., KIM, D., PERKINS, J. H., GILHAM, L., NGUYEN, N., AND RINARD, M. C. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS* (2015).

[13] GROOVY CONSOLE: THE GROOVY SWING CONSOLE. http://groovy-lang.org/groovyconsole.html. [Online; accessed 10-January-2018].

[14] GROOVYCODEVISITOR: AN IMPLEMENTATION OF THE GROOVY VISITOR PATTERNS. http://docs.groovy-lang.org/docs. [Online; accessed 10-January-2018].

[15] GU, B., LI, X., LI, G., CHAMPION, A. C., CHEN, Z., QIN, F., AND XUAN, D. D2Taint: Differentiated and Dynamic Information Flow Tracking on Smartphones for Numerous Data Sources. In *INFOCOM* (2013).

[16] HEALTHSAAS: THE INTERNET OF THINGS (IOT) PLATFORM FOR HEALTHCARE. https://www.healthsaas.net/. [Online; accessed 20-January-2018].

[17] HO, G., LEUNG, D., MISHRA, P., HOSSEINI, A., SONG, D., AND WAGNER, D. Smart Locks: Lessons for Securing Commodity Internet of Things Devices. In *ACM AsiaCCS* (2016).

[18] IFTTT (IF THIS, THEN THAT). https://ifttt.com/, 2017. [Online; accessed 11-January-2018].

[19] IOT PLATFORM COMPARISON. https://goo.gl/y8kzmY. [Online; accessed 29-January-2018].

[20] IOTBENCH: A MICRO-BENCHMARK SUITE TO ASSESS THE EFFECTIVENESS OF TOOLS DESIGNED FOR IOT APPS. https://github.com/IoTBench. [Online; accessed 29-April-2018].

[21] JIA, Y. J., CHEN, Q. A., WANG, S., RAHMATI, A., FERNANDES, E., MAO, Z. M., PRAKASH, A., AND UNVIERSITY, S. J. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS* (2017).

[22] KAAIOT: CONNECTED CAR AND IOT AUTOMOTIVE. https://www.kaaproject.org/automotive/. [Online; accessed 20-January-2018].

[23] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. Dta++: Dynamic Taint Analysis with Targeted Control-flow Propagation. In *NDSS* (2011).

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

[25] MCLAUGHLIN, S., AND MCDANIEL, P. SABOT: Specification-based Payload Generation for Programmable Logic Controllers. In *ACM CCS* (2012).

[26] MYERS, A. C. JFlow: Practical Mostly-static Information Flow Control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999).

[27] NAEINI, P. E., ET AL. Privacy expectations and preferences in an iot world. In *USENIX SOUPS* (2017).

[28] OLUWAFEMI, T., KOHNO, T., GUPTA, S., AND PATEL, S. Experimental Security Analyses of Non-Networked Compact Fluorescent Lamps: A Case Study of Home Automation Security. In *USENIX LASER* (2013).

[29] OPENHAB IOT APP MARKET (ECLIPSE MARKET PLACE). http://docs.openhab.org/eclipseiotmarket. [Online; accessed 9-January-2018].

[30] OPENHAB IOT APP SUBMISSION GUIDELINE. https://goo.gl/W63tEo. [Online; accessed 9-January-2018].

[31] OPENHAB: OPEN SOURCE AUTOMATION SOFTWARE FOR HOME. https://www.openhab.org/. [Online; accessed 9-January-2018].

[32] RAHMATI, A., FERNANDES, E., AND PRAKASH, A. Applying the Opacified Computation Model to Enforce Information Flow Policies in IoT Applications. In *IEEE Cybersecurity Development (SecDev)* (2016).

[33] RONEN, E., SHAMIR, A., WEINGARTEN, A.-O., AND O'FLYNN, C. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *IEEE Security and Privacy (SP)* (2017).

[34] SAMSUNG SMARTTHINGS. https://www.smartthings.com/. [Online; accessed 9-January-2018].

[35] SIKDER, A. K., AKSU, H., AND ULUAGAC, A. S. 6thSense: A Context-aware Sensor-based Attack Detector for Smart Devices. In *USENIX Security* (2017).

[36] SMARTTHINGS CODE REVIEW GUIDELINES AND BEST PRACTICES. http://docs.smartthings.com/en/latest/code-review-guidelines.html. [Online; accessed 29-January-2018].

[37] SMARTTHINGS COMMUNITY FORUM FOR THIRD-PARTY APPS. https://community.smartthings.com/. [Online; accessed 10-January-2018].

[38] SMARTTHINGS OFFICIAL API DOCUMENTATION. http://docs.smartthings.com/en/latest/ref-docs/reference.html. [Online; accessed 9-January-2018].

[39] SMARTTHINGS OFFICIAL APP REPOSITORY. https://github.com/SmartThingsCommunity. [Online; accessed 10-January-2018].

[40] SMARTTHINGS OFFICIAL DEVELOPER DOCUMENTATION. http://docs.smartthings.com. [Online; accessed 29-January-2018].

[41] SMARTTHINGS SUPPORTED IOT PRODUCTS (DEVICES). https://www.smartthings.com/products. [Online; accessed 29-January-2018].

[42] SMARTTHINGS WEB-SERVICE APP OVERVIEW. http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/overview.html, 2017. [Online; accessed 9-January-2018].

[43] VASISHT, D., KAPETANOVIC, Z., WON, J., JIN, X., CHANDRA, R., SINHA, S. N., KAPOOR, A., SUDARSHAN, M., AND STRATMAN, S. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *NSDI* (2017).

[44] WANG, Q., HASSAN, W. U., BATES, A., AND GUNTER, C. Fear and logging in the internet of things. In *NDSS* (2018).

[45] XU, T., WENDT, J. B., AND POTKONJAK, M. Security of IoT Systems: Design Challenges and Opportunities. In *IEEE Computer-Aided Design* (2014).

[46] YANG, Y., ET AL. A survey on Security and Privacy Issues in Internet-of-Things. *IEEE Internet of Things Journal* (2017).

[47] ZENG, E., MARE, S., AND ROESNER, F. End User Security & Privacy Concerns with Smart Homes. In *USENIX SOUPS* (2017).

[48] ZHU, D. Y., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *SIGOPS Operating Systems Review* (2011).

A   Source Code of the Example App

We present the Groovy source code of the home-automation app's IR shown in Figure 4, Sec. 4.

Listing 1: An example home-automation app

```
1  definition(
2      name: "SmartApp",
3      namespace: "mygithubusername",
4      author: "SainT",
5      description: "This is an app for home automation",
6      category: "My Apps",
7      iconUrl: "https://s3.amazonaws.com/smartapp-icons/
           Convenience/Cat-Convenience.png",
8      iconX2Url: "https://s3.amazonaws.com/smartapp-icons/
           Convenience/Cat-Convenience@2x.png",
9      iconX3Url: "https://s3.amazonaws.com/smartapp-icons/
           Convenience/Cat-Convenience@2x.png")
10
11 preferences {
12     section("When you are away/home") {
13         input "presenceSensor", "capability.presenceSensor",
               multiple: true,
14         required: true, title: "Which presence sensor?"
15     }
16
17     section("Turn on the lights") {
18         input "theSwitches", "capability.switch", required:
               true, multiple: true,
19         title: "Which lights?"
20     }
21
22     section("Lock/Unlock door") {
23         input "theDoor", "capability.door", multiple: false,
24         required: true, title: "Which door?"
25     }
26
27     section("Notify between what times?") {
28         input "fromTime", "time", title: "From", required: true
29         input "toTime", "time", title: "To", required: true
30     }
31
32     section("Send Notifications?") {
33         input("recipients", "contact", title: "Send
               notifications to") {
34             input "phone", "phone", title: "Warn security with
                   text message",
35             description: "Phone Number", required: true
36         }
37     }
38 }
39
40 def installed() {
41     initialize()
42 }
43
44 def updated() {
45     log.debug "Updated with settings: ${settings}"
46     unsubscribe()
47     initialize()
```

```
48 }
49
50 def initialize() {
51     log.debug "initialize configured"
52     subscribe(presenceSensor, "present", h1)
53     subscribe(presenceSensor, "not present", h2)
54 }
55
56 def h1(evt) {
57     log.debug "presence active called: $evt"
58     x()
59 }
60
61 def h2(){
62     log.debug "presence not active called: $evt"
63     theSwitches.off()
64     theDoor.unlock()
65
66     def between = y()
67     if (between){
68         z()
69     }
70
71     def currSwitches = theSwitches.currentSwitch
72     def onSwitches = currSwitches.findAll { switchVal ->
73         switchVal == "on" ? true : false
74     }
75     log.debug "${onSwitches.size()} out of ${switches.size
           ()} switches are on"
76 }
77
78 def x(){
79     theSwitches.on()
80     theDoor.unlock()
81     def currSwitches = theSwitches.currentSwitch
82     def onSwitches = currSwitches.findAll { switchVal ->
83         switchVal == "on" ? true : false
84     }
85     log.debug "${onSwitches.size()} out of ${theSwitches.
           size()} switches are on"
86 }
87
88 def y(){
89     log.debug "In time method"
90         return timeOfDayIsBetween(fromTime, toTime, new
               Date(), location.timeZone)
91 }
92
93 def z(){
94     log.debug "recipients configured: $recipients"
95     sendSms(phone, "The ${theDoor.displayName} is locked
           and the ${theSwitches.displayName} is off!")
96     def latestValue = theDoor.latestValue("door")
97     log.debug "message sent, the door status is
           $latestValue"
98 }
```

B   IoTBench Apps

Table 3 presents IOTBENCH apps categorized by their data leak ground-truth. We present three example apps and their privacy violations below.

Our first app "Implicit Permission 1" (ID: 11) sends a short message to household members when everyone is away. We update an existing legitimate app to include a code block that transmits the state of the door via the leak() method to a remote server (see Listing 2). A privacy violation occurs because the door state, which informs households are not at home, is leaked to the malicious server.

Listing 2: Device state leak through Internet interface

```
1  if (everyoneIsAway()){
2      //app logic
3      leak() // invoke when everyone is away
4  }
5  def leak() {
6      Params = [
7      uri: "https://malicious-url",
8      body: ["condition":"$thedoor.latestValue("door")"]]
9      httpPost(Params) // leak
10 }
```

The second app "Explicit-Implicit" (ID: 14) sends a short message to users when a door lock has a low battery. A code block is added to an existing app to send the battery level (implicit permission) and hub id (explicit permission) to a third-party's phone number via `sendSms()` when the `sms_send` variable is true (see Listing 3). Here, `sms_send` is tainted via the `state` object's SMS field. The leaked battery level is a privacy violation.

Listing 3: Leak of battery level and hub ID

```
1  def BatteryPowerHandler(evt) {
2      sms_send = state.SMS // true
3      msg = "$doorBattery.currentValue("battery")
4              power is out in hub ${evt.hubId}!"
5      sendPush(msg) // user gets a push notification
6
7      if (sms_send) { // attacker gets the same message
8          sendSms(attacker_phone, msg) // leak
9      }
10 }
```

Our final example is the "Call by Reflection 1" app (ID: 5). The app is used to trigger the alarm when smoke is detected. This app obtains the method name string from a remote server and uses this string to invoke `$state.method` (see Listing 4). Thus, the `updateApp()` method can be called by reflection. Because SAINT adds all methods in an app as possible call targets, it detects a data leak in `updateApp()`, which disables alarm by unsubscribing the "smoke-detected" event and sends this information to a hardcoded phone number.

Listing 4: Leak via a reflective call

```
1  def attack(){
2    httpGet("http://maliciousServer.com"){
3      resp ->
4          if(resp.status == 200){
5              state.method = resp.data.toString()
6          }
7      "$state.method"() // reflective call
8  }
9  updateApp() {
10     unsubscribe() // revoke smoke detector events
11     sendSms(attacker_phone,"$detector is revoked")
12 }
```

C Taint Source and Taint Sink APIs

We present SmartThings APIs that are taint sinks in Table 1 and APIs that are taint sources in Table 2. We refer the interested reader to SmartThings API documentation for the details [38]. For taint sinks, SmartThings recently announced asynchronous HTTP requests available as a beta development feature [40]. However, the analyzed apps do not use asynchronous HTTP APIs; thus we exclude them from the list. We note that some taint-source APIs are used together with the device names assigned by the developer, or require specific device capabilities to use them. Therefore, the number of taint sources used in an app differs based on the app's context.

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

Table 1: SmartThings taint-sink APIs.

Name of the interface	Definition
Device Information	
capability.<device type or attribute>	Allows to abstract devices into their underlying capabilities
getManufacturerName()	Gets the manufacturer name of the device
getModelName()	Gets the model name of the device
getName()	Gets the internal name of the device, Hub, command, or attribute
getSupportedAttributes()	Gets the list of device attributes
getSupportedCommands()	Gets the list of device commands
hasAttribute()	Determines if the device has the specified attribute
hasCapability()	Determines if the device supports the specified capability
hasCommand()	Determines if the device has the specified command name
latestValue()	Gets the latest reported value for the specified attribute
getFirmwareVersionString()	Gets the firmware version of the Hub device
getId()	The unique system identifier for the device or the Hub
getLocalIP()	The local IP address of the Hub device
getLocalSrvPortTCP()	The local server TCP port of the Hub device
getDataType()	Gets the data type of the device attribute
getValues()	Gets the possible values for the device attribute
getType()	Gets the type of the Hub device
getZigbeeId()	Gets the ZigBee ID of the Hub
getZigbeeEui()	Gets the ZigBee Extended Unique Identifier of the Hub
events()	Gets a list of events for the Device in reverse chronological order
eventsBetween()	Gets a list of events between the specified start and end dates
eventsSince()	Gets a list of events since the specified date
getCapabilities()	The list of capabilities provided by this Device
getDeviceNetworkId()	Gets the device network ID for the device
getDisplayName()	The label of the device assigned by the user
getHub()	The Hub associated with this device
getLabel()	The name of the device in the mobile application or Web IDE
getLastActivity()	The date of the last event from the device
getManufacturerName()	Gets the manufacturer name of the device
getModelName()	Gets the model name of the device
deviceName.capabilities	Gets the device capabilities
getTypeName()	The type of the device
Location	
getContactBookEnabled()	Determine if the Location has Contact Book enabled
getCurrentMode()	Gets the current mode for the location
getId()	Gets the unique internal system identifier for the location
getHubs()	Gets the list of Hubs for the location
getLatitude()	Gets the geographical latitude of the location
getLongitude()	Gets the geographical longitude of the location
getMode()	Gets the current mode name for the location
setMode()	Sets the mode for the location
getTimeZone()	Gets the time zone for the location
getZipCode()	Gets the ZIP code for the location
getLocationId()	The unique identifier for the location associated with the event
getLocation()	The Location associated with the event

Name of the interface	Definition
Device State	
latestState()	Gets the latest Device State record for the specified attribute
statesSince()	Gets a list of Device State since the date specified
getArguments()	Gets the list of argument types for the command
getDateValue()	Gets the value of the event as a Date object
getDescriptionText()	Gets the description of the event
getDoubleValue()	Gets the value of the event as a Double
getFloatValue()	Gets the value of the as a Float
getIntegerValue()	Returns the value of the event as an Integer
getJsonValue()	Gets the value of the event as a parsed JSON
getLastUpdated()	Gets the last time the event was updated
getLongValue()	Gets the value of the event as a Long
getName()	Gets the name of the event
getNumberValue()	Gets the value of the event as a number
getNumericValue()	Gets the value of the event as a number
getUnit()	Gets the unit of measure for the event
getValue()	Gets the value of the event as a String
getData()	Gets a map of any additional data on the event
getDate()	Acquisition time of the device state record
getDescription()	The raw description that generated the event
getDevice()	Gets the device associated with the event
getDisplayName()	Gets the user-friendly name of the source of the event
getDeviceId()	Unique identifier of the Device associated with the event
getIsoDate()	Acquisition time of the event as an ISO-8601 String
getSource()	The source of the event
getXyzValue()	Value of the event as a 3-entry Map
isPhysical()	TRUE if the event is from a physical actuation of the device
isStateChange()	TRUE if the attribute value for the event has changed
isDigital()	TRUE if the event is from a digital actuation of the device
currentState()	Gets the latest State for the specified attribute
currentValue()	Gets the latest reported values of the specified attribute
getStatus()	Gets the current status of the device
User Inputs	
input "someSwitch", "capability.switch"	User preferences for the devices (accessed as $someSwitch)
input "someMessag", "text"	User preferences for message (accessed as $someMessage)
input "someTime", "time"	User preferences for the time (accessed as $someTime)
input "someTime", "time"	User preferences for the time (accessed as $someTime)
input "minutes", "time"	User preferences for time span (accessed as $minutes)
State Variables	
state	Defines the state variable state
atomicState	Defines the state variable atomicState

Table 2: SmartThings taint-source APIs. The complete list can be accessed in our project page [20].

App Category	ID/App Name	App Description‡	Results†
Lifecycle	1- Multiple Entry Points 1	The app stores different sensitive data under the same variable name in different functions and only one of them is leaked.	✔
	2- Multiple Entry Points 2	The app stores different sensitive data under the same variable name in different functions and more than one piece of data is leaked.	✔
Field Sensitivity	3- State Variable 1	A state variable in the `state` object's field stores sensitive data. It is used in different functions and leaked through various sinks.	✔
Closure	4- Leaking via Closure	A variable is tainted with the use of closures. The sensitive data is then leaked via different sinks.	✔
Reflection	5- Call by Reflection 1	A string is requested via Http Get interface and the string is used to invoke a method. One of the app methods leaks device information.	O
	6- Call by Reflection 2	A string is used to invoke a method via call by reflection. A method leaks the state of a door.	X
	7- Call by Reflection 3	A string is used to invoke a method via call by reflection. A method leaks the mode of a user.	X
Device Objects	8- Multiple Devices 1	Various sensitive data is obtained from different devices and leaked via different sinks.	✔
	9- Multiple Devices 2	Sensitive data from various devices is tainted and leaked via different sinks.	✔
	10- Multiple Devices 3	A taint source is obtained from device state and device information and they are leaked via messaging services.	✔
Permissions	11- Implicit 1	A malicious URL is hard-coded, and device states (implicit permission) are leaked to the hard-coded URL.	✔
	12- Implicit 2	A hard-coded phone number leaks the user inputs (implicit permission).	✔
	13- Explicit	The hub ID (explicit permission) and state variables are leaked to a hard-coded phone number.	✔
	14- Explicit-Implicit	A phone number is hard-coded to leak device information (implicit permission) and hub id (explicit permission).	✔
Multiple Leaks	15- Multiple Leaks 1	Various sensitive data obtained from the state of the devices and user inputs are leaked via same sink interface.	✔
	16- Multiple Leaks 2	Various sensitive data is obtained from device states and user inputs, and they are leaked via the Internet and messaging sinks.	✔
	17- Multiple Leaks 3	Various sensitive data is obtained from state variables and devices, and they are leaked via more than one hard-coded contact information.	✔
Side Channel	18- Side Channel 1	A device is misused to leak information (e.g., turning on/-turning off a light to signal adversary).	!
	19- Side Channel 2	A device operating in a specific pattern causes other connected devices to trigger malicious activities.	!

Table 3: Description of IOTBENCH test suite apps and SAINT's results.

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

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