Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling

Haotian Chi*, Qiang Zeng†, Xiaojiang Du*, Jiaping Yu*
*Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA
†Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208, USA
Email: {htchi, dux, jiaping.yu}@temple.edu, Zeng1@cse.sc.edu

***Abstract*—A number of Internet of Things (IoTs) platforms have emerged to enable various IoT apps developed by third-party developers to automate smart homes. Prior research mostly concerns the overprivilege problem in the permission model. Our work, however, reveals that even IoT apps that follow the principle of least privilege, when they *interplay*, can cause unique types of threats, named *Cross-App Interference* (CAI) threats. We describe and categorize the new threats, showing that unexpected automation, security and privacy issues may be caused by such threats, which cannot be handled by existing IoT security mechanisms. To address this problem, we present HOMEGUARD, a system for appified IoT platforms to detect and cope with CAI threats. A symbolic executor module is built to precisely extract the automation semantics from IoT apps. The semantics of different IoT apps are then considered collectively to evaluate their interplay and discover CAI threats systematically. A user interface is presented to users during IoT app installation, interpreting the discovered threats to help them make decisions. We evaluate HOMEGUARD via a proof-of-concept implementation on Samsung's SmartThings and discover many threat instances among apps in the SmartThings public repository. The evaluation shows that it is precise, effective and efficient.**

I. INTRODUCTION

The rapid proliferation of Internet-of-Things (IoTs) has advanced the development of smart homes to a new era where emerging centralized and appified (a.k.a., app-powered) smart home platforms connect heterogenous IoT devices and offer programming frameworks for third-party developers to contribute various home automation ideas. According to the report of German IoT market research firm IoT Analytics in July 2017, the number of IoT platforms on the market has grown from 360 to 450 over the past 12 months and smart home platforms account for 21 percent [1]. Representative examples of such platforms are Samsung SmartThings [7], Apple HomeKit [5], and Google Android Things [4]. By installing IoT apps on a platform, users can monitor, remotely control, and automate their IoT devices to make smarter homes. However, appified IoT platforms also introduce new app-level attack surfaces, which can be exploited by attackers or misused by homeowners, introducing new challenges in security and privacy. For example, burglars can stealthily open a smart door lock via an IoT app to break into homes, which is impossible in non-appified IoT systems.

Fernandes et al. [22] discover design flaws such as the *overprivilege* problem in Samsung's SmartThings, one of the most mature smart home platforms; they demonstrate that malicious apps can be constructed to expose smart homes to severe attacks that exploit the overprivilege problem. Thus, some systems are proposed to handle the problem. ContexIoT [29] proposes a context-based permission system to involve users into making decisions on whether a security-critical command in an IoT app during runtime should proceed under the current context. SmartAuth [46] compares the analysis result of the app code with the code annotations and app description to identify overprivilege in an app, and allows users to specify access control policies. Tyche [40] designs a risk-based permission model over the original model which groups all supported permission-defined commands into three risk levels, and allows users to specify a maximum risk level for any installed app. Thus, permissions greater than the specified level will be denied during the app execution. Instead of handling security threats proactively, ProvThings [47] presents a runtime logging system for forensics and diagnosis purposes.

This paper reveals that new types of threats, which we call *Cross-App Interference* (CAI) threats, can be caused by apps even if the permission system strictly follows the principle of least privilege. Hence, CAI threats do not depend on the overprivilege problem; when IoT apps installed at the same smart environment *interplay*, various CAI threats may arise.

First, when a smart home is installed with apps provided by different third-party developers, there is a chance that some apps may contain *conflicting* logics in terms of how to control an IoT device. For example, given a certain combination of sensor values, two apps issue opposite commands for opening/closing a smart door, which not only confuses homeowners but also causes seriously exploitable security issues.

Second, the action taken by a smart device, due to the execution of one app, may trigger a *chained* execution of another app, but not all such chained executions are desired by homeowners and some may even cause security or privacy issues. For example, assuming an app is to turn off the light when no motion is detected for a period of time, while another app is to open the curtain if the room is *too* dark during the daytime, then opening the curtain becomes a chained action due to the light-off action. Such chained actions may occur

An earlier version of this paper was submitted to ACM CCS'18 on May 9th, 2018. This version contains some minor modifications based on that submission.

only under specific circumstances; thus, they impose security and privacy threats that are hidden from users.

In addition to threats imposed by benign apps, attackers may construct CAI threats by including seemingly benign logic into a malicious app to successfully pass the code review. But the app is to interact with other apps to launch attacks, e.g., opening the smart door. That is, exploitation of such threats can become a *new attack vector* against smart environments.

Therefore, it is important and urgent to present, detect, and handle the new types of threats. The goal of this work is thus to (1) describe CAI threats and categorize them, (2) propose and implement a technique to automatically discover such threats, and (3) present the revealed threats to homeowners in a user-friendly way and allow them to make informed decisions in handling these threats.

Existing techniques cannot discover CAI threats, as they analyze apps individually and most of them focus on IoT app-level threats exploiting overly granted permissions, while CAI threats are essentially due to the intricate interplay between multiple apps. Hence, discovering the new type of threats calls for *cross-app-boundary* semantics-relation analysis (i.e., how the logic defined in one IoT app may interfere with that of another app), which is the main challenge of our work.

Our idea is to extract the automation semantics (also referred to as *rules*) from each app and then discover the threats by evaluating the interaction relations between the extracted semantics collectively and systematically. To precisely capture the semantics of an app, we propose to perform symbolic execution analysis on IoT apps. The semantics of each app is then represented as quantifier-free first-order formulas. Lastly, an automatic method based on constraint solver is applied for discovering CAI threats, which are then interpreted into a human-readable form and presented to homeowners.

Instead of requiring the repetitive intervention of users to handle such threats, we propose a deployment method that is much more user-friendly and easy to deploy. Whenever a new app is to be installed, our system interposes and analyzes whether there exist CAI threats between the new app and the already-installed ones. As a result, the homeowner only needs to spend one-time decision making for each app to be installed.

We develop a proof-of-concept prototype system HOMEGUARD on Samsung SmartThings, which at the time of research supports the largest number of IoT devices and IoT apps (i.e., SmartApps). Our evaluation shows that HOMEGUARD can precisely discover CAI threats from real-world SmartApps, and generate the analysis results instantly.

Our main contributions can be summarized as follows:

- We describe new types of threats in smart environments, *Cross-App Interference* threats, which do not depend on overprivileged apps. Attackers may exploit such threats by inserting seemingly benign logics so that malicious apps can pass the conventional code review but may interact with other apps in a harmful way. Besides, even benign apps, due to their interplay, may cause undesired or even dangerous consequences.

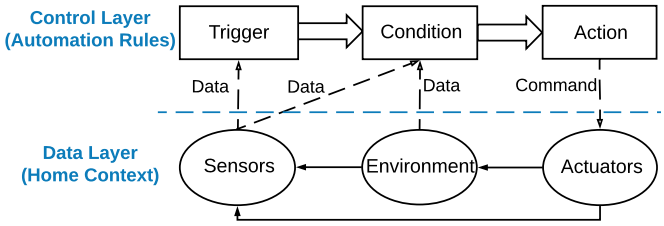


Fig. 1: The home automation model.

- We present a semantics-based user-friendly system HOMEGUARD that not only discovers CAI threats but also assists users to handle them. Our approach introduces zero-modification to the smart home platform architecture and only needs user intervention during app installation.
- We design and build a symbolic execution engine that can precisely extract automation semantics from SmartApps. To our knowledge, this is the first symbolic executor for SmartApp code.
- We evaluate the effectiveness and efficiency of HOMEGUARD. The evaluation shows that it can detect all of the categories of CAI threats precisely and efficiently. Our experiments discover and verify many threat instances from real-world apps in the SmartThings public repository.

II. BACKGROUND

We first describe the home automation model, and then introduce the popular cloud-backed smart home platforms.

A. Home Automation Model

A home automation model can be abstracted into the *data layer* and the *control layer*.

Data Layer. The data layer of a home is its *home context*, consisting of sensors, actuators, and the home environment. (1) A *sensor* in our model may be a traditional device or subsystem that measures a specific feature in the home environment (e.g., the temperature), a device that can report a certain attribute of itself (e.g., a switch can report its `on`/`off` status), or a platform-defined feature (e.g., a location mode in SmartThings). (2) An actuator can be either a controllable device or platform-defined feature. An IoT *device* may be a sensor, an actuator or a combination. (3) The home environment has a set of measurable natural features, including time, temperature, illuminance, humidity, power consumption, etc. The interaction relation of sensors, actuators, and the environment is shown in Fig. 1. Sensors observe the home environment features and output the corresponding measurements while actuators can influence the reading of sensors either directly (e.g., by altering its own device attribute) or via the environment (e.g., by heating the home to change the measurement reading of a temperature sensor).

Control Layer. In appified home automation systems, the control layer consists of the automation *rules* defined by home automation apps. An app usually defines one or more rules. The rule follows a *trigger-condition-action* (TCA) model, as depicted in Fig. 1: (1) The *trigger* is a subscribed event (e.g.,

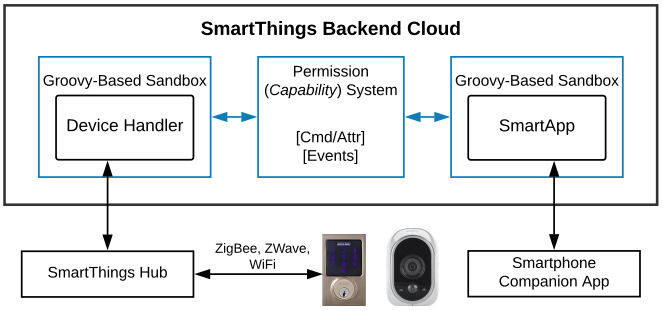


Fig. 2: SmartThings architecture overview.

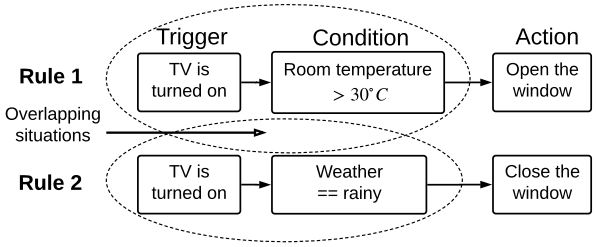


Fig. 3: An example of Action-Interference Threats.

the state of television changes to `on`) that fires the execution of the remaining flow of the rule. The platform listens to all data reported by sensors, and then broadcasts the related events to the subscribers. (2) The *condition* is a set of constraints defined in terms of the sensor data, the environment data (e.g., the time) or a user input (e.g., the room temperature is above 30°C), which must be satisfied in order to take the action. (3) The *action* (e.g., to open the window) is one or more commands issued to the actuators.

The data layer and the control layer interact in both directions. On the one hand, rules obtain data from the home context. For example, the trigger of a rule is usually a specific sensor data update (e.g., the state of TV changes to `on`); the condition takes sensor data and environment measurements as the inputs for constraint check. On the other hand, rules can influence the home context, i.e., the actions issue commands to the actuators, which may, in turn, change the environment features and sensor readings.

B. Appified Smart Home Platforms

Recently, multiple cloud-backed smart home platforms have emerged. They allow third-party developers to publish their IoT apps to help users manage, monitor and control their home devices. The integration of app developers is significant to share novel ideas and build smarter homes. A typical cloud-backed smart home ecosystem comprises three necessary components: a *hub* connecting IoT devices, a *back-end cloud*, and a *smartphone app*.

We use the Samsung SmartThings platform (as shown in Fig. 2), one of the most popular smart home platforms, as an example to describe these components. (1) The hub connects all IoT devices through short-range wireless techniques (ZigBee, Z-Wave, Bluetooth) or WiFi and bridges the communication between IoT devices and the Internet. Each of the low-cost and resource-constrained IoT devices usually only supports one of these wireless access techniques, while a hub is equipped with most, if not all, of the popular wireless capabilities. The hub plays a core role to ensure the interconnectivity and interoperability of IoT devices in the home environment. (2) The backend cloud (also referred to as *platform* in this paper), is another key part in the system. The SmartThings cloud abstracts real devices to *device handler* instances, which are software wrappers of the physical devices. A device handler handles the real underlying communication between the cloud and a physical device and exposes a subset of predefined uniform interfaces to the other components within the platform. *SmartApps* are Groovy programs provided by third-party developers and are used to provide automation rules for the home devices. Specifically, SmartApps can subscribe to the *events* fired by a set of instances of device handlers and issue *commands* to device handlers; thus, SmartApps interact with the real devices indirectly. The SmartThings provide the sandboxed runtime environment for running SmartApp and device handlers. (3) To provide a convenient user interface (UI) for users to manage their hubs, IoT devices and SmartApps, SmartThings also provide a smartphone *companion* app with a SmartApp store. In the companion app, users can install and configure a SmartApp, e.g., granting IoT devices that support the *capabilities*[1] requested by SmartApps, and filling out user-defined values. We call user-provided information in this phase the *configuration information*.

III. Categorization of CAI Threats

A rule can be modeled as a tuple of "*trigger-condition-action*". Suppose $R_1 = (T_1, C_1, A_1)$ and $R_2 = (T_2, C_2, A_2)$ denote two rules that are installed in the same environment, where T_i, C_i, A_i are the trigger, condition, and action of R_i, respectively. A Cross-App Interference threat arises when *the action of R_1 interferes with the trigger, condition, or action of R_2*. R_1 and R_2 may or may not belong to the same app, and our system HomeGuard can handle both cases, so we do not distinguish between the two cases in this paper.

Based on how R_2 is interfered with by R_1, we have identified the following three *basic* classes of CAI threats: *Trigger-Interference Threats*, *Condition-Interference Threats*, and *Action-Interference Threats*, which arise when the trigger, condition, and action of R_2 is interfered with by the action of R_1, respectively. We summarize the different categories of CAI threats in Table I. The caption of Table I gives the notations in this section.

A. Action-Interference Threats

Two rules R_1 and R_2 may be programmed to operate on the same actuator under different circumstances (decided by their own triggers and conditions). It is likely that both rules take effect simultaneously if they are both triggered ($T_1 = T_2$)

[1]See Appendix A for more details about capabilities.

TABLE I: Categorization of CAI threats. $R_i = (T_i, C_i, A_i), i = 1, 2$ denotes two rules, where T_i, C_i, A_i are the trigger, condition and action, respectively. $A_i \mapsto T_j$ denotes A_i satisfies T_j. $A_i \Rightarrow C_j$ and $A_i \not\Rightarrow C_j$ denotes that A_i satisfies and dissatisfies a subset of constraints in C_j, respectively. $G(A_i)$ denotes the goal of A_i.

	Category and description	Pattern
Action-Interference Threats	**Actuator Race (AR)**: Two rules perform contradictory actions on the same actuator(s).	$T_1 = T_2$, $C_1 \cap C_2 \neq \emptyset$, $A_1 = \neg A_2$
	Goal Conflict (GC): Two rules' actions have contradictory goals.	$(T_1 \cup C_1) \cap (T_2 \cup C_2) \neq \emptyset$, $G(A_1) = \neg G(A_2)$
Trigger-Interference Threats	**Covert Triggering (CT)**: A rule triggers the execution of other rules.	$A_1 \mapsto T_2$, $C_1 \cap C_2 \neq \emptyset$
	Self Disabling (SD): A rule triggers other rules which in turn disables it.	$A_1 \mapsto T_2$, $C_1 \cap C_2 \neq \emptyset$, $A_2 = \neg A_1$
	Loop Triggering (LT): Two rules trigger each other but perform contradictory actions on the same actuator(s).	$A_1 \mapsto T_2$, $A_2 \mapsto T_1$, $C_1 \cap C_2 \neq \emptyset$, $A_1 = \neg A_2$
Condition-Interference Threats	**Enabling-Condition (EC)**: A rule enables the execution of other rules.	$A_1 \Rightarrow C_2$
	Disabling-Condition (DC): A rule disables the execution of other rules.	$A_1 \not\Rightarrow C_2$

and their conditions are both satisfied ($C_1 \cap C_2 \neq \emptyset$). If R_1 and R_2 issue contradictory commands to the same actuator ($A_1 = \neg A_2$), an *Actuator Race* threat may arise, such that the final status of the actuator becomes unpredicted.

Fig. 3 shows such an example. If it is raining and the temperature is above 30°C, the conditions of both rules are satisfied, and both rules are triggered when the TV is turned on. However, the two rules issue opposite commands on the window, leading to a race condition. To verify the threat, we have constructed two SmartApps that define **Rule 1** and **Rule 2** shown in Fig. 3, respectively, and run them to control the same window opener's switch. We observed a variety of results: the switch is turned on only, turned off only, turned on then off, and turned off then on, showing that the final state is unpredictable.

Note that such races arise only under specific conditions. As a result, the threats may hide in a smart environment for a prolonged time. Such races may confuse the user and cause annoyance or device damages; they may even be exploited by attackers (e.g., the window is left open without being expected by the user).

A variant of Action-Interference Threats arise when two rules are executed on different actuators in certain situations ($(T_1 \cup C_1) \cap (T_2 \cup C_2) \neq \emptyset$), but their effects contradict ($G(A_1) = \neg G(A_2)$), which we call *Goal Conflicts*. Different from the instant race on an actuator, the two rules do not need to be triggered at the same time. The inter-actuator conflict is more subtle and implicit than intra-actuator races and hence more difficult to be realized by users. For instance, one rule is to turn on a heater, while the other is to open the window if the room is too dark; the two actions conflict in terms of heating up the room.

B. Trigger-Interference Threats

Since all rules interact within a common home context, a rule R_1's action may change the home context to a status that triggers another rule R_2 ($A_1 \mapsto T_2$); if the triggering happens under certain circumstances where R_2's condition is satisfied ($C_1 \cap C_2 \neq \emptyset$), R_2 is also executed after R_1. Thus, a group of explicitly defined rules may lead to new implicit rules, which

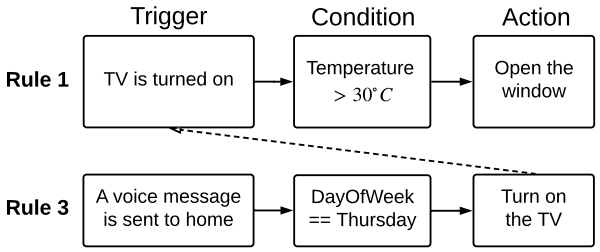


Fig. 4: An example of Trigger-Interference Threats.

we call *covert rules*. As shown in Fig. 4, **Rule 3** turns on the TV, which then triggers the execution of **Rule 1**. A covert rule "***when*** *sending a voice message* ***if*** *it is on Thursday and the temperature is over 30°C* ***then*** *open the window*" is formed.

In this example, a user may only intend to use **Rule 3** to turn on the TV remotely for the purpose of watching a live show immediately when arriving home, but it also triggers **Rule 1** to open the window, creating a chance for a burglar to break in before the user arrives home. In other words, such *Covert Triggering* may not be desired by the users. Thus, it is important to find such covert rules and alert the users when apps are installed.

There are two special cases of Trigger-Interference Threats: (1) *Self Disabling* that happens when R_1 covertly triggers R_2 but R_2's action contradicts that of R_1 ($A_2 = \neg A_1$). For instance, R_1 is defined as "***when*** *the motion sensor detects a motion at the front door* ***if*** *the temperature is above 30°C* ***then*** *turn on the air conditioner*", and R_2 is "***when*** *the energy meter's reading exceeds a threshold* ***then*** *turn off the air conditioner*". If turning on the air conditioner makes the reading of the power meter exceeds a threshold, the air conditioner will be turned off immediately after it is turned on. (2) *Loop Triggering* that occurs when two rules trigger each other ($A_1 \mapsto T_2$, $A_2 \mapsto T_1$, $C_1 \cap C_2 \neq \emptyset$) but perform contradictory actions on the same actuator(s) ($A_1 = \neg A_2$). For example, R_1 is defined as "***when*** *the illuminance is below 30 LUX* ***then*** *turn on the lights*", and R_2 is "***when*** *the illuminance is above 50 LUX* ***then*** *turn off the lights*"; as a result, when the two rules control the same set of lights, the lights may be continuously turned on and off, which can not only cause

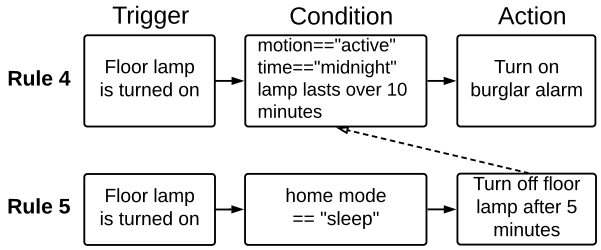


Fig. 5: An example of Condition-Interference Threats.

device damages, but also lead to seizures in photosensitive epilepsy sufferers [42].

C. Condition-Interference Threats

The action due to a rule R_1 may change the satisfaction of another rule R_2's condition, and thus affect the execution of R_2, which are called Condition-Interference Threats. However, unlike Trigger-Interference Threats, the action of R_1 cannot directly trigger the execution of R_2, as R_2 has its own trigger.

There are two types of Condition-Interference Threats: *Enabling-Condition Interference* and *Disabling-Condition Interference*, depending on whether the action of R_1 changes the condition of R_2 from `false` to `true` (Enabling, $A_1 \Rightarrow C_2$) or from `true` to `false` (Disabling, $A_1 \not\Rightarrow C_2$). Fig. 5 shows an example of Disabling-Condition Interference Threats, where **Rule 5** turns off the floor lamp automatically to save energy when the home is in the "`sleep`" mode, while **Rule 4** is used to detect break-ins. The action "turning off the floor lamp" in **Rule 5** disables the floor lamp checking in **Rule 4** and may lead to false negatives.

IV. Problem Scope and System Overview

A. Threat Model

Cross-App Interference threats do not depend on the over-privilege problem of the smart things platform; moreover, they do not rely on malicious code to be inserted into a *single* app, which may not be able to pass the manual code review step (e.g., code review of each submitted app is enforced by Samsung SmartThings). Thus, CAI threats are more stealthy and cannot be handled by existing approaches that analyze or review apps individually.

Our defense system considers and addresses CAI threats that can be caused by three types of IoT apps:

- **Flawless Benign Apps**: Although they contain no malicious or flawed code, they may cause CAI threats because of interaction with other apps.
- **Flawed Benign Apps**: They contain flawed logic, e.g., inconsistent rules, that leads to CAI threats.
- **Malicious Apps**: They contain malicious code that purposely exploits other apps deployed in the same home. The attacker can submit malicious apps onto the app store, or trick users to install them through the web.

Even non-malicious apps can cause CAI threats, which can surprise and confuse users and lead to security and privacy issues without involving any attackers. Although users may sometimes perceive CAI threats and avoid installing risky apps, various reasons lead to the failure of identifying such threats by relying on the carefulness of ordinary users. First, users may not understand all functionalities of an app by only reading its description. Second, a home may have multiple users; even a single user may install an app first, and after a long time, install another controlling the same device for distinct purposes. Third, users may write their own apps or obtain apps from third-party sources to realize novel automation ideas (e.g., controlling devices for a special purpose, achieving the same functionality with the limited devices they own, etc.). However, these apps may be flawed and vulnerable to CAI threats when working with other apps. Therefore, an automatic detection technique is necessary.

New Attack Vectors. Moreover, we identify three new attack vectors that exploit CAI threats. (1) If an attacker can infer or obtain the information about the apps installed at a target home, he can find the CAI threats at this location (e.g., the window is opened at a specific time) and exploit them accordingly. (2) The attacker can publish seemingly benign but malicious apps onto the app store, or trick users into installing them by advertising some useful functionalities. Such apps can be built to cause CAI threats by taking advantage of other apps installed at the target home. (3) If multiple collusive malicious apps are installed at the same home, they can cooperate to construct CAI threats and launch powerful finely-controlled attacks.

B. Goal and Problem Scope

The goal of our system is to detect CAI threats in a smart home, no matter they are caused by benign or malicious apps. This paper broadly uses *threats* to refer to all discovered interferences, acknowledging that *some of them may be unexpected and dangerous while others may be desired by users*. Instead of distinguishing the two cases, which is probably a non-computable problem, we present the detection results to homeowners in a human-readable way, which alerts the owners and allows them to make decisions on whether or not to keep the new app and or re-configure it.

In this paper, we focus on application security with the assumption of trusted and uncompromised platforms, devices and communication protocols. For example, attacks that exploit the hardware vulnerabilities or communication protocol flaws to intercept or fake the data used by an app and thus lead to an incorrect app behavior are out of our scope and should be taken care of by the device manufacturers, protocol designers and IoT platform to guarantee the authenticity and integrity of the data.

C. System Overview

In order to detect the CAI threats in a given home, all the apps and their interaction need to be considered collectively and systematically. To speed up the detection, the design of HomeGuard has the *offline* part and the *online* part: the offline part extracts and represents the automation semantics of each app in the form of rules (Section II-A); when a new app is being installed, the online part is invoked to detect

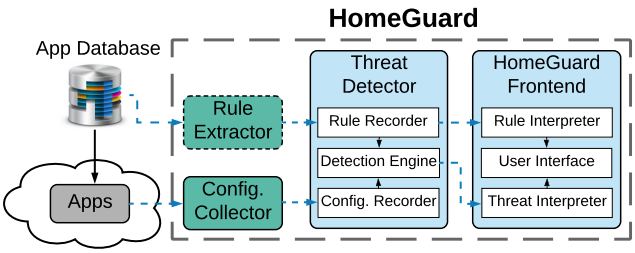


Fig. 6: The architecture of HOMEGUARD.

CAI threats by considering the interplay of all the rules of the new app and installed apps. The detection result is then presented to users for decision making. The advantage is that the online part can be computed efficiently by making use of the rules extracted offline. We build HOMEGUARD using a modular design. As shown in Fig. 6, it comprises the following four modules:

- **Rule extractor** extracts the rules of each smart home app offline. It exposes APIs for querying the rules of an app and also provides online rule extraction services for users who use custom apps. Other modules of HOMEGUARD are per-user processes and are called each time a new app $\mathcal{A}$ is being installed.
- **Configuration collector** collects the *configuration information* related to the installation, which includes the app name, the devices bound to the app $\mathcal{A}$, and the in-app static values (thresholds, boolean values, contact information, etc.).
- **Threat detector** is invoked when the *configuration recorder* receives the configuration information from *configuration collector*. After that, the *rule recorder* requests the rule information of $\mathcal{A}$ from the *rule extractor*. The *rule recorder* and the *configuration recorder* keep track of the historical rule and configuration information of apps, respectively. Based on the newly received rule and configuration information and the historical records, the *detection engine* detects CAI threats by considering the interplay of $\mathcal{A}$ and already installed apps.
- **HOMEGUARD frontend** bridges the system and smart home users. The *rule interpreter* translates rules of $\mathcal{A}$ into a human-readable form and displays them via a *user interface*, such that users can check if $\mathcal{A}$ itself will behave as it claims. The *threat interpreter* displays the detected CAI threats to users in a readable manner, who then decide whether the installation should proceed or whether the configuration should be adjusted.

Note that the *rule extractor* and the *configuration collector* are platform-specific since different IoT platforms use different programming languages and provide different APIs, while the *threat detector* and the HOMEGUARD *frontend* are platform-independent. The details of the *rule extractor* and the *threat detector* will be presented in Section V and Section VI, respectively. We present the other modules in Section VII.

V. RULE REPRESENTATION AND EXTRACTION

We present the rule representation in Section V-A and then the rule extraction technique in Section V-B. While our idea is applicable to multiple smart home platforms, we concretely demonstrate the proposed techniques on Samsung SmartThings platform. For the convenience of discussion, we develop 5 SmartApps, `ComfortTV`, `ColdDefender`, `CatchLiveShow`, `BurglarFinder`, and `NightCare`, which implement **Rule 1-5** depicted in Figures 3, 4 and 5, respectively. We use the code of `ComfortTV` (see Listing 1) to discuss the rule extraction. Discussion about rule extraction on other platforms is in Section VIII-D.

Listing 1: Code snippet of `ComfortTV`. Some irrelevant lines (e.g., metadata definition, UI-related sections and pages) are omitted.

```
input "tv1", "capability.switch", title: "Which TV?"
input "tSensor", "capability.temperatureMeasurement"
input "threshold1", "number", title: "Higher than?"
input "window1", "capability.switch"
def installed () {
  subscribe(tv1, "switch", onHandler)
}
def updated() {
  unsubscribe()
  subscribe(tv1, "switch", onHandler)
}
def onHandler(evt) {
  def t = tSensor.currentValue("temperature")
  if ((evt.value == "on") && (t > threshold)) turnOnWindow()
}
def turnOnWindow() {
  if(window1.currentSwitch == "off")
    window1.on()
}
```

A. Rule Representation

The goal of our rule extraction module is to precisely extract rule-related information from the app code and represent the information in a uniform form.

Listing 2: The rule representation format

```
Trigger:
(:subject).(:attribute)
(:constraint)
Condition:
(:data constraints)
(:predicate constraints)
Action:
(:subject)->(:command)(:paras)(:when)(:period)
(:data constraints)
```

Listing 2 shows the structured rule representation format we use. It contains detailed and precise information about the *trigger*, *condition*, and *action* of a rule. (1) the *trigger* is defined in terms of **subject** (e.g., a certain device), **attribute**, and **constraint** (that should be satisfied for executing the rule). (2) the *condition* comprises the **data constraints** (that describe how variables are assigned values) and the **predicate constraints** (that should be satisfied for proceeding to invoke the action). (3) the *action* depicts the **subject** on which **command** is issued, where **paras.** denotes the parameters related to the command and **data constraints** denotes all quantitative constraints involving the command parameters; plus, **when** denotes the scheduled time and **period** indicates the repetition interval for issuing the command. By default, both **when** and

period are equal to 0, meaning that the command should be issued with no delay and only once, respectively.

B. Symbolic Execution based Rule Extraction

We perform a static analysis on the source code of SmartApps to extract rules. Although most rules defined by SmartApps are static, we also handle rule dynamics due to configuration updating and the dynamic features of Groovy in Section VII and Section VIII-D, respectively.

Why did prior approaches fail? Prior approaches [29], [47] instrument SmartApps to insert runtime logging logic, so that when sensitive commands are issued during runtime, the *context* information can be collected. Such runtime logging approaches do not work for our purpose, as they only provide the information for rules that *have been executed*, while our goal is to extract *all* the rules *before* they are executed. SmartAuth [46] searches the Abstract Syntax Tree (AST) of the SmartApp source code to look for information of interest (e.g., the trigger event, the attribute, and the action) *without tracking the data flows*, so it cannot precisely retrieve the constraint information due to variable assignments and nested branches.

Why symbolic execution? In order to extract the rules of a SmartApp *completely* and *precisely*, we propose to symbolically execute the app, exploring all of its execution paths. Each path starts from an entry point and ends at a sensitive command (i.e., sink): the command reveals the *action* of a rule, while the path condition exposes the rule *trigger* and the *condition*. In order to enable symbolic execution on SmartApps, *the following questions and technical challenges need to be resolved*.

Path search strategy. A well-known limitation about symbolic execution is its poor scalability due to path explosion. However, as SmartApps are small and have limited paths, we are able to analyze them without encountering the path explosion problem. A simple *depth-first path search strategy* works well in our system.

Symbolic inputs. Data whose values are not dependent on other data are handled as *symbolic inputs* or *sources*. In SmartApps, sources include device references, device attribute values, device events, user input, HTTP response, constant values and return values of several APIs (see **API modeling** below). We achieve a completely automatic symbolic input identification. We parse all `input` method calls to collect device references (each device reference points to a globally unique 128-bit identifier for a home device connected to SmartThings) and user inputs (variables whose values are specified by users during app installation or update) and add a *symbolic input* label to each of them. Besides, we define variables to denote device attribute values used in the code and label them as symbolic inputs. Similarly, variables which accept HTTP responses and constant value are also labeled as sources. A special case is `State` and `atomicState`, which are objects for storing a small amount of data which can be shared across multiple SmartApp executions. We regard them as symbolic inputs as well. For example, in Listing 1, the devices references (`tv1`, `tSensor`, `window1`), the user input (`threshold`), and the return value of the API call at Line 13 are automatically labeled as symbolic inputs.

Analysis entry points and sinks. In our implementation, the analysis entry points include the lifecycle methods, `installed`, `updated` and `uninstalled`. The analysis *sinks* include capability-protected device commands and security sensitive SmartThings APIs (such as `setLocationMode()`). We consider 126 device control commands protected by 104 capabilities [8] and 21 SmartApp APIs (See Appendix A).

Generating Control-Flow Graph (CFG). We follow the approach in [29] to generate a control-flow graph built on AST transformation. Our design is to model the trigger-condition-action structure of a rule. A rule with a trigger usually starts from an event subscribed by in a `subscribe(dev,attr,hndl)` method (typically invoked in one of the analysis entry point methods). The `subscribe()` call means that when the device `dev`'s attribute `attr` changes, the event handler `hndl` should be invoked. Therefore, each `subscribe` method represents a trigger. Then we trace into the handler to identify sinks along the execution path. The path branches at conditional statement (e.g., `if` or `switch` statement) so we may reach different sinks, which are extracted as actions; the boolean expressions within the condition statements along the execution path from an entry point to a sink are used to construct the condition for that sink. The corresponding trigger, condition, and action are assembled into a rule.

Constraints for the *trigger* and *condition*. The `subscribe` method may define a trigger in different ways, i.e., using a state change (e.g., `subscribe(tv1, "switch", onHandler)`) or a certain value (e.g., `subscribe(tv1, "switch.on", onHandler)` to trigger the execution of the handler. If a conditional statement follows along the execution path to compare the subscribed event's value (e.g., Line 14 in Listing 1), the comparison in terms of the event's value is regarded as part of the trigger constraint; otherwise, the trigger is only a state change and has no constraint.

We track all data constraints and predicate constraints along the execution path from the entry point to sinks and attach them (excluding the trigger constraint) to the rule condition. We establish data constraints from each value assignment statement. Specifically, we write callback methods in the compiler to handle the 38 expression types defined in Groovy's documentation [24]. On the other hand, we also build predicate constraints from condition statements, i.e., each boolean expression in an `if` statement or each case expression in a `switch` statement is translated into a constraint. We also handle the *ternary* expressions by breaking each of them into two branches.

API modeling. The main challenge is to deal with the closed-source APIs provided by SmartThings. We first model the

TABLE II: Rule representation of **Rule 1**.

Trigger	Condition	Action
subject: tv1 **attribute**: switch **constraint**: tv1.switch==on	**data constraints**: t = tSensor.temperature tSensor.temperature=#DevState threshold1 = 30 **predicate constraints**: t > threshold1 window1.switch == off	**subject**: window1 **command**: on **paras**: [] **data constraints**: [] **when**: 0 **period**: 0

10 SmartApp APIs[2] that schedule the execution of specified methods according to their arguments and functionalities. For example, `runIn(delay, method)` executes `method` with the specified time delay. We attach the delay information to the scheduled method and continue to trace into the scheduled method to identify sinks. The successive sinks are also attached with the delay (note that we use a *when* property to handle delayed commands).

To model APIs that may be involved in the constraint construction, we model the objects, methods and object property accesses by manually reviewing the SmartThings developer documentation [10]. The return values of API methods and the object properties that do not rely on other data are also labeled as *symbolic inputs*. We model 173 API methods and 94 object property accesses in total and rewrite a static modeling function for each method or property access according to its arguments and return value. We further model a portion of external Java APIs that are used by SmartApps. Based on these modeling functions, we are able to construct constraints over expressions that contain API calls.

Compiler customization. To build the symbolic executor, we implement a compilation customizer instance and add it to the compiler configuration, which is supported by Groovy to allow developers to modify the compilation process at a certain phase. We work at the semantic analysis phase where the compiler creates a class node for each element (variable, method, expression, statement) in the source code and a set of visit methods that follow the generic Visitor pattern [38] can be implemented for different class node types to specify how the compiler processes these nodes.

As a concrete example, Table II shows the result of rule extraction on the code in Listing 1.

VI. CAI Threat Detection

Whenever a new app is installed or the configuration of an installed app is updated, our *Threat Detector* detects CAI threats by evaluating the interaction relations between the rules of the installed or updated app and those of apps already installed in the smart home.

[2]See Appendix A for the details.

A. Detecting Action-Interference Threats

We detect the Actuator Race (AR) and Goal Conflict (GC) threat (see Table I) between two rules R_1 and R_2 in two steps: action analysis and then overlapping-constraint detection.

1) Action Analysis: To detect Actuator Races, we first examine if the actions of R_1 and R_2 issue contradictory commands to the same actuator, or issue the same command with contradictory parameters; either of the two situations indicates the rule pair is an Actuator Race candidate. We use device IDs from the *configuration recorder* (see Section VII) to determine if the two rules control the same device. We maintain a global mapping M_{AR} that maps a device ID to the list of rules that operate on the device along with the command and parameter information. The mapping will be used in the subsequent overlapping-condition detection step.

To detect Goal Conflicts, we perform a goal analysis to determine if two actions contradict over a common goal. A goal in smart home consists of many measurable properties, such as temperature, illuminance, humidity, noise, etc. We consider how these properties are affected by each command of a device type. The effects are denoted as $+$ (increasing), $-$ (decreasing) and $\#$ (irrelevant). Accordingly, we construct another global mapping M_{GC} for the goal analysis. Note that virtual actuators (e.g., mode) that have no direct effect on the goal properties are not included in M_{GC}. R_1 and R_2 are considered as a Goal Conflict candidate if their actions have opposite effects on the same goal property.

2) Overlapping-Condition Detection: To determine whether a candidate rule pair R_1 and R_2 can lead to a real threat, we also need to know if they can be executed simultaneously in a certain situation, i.e., if they have overlapping trigger and/or condition. Note that we establish constraints for trigger and condition in our rule extraction (Section V-B). Therefore, the overlapping detection is transformed into a *constraint satisfaction* problem. We merge all constraints of the two rules as well as additional device constraints[3]; if the problem is solvable[4], it means two rules take effect together under certain situations (which can be derived from the resolution results), and the rule pair causes a true Action-Interference Threat.

B. Detecting Trigger-Interference Threats

The detection of Covert Triggering (CT) threat over two rules R_1 and R_2 is a directed process since Covert Triggering is not mutative. There are two ways that R_1 triggers R_2: (1) R_1 issues a command to an actuator (e.g., a switch), changing its certain state that is the trigger of R_2; (2) R_1 changes an environment feature (e.g., temperature) sensed by a sensor device whose reading is used as R_2's trigger. The trigger checking step follows the above two ways to determine whether R_1 could trigger R_2.

[3]Device constraints are to determine if two rules use the same device. See Section VII for more information.

[4]We choose the Java Constraint Programming (JaCoP) library as the solver since it is efficient and open-source in our implementation.

If the rule pair passes the trigger checking step, it is a CT candidate and an overlapping-condition detection is performed on the conditions of R_1 and R_2 to see if they are likely to execute together in certain situations. If the rule pair is a CT candidate and has overlapping conditions, R_1 and R_2 constitute a CT pair, denoted as $CT_{R_1 \to R_2}$.

When $CT_{R_1 \to R_2}$ (or $CT_{R_2 \to R_1}$) is detected, we further detect Self Disabling (SD) threats by examining the action analysis result in Section VI-A1. If R_1 and R_2 are also an Actuator Race candidate, an SD threat is detected. In addition, if both $CT_{R_1 \to R_2}$ and $CT_{R_2 \to R_1}$ are detected and R_1 and R_2 are AR candidate, a Loop Triggering Interference (LT) threat is discovered.

C. Detecting Condition-Interference Threats

To detect whether the rule R_1 has Enabling Condition (EC) Interference and Disabling Condition (DC) Interference with rule R_2, we evaluate whether R_1 enables/disables R_2's condition. Similar to the detection of Trigger Interference threats, there are two ways that R_1 can affect R_2's condition: (1) R_1 issues a command to an actuator, which changes the satisfaction of R_2's condition (e.g., R_1 turns on a heater and R_2 checks if the heater's state is `on`); and (2) R_1 changes an environment feature to enable/disable the condition of R_2 (e.g., R_1 turns on the heater and the condition of R_2 involves the room temperature). If R_1 affects R_2 in either of the above ways, the rule pair is a condition-interference threat candidate.

We then determine whether R_1 enables or disables R_2's condition by another overlapping-condition detection. Specifically, we first create an *effect constraint* to denote the effect of R_1's action. For instance, if R_1's action locks a door (`door1`), we generate the constraint `door1.lock==locked`; if R_1 sets the heating temperature of a thermostat to a value `T` and R_2 uses a temperature sensor (`tSensor`) in its condition, the effect constraint is `tSensor.temperature>=T`. We then merge the effect constraint with R_2's condition and solve the new constraint satisfaction problem. If the problem is solvable, R_2's condition may be enabled and otherwise disabled by R_1.

D. Detecting Chained CAI Threats

There may exist apps which satisfy one of the interference patterns but are still installed, decided by users. Hence, when we detect a rule r_1 defined by a new app interferes with (or is interfered with by) an existing rule r_2, we also need to detect if r_1 interferes with other rules indirectly via r_2 (or other rules interfere with r_1 indirectly). To this end, we record all rule pairs that satisfy certain pattern but are still installed by users in a list $Allowed$ in a bottom-up manner (from the user installed the first app for his home). After the pairwise detection between new rules and old rules, we search this detection result and the $Allowed$ list to find long-chained rules.

VII. Configuration Information Collection

Challenge. In Section V, we track the data flow by constructing data constraints starting from *sources*. Recall that we take as sources the `input` methods, which are rendered as user-friendly graphical interfaces for users to specify values for variables (referred to as configuration information in this paper) during app installation or updating. To precisely detect CAI threats, we need to know the values of such sources. Take the Actuator Race in the category of action-interference threats as an example: the two rules R_1 and R_2 should operate on the same actuator as a precondition to cause an Actuator Race. Such device binding as well as other configuration information (e.g., a user-defined threshold for comparison) cannot be obtained through static analysis. Therefore, how to collect the configuration information without modifying the SmartThings platform becomes a challenge since there are no APIs available for obtaining the configuration information from the SmartThings cloud or the companion app.

Solution. An ideal solution is to integrate our *threat detector* and *frontend* modules into SmartThings companion app, so that configuration information can be easily collected within the app. However, the companion app is not open-source and we do not assume any modification on the platform's cloud or mobile app for HomeGuard to work. Hence, we propose a practical solution to obtain configuration information, including a code instrumentation technique to collect configuration information from inside an app, and a messaging mechanism for transmitting the collected information to the HomeGuard app. The HomeGuard app implements the *threat detector* and *frontend* modules as another mobile app.

A. Code Instrumentation

Code instrumentation has been used in previous research on the SmartThings platform [29], [46], [47]. In our case, the instrumentation is only to gather the configuration information during app installation or configuration update, so it introduces a negligible overhead. We implement the SmartApp instrumentation with a Groovy script. Listing 3, as an example, shows how the app in Listing 1 is instrumented (most of the unaltered lines are omitted).

The lifecycle method `updated` is invoked during app installation or configuration update, so we insert configuration collection logic in `updated`. The inserted lines are the same for all apps except for Lines 8-10, which collect the configuration information specific to each app. The `appname` can be obtained from the metadata in the `definition` method[5]. In each item of the lists `devices` and `values`, the `devRefStr` and `varStr` are the variable names defined in `input` methods, and `devRef` and `var` denote the real values specified by users in the mobile app. The Groovy script reuses some code of the rule extractor to identify the `appname`, `devRefStr`, and `varStr`, so that the code instrumentation is a completely automatic process. The `collectConfigInfo` method (Line 14) assembles a Uniform Resource Identifier `uri` that contains the app name, a list of mappings between the device variable name

[5] `Definition` is a method that determines how the SmartApp is described in the mobile app UI.

Listing 3: Code snippet showing how the app in Listing 1 is instrumented. Line 3, Lines 8–11 and Lines 14–23 are the inserted code.

```
// ...                                                                    1
// Specify the phone that runs HomeGuard                                  2
input "patchedphone", "phone", required: true, title: "Phone             3
      number?"
// ...                                                                    4
def updated() {                                                          5
    // ...                                                              6
    // inserted code                                                     7
    def appname = "ComfortTV"                                            8
    def devices = [[devRefStr:"tv1", devRef:tv1], [devRefStr:"tSensor",  9
          devRef:tSensor], [devRefStr:"window1", devRef:window1]]
    def values = [[ varStr:"threshold1", var:threshold1]]                10
    collectConfigInfo (appname, devices, values)                         11
}                                                                        12
// ...                                                                   13
def collectConfigInfo (appname, devices, values) {                       14
    def uri = "http://my.com/appname:${appname}/"                        15
    devices.each { dev ->                                                16
        uri = uri + dev.devRefStr + ":" + dev.devRef.getId() + "/"       17
    }                                                                    18
    values.each { val->                                                  19
        uri = uri + val.varStr + ":" +val.var + "/"                      20
    }                                                                    21
    sendSmsMessage(patchedphone,uri)                                     22
}                                                                        23
```

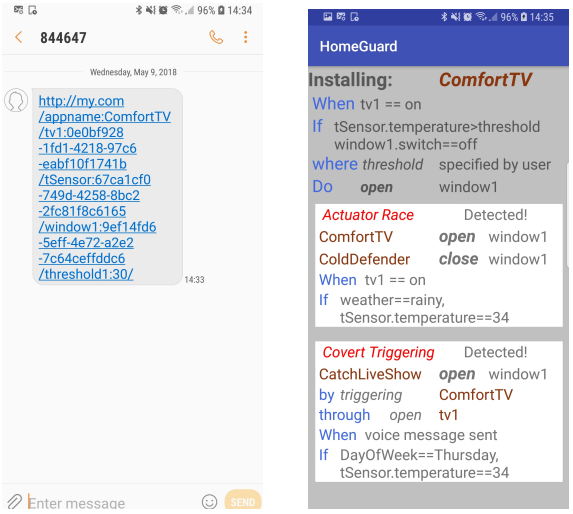
devRefStr and the underlying unique 128-bit device ID (devRef.getId() in Line 17), and the variable names and values (Line 20).

B. Messaging Deployment

We have two options for sending the above URI from SmartThings cloud[6] to the HomeGuard app: SMS (short messaging service) and HTTP based messaging. Both approaches are widely used for messaging and have their own pros and cons. For instance, SMS is easy to deploy but may fail to work if users go abroad, while HTTP works internationally but requires a relay server due to mobile IP addressing issues. We implement both SMS and HTTP solutions in our proof-of-concept prototype.

SMS messaging is easier to implement since SmartThings provide a convenient API sendSmsMessage for sending messages to a specified phone number. In Listing 1, Line 3 will be rendered as a UI for homeowners to fill out the phone number and Line 22 sends the collected configuration information (i.e., uri) to the specified phone. When installing or updating an app, the homeowner receives an SMS message containing uri. Fig. 7(a) shows an example of the SMS message received by the homeowner's smartphone. When the received link is clicked, the HomeGuard app installed on the same phone is launched and receives the link (e.g., by declaring an Intent Filter in Android OS [2]); it parses the link to obtain the configuration information. The HomeGuard app then sends an HTTP request with the *appname* to our backend server (hosting the rule extractor module) to retrieve the rules of the app. The backend server maintains a database to store the rules extracted from the public apps in SmartThings app store, in order to speed up the response time (note that the

[6]Note that SmartApps run on SmartThings backend cloud.



(a) The unique 128-bit device IDs and static values are encapsulated in a URI.

(b) The frontend app shows the rule defined by ComfortTV and the detected CAI threats.

Fig. 7: Screenshots showing the collected configuration information and the HomeGuard frontend app interface when installing ComfortTV to a home with ColdDefender and CatchLiveShow already installed.

backend server also provides an interface to extract rules from a custom app on demand). Besides, the HomeGuard app also generates device constraints (e.g., tv1=0e0b...741b) and user-defined value constraints (e.g., threshold1=30) based on the collected configuration information for detecting CAI threats. As the HomeGuard app records all the history information of the already installed apps, the threat detection can be performed efficiently. After that, the detection result is transformed and presented to the user; Fig. 7(b) shows an example. The user then can determine whether to keep or delete the new SmartApp, or change its configuration.

The HTTP-based implementation shares a similar design but employs Firebase Cloud Messaging (FCM) [6] for relaying the messages from SmartThings cloud to the homeowner's smartphone. At the initial startup of the HomeGuard app, Firebase generates a unique registration token for each HomeGuard app instance. In the instrumentation code, we ask the homeowner to specify the registration token instead of the phone number (Line 3 in Listing 3). Thus, the uri is sent to Firebase via the HTTP API httpPost (instead of sendSmsMessage) and the firebase server then pushes uri as a notification message to the HomeGuard app instance specified by the registration token. The notification message is rendered as a Notification [3] for the HomeGuard app. By clicking the notification, the HomeGuard app is launched to detect CAI threats.

VIII. Evaluation

We demonstrate how CAI threats can be constructed using some seemingly benign malicious apps and how the threats can be exploited VIII-A. We then evaluate the effectiveness

and efficiency of HOMEGUARD in Section VIII-B and Section VIII-C, respectively.

A. Exploitation Experiments

As presented in Section IV-A, new attack vectors exploiting CAI threats arise. In order to demonstrate the exploitation feasibility, we act as an attacker and use 5 SmartApps we have developed, i.e., `ComfortTV`, `ColdDefender`, `CatchLiveShow`, `BurglarFinder`, and `NightCare`, to simulate the exploitation of CAI threats. Each of these apps seems to provide useful functionalities. But when these apps are installed in the same home, their interplay introduces CAI threats and can be exploited. For instance, `BurglarFinder` and `NightCare` cause a condition-interference threat, and the user may be unaware that her `BurglarFinder` is already disabled by `NightCare`, and the attacker may break in without triggering alerts. We install all the five apps on SmartThings Web IDE and observe that these apps interfere with each other through the trigger, condition, or action, as discussed in Section III. In this experiment, we prove that SmartThings currently has no mechanisms to detect or handle CAI threats, which validate the criticality and urgency of our research.

B. Effectiveness

Rule Extraction. We first evaluate the implemented *rule extractor*'s ability to extract automation rules from SmartApps. We collect all the 182 SmartApps in the public repository [11] and manually remove the 36 Web Services SmartApps, as these web services SmartApps expose web endpoints for external applications to get device information or control devices through web API calls and do not define automation rules themselves [10]. We manually review the code of the remaining 146 SmartApps and record the rules per app. To avoid human errors, we also install these apps and use the simulated devices provided by SmartThings to verify the correctness. The manual analysis results are used as *ground truth*. Then we use our rule extractor to automatically extract rules from these apps and compare the results with the ground truth.

Our rule extractor can analyze most apps (124 out of 146) correctly. There were several special cases we did not expect. `Feed My Pet` uses `device.petfeedershield` in the `input` method instead of a capability; `Sleepy Time` uses `device.jawboneUser`; and `Camera Power Scheduler` uses a public API `runDaily`, which is not documented by SmartThings. We have added the non-standard device types into the capability list and modeled the undocumented APIs we encountered to fix the issue.

As our rule extractor employs symbolic execution to explore all the paths in apps systematically, it is able to extract the rules in an app precisely and completely. Thus, we envision that it can help the SmartThings staff review the code of SmartApps to check whether a SmartApp contains malicious behaviors. To evaluate its effectiveness in extracting rules from malware, which may hide malicious logics and thus impose extra challenges to our rule extractor, we ran it over 18 representative malicious SmartApps collected from Literatures [22], [29], [46], [47] to test if it can extract rules correctly. As shown in Table III, the rule extractor can obtain the precise rules for the majority of the cases. Therefore, the rule extractor can be applied to better detect malicious apps.

Two exceptions are the *endpoint attack* and the *app update attack*. The SmartApps used for endpoint attacks are web service apps which do not define automation themselves but expose callable endpoints for third parties, i.e., the automation rules are defined outside of SmartApps, so the rule extractor cannot obtain the complete automation logics by only analyzing SmartApps. However, the rule extractor can still identify malicious logics embedded in the request handler methods. The app update attack is difficult to detect by static methods since SmartApp developers can update the cloud instances for all users without any user awareness. This can be solved through the platform by enforcing the checking whenever an app is updated.

CAI Threat Detection on Real Cases. In order to demonstrate the capability of HOMEGUARD in finding CAI threats in real-world cases, we pick up 90 out of the 146 apps in the SmartThings app repository for testing. We exclude 56 apps because their functionalities are to send notifications to the home owner's smartphone and do not control devices. As this test is to find all possible pairs having CAI threats from a pool of apps and it is infeasible to exhaustively try all device-app binding situations, we consider two rules use the same device if they use devices of the same type[7]. To avoid excessive false positives due to this setting, we classify devices using `capability.switch` into different types according to the app description, since various types of devices support `capability.switch` to indicate `on/off` states. Note that in the real deployment, HOMEGUARD distinguish if two rules operate on the same device according to the 128-bit device IDs in the collected configuration information (see Section VII). We perform the CAI threat detection for each pair of the 90 apps and record the results. The SmartThings platform provides a well-built simulator and simulated devices, allowing us to verify the correctness of the discovered threats. We also purchase some real devices, including a SmartThings hub v2, a motion sensor, a presence sensor, a multipurpose sensor (combining a contact sensor and a temperature sensor), two bulbs, two smart outlets to reproduce and verify a subset of the discovered threats.

We find that a lot of apps can cause CAI threats and show the statistics in Fig. 8. Apps that control a commonly used switch or a mode tend to involve all kind of the threats. Below, we describe some detected CAI threats in SmartApps.

1) `SwitchChangesMode`[8] changes the current mode of a smart home according to the `on/off` state of a switch, and `MakeItSo` binds a group of states of several switches,

[7]The device types used by a SmartApp can be determined by examining their associated `capabilities`.

[8]The spaces in SmartApp names are omitted.

TABLE III: Extracting rules from malicious apps.

Attack	Description	Malicious SmartApp Name	Can handle?
Malicious Control	Embed malicious logics beyond app description	CreatingSeizuresUsingStrobedLight	✔
Abusing Permission	Exploit overprivilege to perform attacks	shiqiBatteryMonitor	✔
Adware	Embed ads into notification messages	HelloHome/CODetector	✔
Spyware	Leak private information via HTTP/side channel	LockManager/shiqiLightController/DoorLockPinCodeSnooping	✔
Ransomware	Refuse to take actions until user pay money	WaterValve	✔
Remote Control	Execute dynamic commands according to HTTP response	SmokeDetector/FireAlarm	✔
IPC	Malicious apps exchange information by IPC	MaliciousCameraIPC & PresenceSensor	✔
Shadow Payload	Send sensitive information to attacker's encrypted url	AutoCamera2	✔
Endpoint Attack	Trigger malicious functions via HTTP requests	BackdoorPinCodeInjection/DisablingVacationMode	✗
App Update	Edit the original codes after released	BonVoyageRepackaging/PowersOutAlert	✗

locks, and thermostats to a mode and restores the group of states each time the home goes into that mode. The two apps may create a covert rule that a switch's state triggers the action of unlocking a door.

2) `CurlingIron`, which turns on a set of outlets (switches) if a motion is detected, may create a covert rule by chaining with `SwitchChangesMode` and `MakeItSo`. This covert rule unlocks a door when motion is detected. The covert rule introduces a new attack surface that a burglar unlocks the door by spoofing the motion sensor (e.g., using CO2 laser [43]).
3) `NFCTagToggle` allows a user to toggle a set of appliances and door locks by tapping a smartphone app button, and `LockItWhenILeave` locks doors automatically if the user's presence sensor leaves a location. If the user taps the app button to turn off appliances and lock doors after leaving home but `LockItWhenILeave` has already locked the door, the door will be unlocked.
4) `LetThereBeDark` leads to action races on the same lights when working with other light control apps, such as `UndeadEarlyWarning`, `LightsOffWhenClosed`, `SmartNightlight`, `TurnItOnFor5Minutes`, etc.
5) `It'sTooHot` and `EnergySaver` impose a Self-Disabling threat. `EnergySaver` turns off a set of devices when the real-time electricity usage is over the user-defined threshold, which may disable `It'sTooHot` to turn on air conditioners or fans in a narrow situation: the turning-on of air conditioners is the last straw that makes the electricity usage exceed the threshold.
6) `LightUptheNight` contains a Loop-Triggering interference, leading to unexpected light flashing. This app is a real-world case of the LT example in Section III-B.

C. Efficiency

Rule Extraction Computation and Storage. To test the efficiency of our rule extractor, we run it 10 times on all 146 apps and get the average execution time. The time is 1341ms per app on average on a desktop with 3.4GHz Intel Core i7 CPU-6700, 8GB memory, and Ubuntu 16.04 LTS. Note that the rule extraction for the publicly available apps is a one-time cost and can be done offline. The performance is satisfactory even for providing online services to users who would like to install their own custom apps. We also test the size of the rule file of a SmartApp, which needs to be stored on the HOMEGUARD server hosted by *rule extractor* and transmitted to the user's mobile device. In our implementation, we use JSON strings to store rules and the rule file for an app is 6.2KB on average.

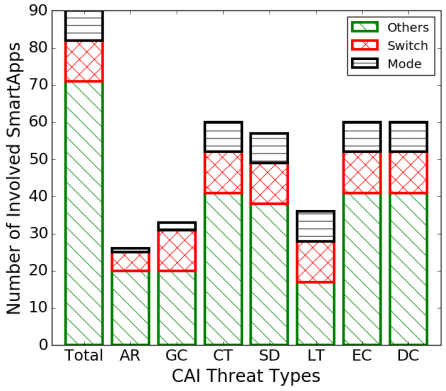


Fig. 8: Statistics of the detection result on 90 SmartApps. *Switch*: controlling a `capability.switch` without specifying the specific device type (e.g., light, valve, television, etc.); *Mode*: controlling location `mode`; *Others*: controlling other devices. The threat acronyms are defined in Table I.

Configuration Information Collection Speed. Both configuration information collection and threat detection are done online and their speed directly affects user experience. In our implementation, the latency introduced by configuration information collection depends on the response time of SmartThings cloud and the SMS/HTTP transmission latency. We time the response time by inserting invoking (`now()`) to get the Unix epochs T_1 and T_2 before and after the instrumentation code, respectively. The time duration ($T_2 - T_1$) is 27 ms. Similarly, we record the epoch T_3 when we receive the SMS/HTTP message on the phone and calculate the transmission latency ($T_3 - T_2$). The averaged latency from 100 trials is 3120ms for SMS and 1058ms for HTTP, which we believe is acceptable during installation.

CAI Detection Speed. CAI detection is a real-time process and runs on users' mobile devices. To evaluate the efficiency of HOMEGUARD, we test the averaged execution time for

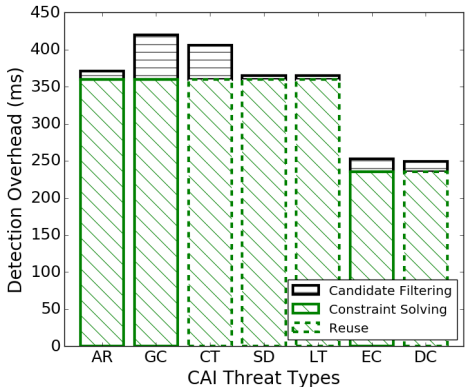


Fig. 9: CAI detection overhead for a pair of rules. Green dotted lines mean the constraint solving for detecting CT, SD, and LT threats can reuse the solving result of AR and the constraint solving for DC can reuse that of EC. The threat acronyms are defined in Table I.

detecting a specific CAI threat between two rules on a Samsung Galaxy S8 smartphone with Android OS 8.1.0. As shown in Fig. 9, the most time-consuming operation is constraint solving. The constraints solving overhead in EC is lower since involved constraint number is half of that in AR and GC. To avoid unnecessary constraint solving, we first perform a light-weight candidate filtering based on the pre-stored mapping lists and reuse the constraint solving result in detecting different threats. For an arbitrary pair of rules, the maximum total time for detecting all CAI threats is 1156 ms. The actual detection time is much less since two rules rarely fit all threat patterns and thus constraint solving overhead can be avoided.

D. Discussion and Limitations

1) User Intervention: Due to the inherent weakness of dynamic logging, previous work [29] needs multiple occurrences of user intervention when an app issues a command. By utilizing the powerful path searching capability of symbolic execution, HOMEGUARD extracts the rules of an app all at once. Hence, users in this paper only participate in a one-time decision making on the HOMEGUARD frontend app when they manually install a new app or update the configuration of an already installed app. Users focus more on handling app functionalities during installation than during the app's daily execution, so HOMEGUARD does not risk minimum user habituation or annoyance. It is infeasible to further eliminate the one-time decision making because inter-app interactions are somewhat subjective, as we discussed in Section IV-B.

2) Dynamic Features of Programming Languages: A concern may be that our static symbolic execution engine cannot effectively deal with the dynamic features of Groovy. However, all SmartApps are instances of an abstract class `Executor` which provides a variety of methods available for app development and run in the sandboxed environment, which is customized and enforced by SmartThings. SmartThings enforces all SmartApps to implement an abstract class `Executor` which provides a set of methods and restricts access to many Groovy methods and features. Thus, only `GString` is an allowable dynamic feature in SmartApps. `GString` enables remote servers to control the property accesses and method calls in a SmartApp by manipulating the string values. However, this concern is relieved because the SmartThings code review bans dynamic method execution and needs developers to use a `switch` statement on all possible `GString` values before doing anything with it [9]. Therefore, our static analysis handles all possible values of `GString` separately and branches the execution path when encountering a switch statement.

TABLE IV: Manners for defining rules on different platforms

Platform	Manner	Language	Specific APIs?
Android Things	program	Java	✔
HomeKit	program	Swift/Objective C	✔
OpenHAB	program	Domain Specific Language	✔
SmartThings	program	Groovy	✔
IFTTT	template	–	–

3) Backward Compatibility: A practical concern is how to detect CAI threats in apps that were installed *before* the employment of the proposed system. This problem can be solved by HOMEGUARD conveniently without developer efforts. Users can reinstall the instrumented versions of these apps on the phone companion app without changing their configurations; the instrumentation code in `updated` methods will be invoked and then the HOMEGUARD app starts to detect CAI threats, as we discussed in Section VII.

4) Multi-Platform Applicability: HOMEGUARD supports different IoT platforms by design, considering home members may work with multiple platforms simultaneously. The rule extractor module is platform-specific since platforms may use different languages and customized APIs, as shown in Table IV. Our work shows that symbolic execution works well in extracting automation rules from the source code of Groovy-based SmartApps. Symbolic execution has been widely used for software testing in various systems [26], [35] and shows its powerfulness for supporting multiple languages such as source code [17], bytecode [39] and custom intermediate representations (IRs) [25]. Therefore, the only engineering effort for supporting multiple platforms is to implement our symbolic execution based rule extractor for other languages. Other than programs/apps, some platforms define rules through templates. For example, IFTTT provides graphical interfaces on its mobile app and web page for users to define automation rules by selecting pre-defined templates and filling out parameters. Rules can be extracted by crawling text data on the related pages and parse the texts with natural language processing (NLP) technologies [28].

TABLE V: Comparison with related work.

Name	Inter-app Analysis	Proactive Defense	Low Overhead	No Runtime Intervention
ContexIoT	✗	✗	✗	✗
ProvThings	✔	✗	✗	✔
SmartAuth	✗	✔	✔	✔
HOMEGUARD	✔	✔	✔	✔

IX. RELATED WORK

A. IoT Security and Privacy

Recently, IoT platform security has been extensively studied. Fernandes et al. use a black-boxed static analysis on Samsung SmartThings, revealing several significant design flaws, such as coarse capability, coarse SmartApp-SmartDevice binding, and insufficient event data protection, which can or have been exploited by SmartApps to perform overpriviledged actions [22]. ContexIoT proposes a context-based permission system for IoT platforms to identify fine-grained context information and prompt the information at runtime for users to make an access control decision [29]. To separation the execution of a SmartApp into context collection and permission granting phases, they are the first to use patching mechanism to collect runtime data and pause the execution of SmartApps. However, ContexIoT needs users to participate in making decisions at runtime, which overshadows the benefit of home automation, and worse, may violate the 20-second execution time limits for each method in SmartThings if users do not respond in time. SmartAuth performs static analysis of the source code and uses NLP techniques to analyze code annotations, capability requests of SmartApps and developers' descriptions in the app store, to detect whether a SmartApp's functionalities faithfully follow the expectation of users [46]. Due to the nature of NLP, SmartAuth suffers if a malicious app uses customized meaningless or even malicious method and property names to hide the real SmartThings commands and attributes. Also, the static analysis for extracting rules of a SmartApp can be improved since the authors only target a subset of data types. Wang et al. presents *ProvThings* [47], a platform-centric logging framework that can construct data provenance graphs for all activities in an IoT system and use them to find out reasons for troubleshooting when an abnormality occurs. The ProvThings is prototyped for the SmartThings by instrumenting SmartApps. ProvThings is mainly for forensics rather than proactive defense. Tyche introduces a risk-based permission system to solve the overprivilege problems by grouping permissions according to their risk levels and allows users to grant permissions for a device by specifying an allowed risk level [40]. Tyche enforces the risk-based model by rewriting SmartApps based on AST transformations. FlowFence protects IoT data from leakage and misuse by using sandboxes and taint-tracking to enforce data flows between data sources and data sinks. The implementation is done on an Android OS which is not the mainstream in current IoT hubs and clouds. A comparison of existing appified IoT security solutions is shown in Table V.

In addition to platforms, IoT security and privacy in other dimensions also have been widely studied. Fernandes et al. shed light on the distinctive characteristics of IoT security from classic IT security in hardware, software, network, and application layers [23]. Arias et al. [14] and Liu et al. [33] conduct broad studies on the security threats in IoT device development and deployment. Tan et al. [45] propose an attestation solution to ensure the device firmware and software integrity. Chen et al. propose a fuzzing-based detection framework to find memory corruption vulnerabilities in IoT devices [18]. A lot of work focus on identifying cyber attacks [13], [44] and hardening communication and authentication protocols [49], [27], [12]. Siby et al. design a framework *IoTScanner* for passively monitoring and analyzing IoT traffics [44]. Also, formal analytics based on probabilistic models are employed to estimate the security risks and guide the configurations of IoT systems [36], [37].

B. Collusive Attacks in Mobile Apps

We discuss collusion attacks in the mobile application domain since collusion attacks share similarities with CAI threats. That is, collusion attacks also leverage multiple apps to bypass malicious code anti-malware techniques that detect per single app, such as Google Play Protect which is based on machine learning and anti-malware apps (e.g., Avast).

A lot of work have been done to characterize and detect mobile app collusion attacks. Davi et al. [20] shed light upon possible privilege escalation attacks in Android. Xu et al. [48] study app collusion where one app surreptitiously launches others in background and develop a static analysis tool for detecting app collusion by examining app binaries. Marforio et al. [34] give a comprehensive introduction of possible collusion channels. Chin et al. [19] firstly present a comprehensive analysis of threats based on inter-app ICC (Inter-Component Communication). Many collusion app detection works are based on examining inter-app ICC data flows [32], [30], [41], [31]. Bosu et al. [16] enhances the scalability and accuracy for large-scale detection by designing a new open-source resolution tool for inspecting inter-app ICC data flows.

However, the collusion attacks and CAI threats are different in many aspects. Collusion attacks need two or more malicious apps to actively collude by design to launch attacks. That is, a privileged app collects sensitive information and sends it to another app (or apps), which then sends the information outside the boundaries of the device to steal private data, or perform harmful actions (e.g., financial transactions) with the received information. Therefore, explicit collusion logics and inter-app communication supported by the mobile system architecture are necessities for apps to collude. Thus, code analysis and information tracking (e.g., taint analysis [15], [21]) techniques are effective to identify collusions by focusing on inter-app communication channels. In CAI threats, apps do not have explicit malicious code for collusion and do not need to convey data. The interplay between apps is through

controlling devices and affecting the home environment, which needs to be handled by totally different techniques. Moreover, collusion attacks usually abuse permission to, for example, collect and disclose private data, while CAI threats cause various conflicts and chained execution without violating or abusing permissions. Last but not least, collusive attacks are achieved via deliberately distributed malicious mobile apps or SDKs (Software Development Kits); other than malicious apps, CAI threats can also be caused due to installing and/or configuring benign apps.

X. Conclusion

In an appified smart home, multiple independently developed apps may interplay and interfere with each other, causing undesired and even dangerous conflicts and covert rules, which we call Cross-App Interference (CAI) threats. Such threats may not only lead to unexpected automation, but also introduce security and privacy problems to the smart home. We have categorized CAI threats and introduced new attack vectors that exploit them. Without proper handling, the problem will exacerbate when an increasing number of smart devices and apps are installed at a smart home. We have designed and built a system HomeGuard to address the problem. It applies symbolic execution to extract rules from apps completely and precisely and employs a constraint solver to evaluate the relation between rules for systematic threat detection. Moreover, we have proposed a practical deployment path that utilizes code instrumentation to collect the installation information and a frontend app to perform the detection on the user's smartphone. We evaluated HomeGuard using real SmartApps in the app store and discovered a large number of potential threats. The evaluation results show that HomeGuard is effective, efficient and precise.

References

[1] "Iot platforms company list 2017 update," https://iot-analytics.com/iot-platforms-company-list-2017-update/, 2017.

[2] "Android intents and intent filters," https://developer.android.com/guide/components/intents-filters, 2018.

[3] "Android notifications," https://developer.android.com/guide/topics/ui/notifiers/notifications, 2018.

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

[5] "Apple homekit," https://www.apple.com/ios/home/, 2018.

[6] "Firebase cloud messaging," https://firebase.google.com/docs/cloud-messaging/, 2018.

[7] "Smartthings," https://www.smartthings.com/, 2018.

[8] "Smartthings capabilities reference," https://docs.smartthings.com/en/latest/capabilities-reference.html, 2018.

[9] "Smartthings code review guidelines," http://docs.smartthings.com/en/latest/code-review-guidelines.html, 2018.

[10] "Smartthings developer documentation," http://docs.smartthings.com/en/latest/, 2018.

[11] "Smartthings public github repository," https://github.com/SmartThingsCommunity/SmartThingsPublic, 2018.

[12] S. A. Anand and N. Saxena, "Vibreaker: Securing vibrational pairing with deliberate acoustic noise," in *ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016.

[13] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman *et al.*, "Understanding the mirai botnet," in *USENIX Security Symposium*, 2017.

[14] O. Arias, J. Wurm, K. Hoang, and Y. Jin, "Privacy and security in internet of things and wearable devices," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 1, no. 2, pp. 99–109, 2015.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[16] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *ACM on Asia Conference on Computer and Communications Security*, 2017.

[17] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[18] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *NDSS*, 2018.

[19] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *ACM conference on Mobile systems, applications, and services*, 2011.

[20] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Springer International Conference on Information security*, 2010.

[21] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[22] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *IEEE Symposium on Security and Privacy (SP)*, 2016.

[23] E. Fernandes, A. Rahmati, K. Eykholt, and A. Prakash, "Internet of things security research: A rehash of old ideas or new intellectual challenges?" *IEEE Security & Privacy*, vol. 15, no. 4, pp. 79–84, 2017.

[24] A. S. Foundation, "Java constraint programming solver," http://groovy-lang.org/documentation.html, 2018.

[25] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *IEEE Symposium on Security and Privacy (SP)*, 2016.

[26] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[27] N. Z. Gong, A. Ozen, Y. Wu, X. Cao, R. Shin, D. Song, H. Jin, and X. Bao, "Piano: Proximity-based user authentication on voice-powered internet-of-things devices," in *IEEE ICDCS*, 2017.

[28] I. Hwang, M. Kim, and H. J. Ahn, "Data pipeline for generation and recommendation of the iot rules based on open text data," in *IEEE WAINA*, 2016.

[29] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "Contexiot: Towards providing contextual integrity to appified iot platforms," in *NDSS*, 2017.

[30] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014.

[31] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *International Conference on Software Engineering-Volume 1*. IEEE Press, 2015.

[32] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder, "Mr-droid: A scalable and prioritized analysis of inter-app communication risks," in *IEEE Security and Privacy Workshops (SPW)*, 2017.

[33] H. Liu, C. Li, X. Jin, J. Li, Y. Zhang, and D. Gu, "Smart solution, poor protection: An empirical study of security and privacy issues in developing and deploying smart home devices," in *ACM Workshop on Internet of Things Security and Privacy*, 2017.

[34] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *ACM Annual Computer Security Applications Conference*, 2012.

[35] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[36] M. Mohsin, Z. Anwar, G. Husari, E. Al-Shaer, and M. A. Rahman, "Iotsat: A formal framework for security analysis of the internet of things (iot)," in *IEEE CNS*, 2016.

[37] M. Mohsin, Z. Anwar, F. Zaman, and E. Al-Shaer, "Iotchecker: A data-driven framework for security analytics of internet of things configurations," *Computers & Security*, vol. 70, pp. 199–223, 2017.

[38] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *IEEE Computer Software and Applications Conference (COMPSAC)*, 1998.
[39] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *IEEE/ACM International Conference on Automated Software Engineering*, 2010.
[40] A. Rahmati, E. Fernandes, K. Eykholt, and A. Prakash, "Tyche: Risk-based permissions for smart home platforms," *arXiv preprint arXiv:1801.04609*, 2018.
[41] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, "Multi-app security analysis with fuse: Statically detecting android app collusion," in *ACM Program Protection and Reverse Engineering Workshop*, 2014.
[42] E. Ronen and A. Shamir, "Extended functionality attacks on iot devices: The case of smart lights," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.
[43] R. H. Schleijpen and F. J. van Putten, "Using a co2 laser for pir-detector spoofing," in *Technologies for Optical Countermeasures XIII*. International Society for Optics and Photonics, 2016.
[44] S. Siby, R. R. Maiti, and N. O. Tippenhauer, "Iotscanner: detecting privacy threats in iot neighborhoods," in *ACM International Workshop on IoT Privacy, Trust, and Security*, 2017.
[45] H. Tan, G. Tsudik, and S. Jha, "Mtra: Multiple-tier remote attestation in iot networks," in *IEEE CNS*, 2017.
[46] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in *USENIX Security Symposium*, 2017.
[47] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *NDSS*, 2018.
[48] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu, "Appholmes: Detecting and characterizing app collusion among third-party android markets," in *International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017.
[49] J. Zhang, Z. Wang, Z. Yang, and Q. Zhang, "Proximity based iot device authentication," in *IEEE INFOCOM*, 2017.

APPENDIX

A. Sinks in the Symbolic Executor

Sinks are defined for the symbolic executor to identify rule actions. The most common *sinks* are commands supported by *capabilities*. Capabilities, similar to *permissions* in mobile applications, abstract various types or subtypes of real devices according to functionalities. A real device supports one or more capabilities and can be granted to SmartApps which request the supported capabilities through `input` methods. Each capability contains a set of *attributes* that can be accessed by apps and *commands* that control the real device. For example, `capability.lock` defines an attribute `lock` which indicates a lock device's `locked`/`unlocked` state and two commands `lock()` and `unlock()` for controlling a lock. The capability-defined commands are considered as sinks.

In addition, SmartThings provide a lot of APIs for building SmartApps and device handlers. To filter out irrelevant method calls that are not home automation actions, we only consider APIs that perform sensitive actions (see Table VI), e.g., read attributes from sensors, issue commands to actuators, or send data to third-party devices or servers, as sinks (i.e., rule actions) in our rule extraction.

TABLE VI: SmartThings provided APIs we considered as *sinks* in rule extraction.

API Name	Description

`httpDelete`
`httpGet`
`httpHead`
`httpPost`
`httpPostJson`
`httpPut`
`httpPutJson`
	Executes an HTTP DELETE/GET/HEAD/POST/PUT request

`runIn`	Executes the specified method after the specified seconds

`runEvery1Minute`
`runEvery5Minutes`
`runEvery10Minutes`
`runEvery15Minutes`
`runEvery30Minutes`
`runEvery1Hour`
`runEvery3Hours`
	Creates a recurring schedule that executes the specified method periodically, as indicated by the method name

`runOnce`	Executes the specified method once at the specified date and time
`schedule`	Creates a scheduled job that calls the specified method once per day at the specified time
`sendHubCommand`	Sends a command to the SmartThings hub, which then issues a command defined in the arguments to LAN-connected devices to access device attributes or control these devices
`sendSms` `sendSmsMessage`	Sends an SMS message to the specified phone number
`setLocationMode`	Set the home's `Mode` to the specified value, which can be subscribed or accessed by SmartApps