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

Haotian Chi*, Qiang Zeng†, Xiaojiang Du* and 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*—Internet of Thing platforms prosper home automation applications (apps). Prior research concerns intra-app security. Our work reveals that automation apps, even secured individually, still cause a family of threats when they interplay, termed as *Cross-App Interference* (CAI) threats. We systematically categorize such threats and encode them using satisfiability modulo theories (SMT). We present HOMEGUARD, a system for detecting and handling CAI threats in real deployments. A symbolic executor is built to extract rule semantics, and instrumentation is utilized to capture configuration during app installation. Rules and configuration are checked against SMT models, the solutions of which indicate the existence of corresponding CAI threats. We further combine app functionalities, device attributes and CAI types to label the risk level of CAI instances. In our evaluation, HOMEGUARD discovers 663 CAI instances from 146 SmartThings market apps, imposing minor latency upon app installation and no runtime overhead.**

I. INTRODUCTION

The rapid proliferation of Internet-of-Things (IoTs) has advanced the development of smart homes to a new era. Moreover, smart home platforms connect IoT devices and offer programming frameworks for deploying home automation applications. Representative platforms include Samsung SmartThings [2], Apple HomeKit [3], and IFTTT [4]. However, appified frameworks also introduce new app-level surfaces, which could be misused by homeowners and exploited by attackers. For example, a burglar can exploit a vulnerable IoT app to open a smart lock [5], which is impossible in non-appified homes.

Many works contribute to enhancing the app-level security in smart home systems [5]–[15]. Fernandes et al. [5] revealed the *overprivilege* problem in the permission (a.k.a., capability) system of SmartThings and demonstrated exploits that expose smart homes to severe attacks. Follow-up works propose to resolve *overprivilege* by patching the existing permission system [11], developing new permission mechanisms [9], detecting overprivileged apps [12], [13], or enforcing non-overprivileged authorization [12]. This work, however, shows that *Cross-App Interference* (CAI) threats may be caused even if apps are secured individually: CAI threats arise when IoT apps–coded for distinct automation purposes but interplaying over the same home–are misused by homeowners.

Individual IoT app developers are unlikely to avoid CAI threats completely due to the lack of a predictive and global view of what apps will be installed and how they are configured by a user. As an increasing number of devices and apps are installed at a smart home, CAI threats will exacerbate. The goal of this paper is to (1) systematically categorize CAI threats; (2) propose techniques to precisely discover CAI threats; (3) evaluate the risks of identified CAI instances to assist users to make informed decisions.

In this paper, each IoT app is modeled as automation rules following a *trigger-condition-action* (TCA) paradigm. We systematically categorize CAI by considering how the action of one rule may affect the *trigger*, *condition*, and *action* of another rule, and obtain three corresponding categories of CAI threats with totally ten types. Each CAI type is encoded into a SMT (Satisfiability Modulo Theory) model which comprises a set of constraints. The constraints describe *cross-app-boundary* semantic relations, i.e., how the automation in one app interferes with that in another app.

With the SMT models of CAI threats, detecting CAI threats is converted to a theorem-proving problem, i.e., checking every pair of automation rules, along with the associated configuration, against SMT models; if a SMT model is solvable, the rule pair could cause the corresponding CAI threat. Therefore, how to extract TCA rule semantics defined by IoT apps and how to collect user configuration during app installation are two vital questions to answer for automated CAI detection. In this paper, we explore viable techniques to work with Samsung SmartThings, which at the time of research supports the largest number of IoT devices and apps. We develop a symbolic executor to perform static analyses on the AST (Abstract Syntax Tree) representations of SmartApps to extract TCA rule semantics along all paths. Moreover, to overcome the challenge that SmartThings does not provide interfaces for collecting user configuration, we exploit an app instrumentation approach to resolving it.

The proposed system HOMEGUARD interposes whenever a new app is to be installed and analyzes whether there exist CAI threats between the new app and already-installed ones. We develop a proof-of-concept prototype of HOMEGUARD.

*An early version of this paper was posted on arXiv in August 2018 [1].

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00056

Our evaluation shows that HOMEGUARD can precisely discover CAI threats from real-world SmartApps, and generate the analysis results instantly.

Our main contributions are summarized as follows:

- **A comprehensive categorization and modeling of CAI threats** – To our best knowledge, this is the first work that comprehensively categorizes CAI threats and the first work that uses symbolic constraints to model CAI threats.
- **A precise rule extractor** – We design and implement a symbolic executor for extracting rule semantics from SmartApps. Compared to code analyses in previous work [9], [12], [16], our approach is more complete and precise by deriving constraints from path-level analysis and modeling APIs residing in the opaque cloud.
- **Accurate and usable detection techniques** – Recent work [17]–[19] introduces model checking based techniques for detecting CAI, and requires users to provide safety specifications/policies. Unlike these solutions, we are the first to leverage SMT solving for CAI detection. Our detection technique does not need specification input and could comprehensively detect CAI threats hidden from specification definers (usually end users). Plus, we combine rule semantics and home-specific configuration to detect threats, reducing false alarms compared to prior work that only considers rule semantics [16], [17], [20].
- **Risk ranking** – we propose a risk ranking model to evaluate risk levels of discovered CAI threats, which takes into consideration three key factors of a CAI instance: its influence on rule execution, functionality categories of the involved rules, and security criticality of the devices being controlled. The risk ranking model assigns a risk level (*high*, *medium*, or *low*) to every CAI instance, assisting end-users to prioritize handling higher-risk threats.
- **Implementation and evaluation** – we build a prototype HOMEGUARD with a viable deployment path that does not require framework modifications of SmartThings. We evaluate the effectiveness and efficiency of HOMEGUARD over market apps. HOMEGUARD identifies 663 potential CAI threat instances in 146 real market apps. The end-to-end CAI detection incurs an averaged app installation latency of 2.7s and no runtime latency.

II. RELATED WORK

With the popularization of appified IoT platforms, security issues caused by app interference draw much research attention in very recent years. Table I illustrates the comparison of our work with related work. SIFT [21] is a safety-centric programming platform with action-conflict detection and resolution. Surbatovich et al. [20] study the security and privacy threats caused by the chained execution of IFTTT recipes. IoTA [22] introduces a process calculus for modeling IoT automation languages, and transforms the calculus to model checking for conflict detection. Soteria [17], IoTSan [18] and IoTGuard [19] employ model checking to detect action conflicts and chained execution; the CAI threat detection relies on the *correctness and completeness* of deployment-specific safety specifications input by users. In contrast, our work detects threats without relying on the expertise of end users. IoTMon [16] focuses on physical channels via which actuators and sensors interact, leading to chained execution of apps. SafeChain [23] detects privilege escalation and privacy leakage attacks caused by chained execution. None of these works performs systematic categorization of CAI threats. Like our work, iRuler [24] also uses symbolic constraints to describe different types of CAI threats. iRuler uses NLP to extract rules from IFTTT applets, but cannot handle code like SmartApps. Plus, it does not rank the detected threats.

In short, HOMEGUARD (posted on arXiv in **August 2018** [1]) is the first work that comprehensively categorizes different types of CAI threats. It is also the first work that uses symbolic constraints to precisely describe CAI threats, and the first that leverages SMT solving for threat detection.

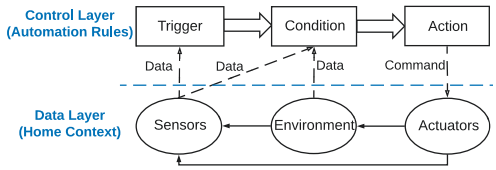


Figure 1: The home automation model.

III. HOME AUTOMATION MODEL

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

Data Layer. The data layer consists of sensors, actuators, and the environment. (1) A *sensor* may be a sensing component that measures a physical feature (e.g., the temperature), a device that reports its states (e.g., the `on`/`off` state of an outlet), or a system state (e.g., `mode`). (2) An *actuator* can be either a controllable device or a system state. An IoT *device* may be a sensor, an actuator or a combination. (3) The environment has a set of physical features such as time, temperature, illuminance, etc. The interaction among sensors, actuators and the environment is shown in Fig. 1. Sensors observe the environment, while actuators can affect sensors via device/system state (e.g., turning on a switch produces an `on` event) or via physical channels [16] (e.g., a thermostat influences a temperature sensor via temperature).

Control Layer. The control layer consists of automation *rules* defined by apps; an app usually defines one or more rules. In emerging appified home systems, the rule model follows a *trigger-condition-action* paradigm, as depicted in Fig. 1. *Trigger* subscribes to an event (e.g., television is turned on) that *activates* the execution of a rule. *Condition* is a set of constraints on home-related data (sensor readings, device/system states, time, etc.) that must be satisfied for the rule to proceed. The *difference* between *trigger* and *condition* is that a fired trigger activates the execution while

Table I: Comparison between HOMEGUARD and related work.

	Publication Date[1]	# of CAI Threat Types	Systematic Categorization?	Symbolic Threat Modeling?	CAI Threat Detection			Risk Ranking?
					Precise Semantics Extraction?	Leverage App Configuration?	No Need For Specification?	
SIFT [21]	Apr 2015	1	✗	✗	✗[2]	✓	✓	✗
Surbatovish et al. [20]	Apr 2017	1	✗	✗	✗	✗	✓	✗
IoTA [22]	Oct 2017	3	✗	✗	✗	✓	✓	✗
Soteria [17]	May 2018	3	✗	✗	✓	✗	✗	✗
IoTSan [18]	Oct 2018	2	✗	✗	✓	✓	✗	✗
IoTMon [16]	Oct 2018	1	✗	✗	✓[3]	✗	✓	✓
IoTGuard [19]	Feb 2019	3	✗	✗	✓	✓	✗	✗
SafeChain [23]	Oct 2019	1	✗	✗	✗	✗	✗	✓
iRuler [24]	Nov 2019	8	✓	✓	✗[4]	✓	✓	✗
HOMEGUARD	**Aug 2018**	**10**	✓	✓	✓	✓	✓	✓

[1] The earliest time when the work was published, including arXiv preprints. [2] Needs users to define rules on interfaces provided by the researchers.
[3] Does not extract rule conditions. [4] Natural language processing (NLP) based approaches are less precise than code analysis (Section 9 in [24]).

a condition does not; also, *condition* is optional and can be empty. The *action* is typically one or more commands issued to actuators or notifications to users.

Interaction. The data layer and the control layer interact in both directions. On the one hand, rules obtain data from sensors and the environment (e.g., time). On the other hand, rules send commands to actuators which further affect the sensors and the environment.

IV. CATEGORIZATION OF CAI THREATS

Given a pair of automation rules R_1 and R_2, we comprehensively examine how R_1 may interfere with R_2 and, accordingly, have identified three *basic* categories: *Trigger-*, *Condition-*, 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 (see **Basic Pattern** in Table II). R_1 and R_2 may or may not belong to the same app, and our HOMEGUARD system can handle both cases, so we do not distinguish the two cases for the simplicity of presentation. Next, by looking at the specific interference contexts and effects in each category (see **Auxiliary Pattern** in Table II), we identify multiple types of CAI threats, as summarized in Table II.

A. Action-Interference Threats

Two rules may operate on the same actuator, but issue conflicting commands (i.e., converse commands such as `open`/`close` or the same command such as `setOvenSetpoint` with different arguments) due to different automation purposes. When both rules are triggered by the same event and pass the condition check, conflicting commands issued by the two rules impose an *Race* on the same actuator (**A.1** in Table II); thus, the final status of the actuator turns out to be uncertain (may be a bad state), varying with factors such as the arrival order of commands and the device's communication and processing sensitivity. To validate, we ran two SmartApps on SmartThings which turn on and off a light switch respectively when a door sensor detects the door being opened. We observed a variety of results: the switch is turned on only, turned off only, turned on then off, and turned off then on.

Another Action-Interference type (**A.2** in Table II) is more subtle than **A.1** since two rules are not triggered by the same event and therefore not executed simultaneously, but perform conflicting commands. The two rules work separately but the latter rule overrides the command issued by the former one immediately or after a while, which might or might not cause a real threat. Fig. 2(a) shows an example of **A.2**.

B. Trigger-Interference Threats

A rule's action may change the home context, producing an event that triggers other rules; thus, new *covert* rules are derived from a group of explicitly defined rules. *Covert Rules* may or may not be desired by users.

Fig. 2(b) shows an example of **T.1**. A covert rule "***when** a voice command is issued **then** disarm the **cameras***" is formed. If the user perceives the safety implication and only uses the voice command when she is at home, it is not a real threat but a feature for her. However, if she is not clearly aware of the covert rule and uses the voice command while not home, a real safety threat arises.

Suppose rule R_1 triggers rule R_2 first; if R_2's action in turn has impacts on R_1's action or trigger, three special cases of Trigger-Interference, i.e., **T.2**, **T.3**, **T.4** (in Table II), can be derived. In **T.2**, R_2's action incurs a race with R_1's action on the same actuator; as a result, the execution of R_1 yields an opposite effect. In **T.3** and **T.4**, the execution of R_2 triggers R_1 such that R_1 and R_2 trigger each other in a loop; the difference between **T.3** and **T.4** is that R_1 and R_2 perform conflicting actions in **T.4** but not in **T.3**. Fig. 3(a) and 3(b) shows an example of **T.2** and **T.4**, respectively. These threats may lead to user confusion (e.g., cannot turn on the heater), device damages (e.g., due to frequent toggling), or even security and safety threats (e.g., light flashing in Fig. 3(b) causes seizures to photosensitive epilepsy sufferers [25]).

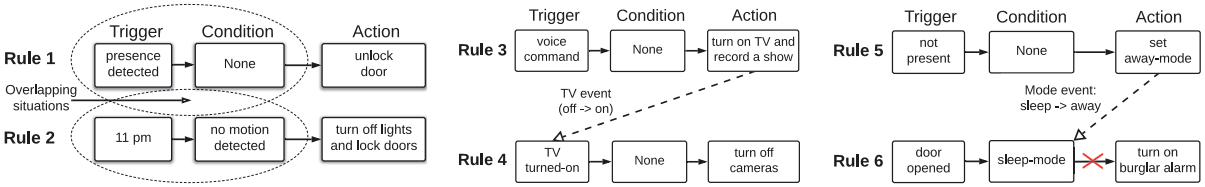
C. Condition-Interference Threats

A rule R_1's action may change the satisfaction of another rule R_2's condition and thus affect the execution of R_2, which is referred to as Condition-Interference Threats. Unlike Trigger-Interference, the action of R_1 does not necessarily trigger the execution of R_2, as R_2 has its own trigger.

Table II: Categorization of CAI threats. Let $R_i = (T_i, C_i, A_i), i = 1,2$ denote two arbitrary rules, where T_i, C_i, A_i are the trigger, condition and action, respectively. $= \neg$ denotes "conflicts with"; $\sim$ denote negation; $\mapsto$ denotes "triggers"; $\Rightarrow$ and $\not\Rightarrow$ denote "enables" and "disables", respectively. **Validation** indicates whether R_1 and R_2 actually interact when both the basic and auxiliary pattern hold, according to our validation in SmartThings: ✔: always happen, ✗: never happen, and ✔*: conditionally happen (depending on platform-specific features, see Section VII-C).

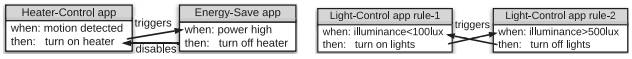
Category	Basic Pattern	Auxiliary Pattern[1]	ID	Validation	Description
Action-Interference Threats	$A_1 = \neg A_2$	$T_1 = T_2$, $C_1 \wedge C_2$	**A.1**	✓	R_1 and R_2 are executed simultaneously to perform conflict actions.
		$T_1 = T_2$, $\sim(C_1 \wedge C_2)$	–	✗	R_1 and R_2 cannot be both executed although they are both triggered.
		$T_1 \neq T_2$, $C_1 \wedge C_2$	**A.2**	✓	R_1 and R_2 may be executed within a short period to perform conflict actions.
		$T_1 \neq T_2$, $\sim(C_1 \wedge C_2)$	–	✗	R_1 and R_2 are unrelated and have no interaction.
Trigger-Interference Threats	$A_1 \mapsto T_2$	$C_1 \wedge C_2$, $\sim(A_2 \mapsto T_1)$, $A_1 \neq \neg A_2$	**T.1**	✓	R_1 triggers R_2, which does not interfere with R_1 in turn.
		$C_1 \wedge C_2$, $\sim(A_2 \mapsto T_1)$, $A_1 = \neg A_2$	**T.2**	✓	R_1 triggers R_2, which performs a conflict action and thus invalidate R_1.
		$C_1 \wedge C_2$, $A_2 \mapsto T_1$, $A_1 \neq \neg A_2$	**T.3**	✓	R_1 and R_2 trigger each other alternately.
		$C_1 \wedge C_2$, $A_2 \mapsto T_1$, $A_1 = \neg A_2$	**T.4**	✓	R_1 and R_2 trigger each other and perform conflict actions alternately.
		$\sim(C_1 \wedge C_2)$		✗	R_2 fails its condition checking and cannot be executed.
Condition-Interference Threats	$A_1 \Rightarrow C_2$	$T_1 = T_2$	**C.1**	✓*	R_1 turns a constraint in R_2's condition to true, which increases the
		$T_1 \neq T_2$	**C.2**	✓	probability of R_2 being executed.
	$A_1 \not\Rightarrow C_2$	$T_1 = T_2$	**C.3**	✓*	R_1 turns a constraint in R_2's condition to false, which decreases the
		$T_1 \neq T_2$	**C.4**	✓	probability of R_2 being executed.

[1] The auxiliary pattern of each CAI type does not conform to the basic pattern in other categories if not explicitly specified. We elide the negation constraints for conciseness.



(a) Action-Interference Threat (**A.2**) (b) Trigger-Interference Threat (**T.1**) (c) Condition-Interference Threat (**C.4**)

Figure 2: Examples of CAI threats. **Rule 1** unlocks the front door when the user arrives home; **Rule 2** turns off all lights and locks the front door at 11pm if no motion is detected; **Rule 3** uses voice commands to turn on TV and record a TV show; **Rule 4** turns off cameras while watching TV; **Rule 5** sets the home to away mode when the user leaves; **Rule 6** detects burglar break-in when the user is sleeping.



(a) **T.2** (b) **T.4**

Figure 3: Examples of other Trigger-Interference Threats: **T.2** and **T.4**

There are two types of Condition-Interference Threats: *Enabling-/Disabling-Condition Interference*, based on whether R_1's action changes R_2's condition from `false` to `true` (Enabling) or from `true` to `false` (Disabling). Fig. 2(c) shows an example of Disabling-Condition Interference (**C.4**). **Rule 5** sets "`away`" mode when a member leaves, disabling **Rule 6** to detect break-ins when another member is sleeping.

Our validation of **C.1** and **C.3** shows that they happen conditionally (marked as ✔* in Table II). In **C.1** or **C.3**, when R_1 and R_2 are triggered by the same event, whether R_1's action actually interferes with the condition checking of R_2 (although the patterns match) depends on multiple factors in the underlying platform design and runtime, e.g., processor scheduling, task scheduling, database I/O synchronization, etc. See Section VII-C for more details.

V. Threat Model and Problem Scope

Cross-App Interference occurs when multiple apps interplay, without relying on intra-app vulnerabilities. Hence, CAI threats are stealthy and cannot be handled by approaches that analyze apps individually. CAI threats may be caused for various reasons: (1) users misunderstand the full functionalities of apps based on app descriptions (which may be imprecise) [18]; (2) users lack domain knowledge to perceive subtle app interactions; (3) a global view is difficult to acquire when it comes to app installation by multiple homeowners over a long time span; and (4) users misconfigure apps, which consequently leads to interference.

This paper broadly uses *threats* to refer to all discovered interactions between rules, among which *some may be security-critical threats, some may be annoying-but-innocuous, and others may be desired by users*. Distinguishing different cases is not a completely computable problem but depends on user intention. This paper focuses on detecting all CAI threats in specific deployment, ranking their risks, and presenting the detailed results to users in a user-friendly manner (see Figure 5) for them to dictate whether or not to keep the new app and/or re-configure it.

This work presents a technique for extracting automation rules from the source code of smart apps, which is readily available. How to extract rules from compiled code or obfuscated code [26] is out of scope of this work.

VI. HomeGuard Design

In this section, we present HomeGuard. As shown in Fig. 4, modeling the automation (*rule extractor* and *config. collector*) in a smart home (Section VI-B) is the foundation for precise CAI detection. With precise modeling, HomeGuard detects CAI threats (*threat detector*) when an app is being installed *or* re-configured (Section VI-C). Moreover, HomeGuard ranks the risk of every detected CAI instance

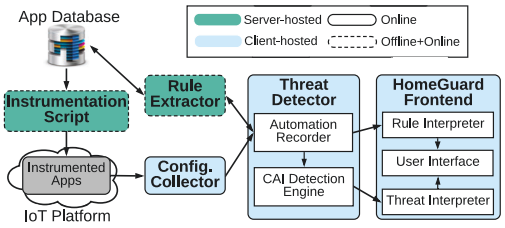


Figure 4: Architecture of HOMEGUARD.

to assist handling (Section VI-D). HOMEGUARD provides a frontend app for users to view detection results.

A. Home Automation Rule Representation

Listing 1 shows the structured rule representation format we use. It encapsulates the detailed information about a rule: (1) *trigger* contains **subject** (e.g., a certain device), **attribute**, and **constraint** (that should be satisfied); (2) *condition* comprises **data constraints** (i.e., data quantitative relations) and **predicate constraints** (i.e., boolean expressions in path conditions); (3) *action* issues **command** to control **subject** (e.g., a device), **paras.** denotes parameters of the command, and **data constraint** denotes all quantitative constraints related to the parameters; besides, **when** denotes the scheduled time and **period** indicates the repetition interval for issuing the command; by default, **when** and **period** are equal to 0, meaning that the command is issued with no delay and only once, respectively. In Section VI-B, we will use an example to show how a rule is extracted from the code in Listing 1 and is represented in Table III.

Listing 1: The rule representation format

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

B. Home Automation Modeling

Home automation modeling includes three aspects: *rule semantics extraction* (Section VI-B1), *configuration information collection* (Section VI-B2) and *rule assembly* based on the two kinds of information (Section VI-B3). Given an app, its execution varies with distinct configurations; for example, users may bind different devices to the app or specify different values for variables. Hence, configuration information collection is vital. Note that Home Automation Modeling is platform-specific because different smart homes may employ different IoT platforms which support distinct programming languages and APIs. We perform code analysis and code instrumentation to extract rule semantics and collect configuration from apps, respectively. To prove this concept, we concretely implement the techniques on Samsung SmartThings platform. We use a SmartApp snippet

Listing 2: Code snippet of `ComfortTemp`. Irrelevant lines (e.g., metadata definition, UI-related sections) are omitted.

```
input "mSensor", "capability.motionSensor", title:                    1
    "Which motion sensor?"
input "tSensor", "capability.temperatureMeasurement"                  2
input "threshold", "number", title: "Lower than?"                     3
input "fan", "capability.switch", title: "Which fan?"                 4
input "ac", "capability.switch", title: "Which A/C?"                  5
def installed() {subscribe(mSensor,"motion",actHandler)}              6
def updated(){                                                        7
   unsubscribe(); subscribe(mSensor,"motion",actHandler)}             8
def actHandler(evt) {                                                 9
   def t=tSensor.currentValue("temperature")                          10
   if ((evt.value=="active")&&(t<threshold))                          11
      adjustTemp() }                                                  12
def adjustTemp() {                                                    13
   ac.on()                                                            14
   if(fan.currentSwitch=="on") fan.off() }                            15
```

(see Listing 2) that turns off fans and turn on A/C when motion is detected and the room temperature is below $70°F$ as an example to help present our techniques.

1) Rule Semantics Extraction: Prior approaches [9], [27] insert runtime logging logic to collect *context* information when sensitive commands are issued. Such dynamic approaches do not work for our purpose as they only explore the paths that *have been executed*, while our goal is to extract *all* the rules *before* they are executed. Approaches [12], [13], [16] search the Abstract Syntax Tree (AST) of SmartApps to look for information of interest (e.g., the trigger event, the attribute, and the action) *without tracking the data flows*, so they cannot fully retrieve the constraint information due to variable assignments, nested branches, API calls, etc., which is critical for precise CAI threat detection.

In order to conquer the setback above and extract rules from an app *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 *condition*. To this end, the following *questions* or *technical challenges* are addressed.

Path search strategy. A well-known limitation about symbolic execution is poor scalability due to path explosion. However, IoT apps are much smaller than applications in other platforms (e.g., desktop, mobile) and have limited number of paths, so a simple *depth-first path search strategy* works well without encountering path explosion.

Symbolic inputs. Data whose values are not dependent on other data are handled as *symbolic inputs*. In SmartApps, they include device references, device attribute values, device events, user input, constant values, and API return values (see **API modeling** below). To achieve automated 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 device connected to SmartThings) and user inputs (variables whose values are specified by users during app configuration), 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 constant value are also labeled as symbolic inputs. Consider the example in Listing 2, the devices references (`mSensor`, `tSensor`, `fan`, `ac`), user input (`threshold`), and return values of the API call at Line 10 and 15 are identified as symbolic inputs.

Analysis entry points and sinks. In our implementation, the analysis entry points include the lifecycle methods, e.g., `installed`, `updated`. 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 [28] and 21 SmartApp APIs.

Generating Control-Flow Graph (CFG). We adapt the approach in [9] to generate a control-flow graph from the AST of each app. Our goal is to model the *trigger-condition-action* structure of a rule. A rule with a trigger usually starts from an event subscription method `subscribe(dev,attr,hndl)` (typically invoked in the analysis entry point methods). A `subscribe` call defines that when an event (device `dev`'s attribute `attr` changes) occurs, the handler method `hndl` will be invoked. Therefore, each `subscribe` represents a rule trigger. Then we trace into the invoked handler `hndl` to identify sinks along the execution path. The path branches at conditional statements (e.g., `if` or `switch`) so we may reach different sinks, which are extracted as rule actions; the boolean expressions within the condition statements along the execution path from an entry point to a sink are used to construct the rule condition for that sink. The corresponding trigger, condition, and action are assembled into a rule.

Constraints for the rule *trigger* and *condition*. The `subscribe` method yields to a rule trigger by subscribing to an event (e.g., Line 8 in Listing 2). If a conditional statement follows along the execution path to compare the event's value (e.g., Line 11 in Listing 2), the comparison is regarded as part of the trigger constraint; otherwise, the trigger is only a state change and has no constraint.

We track all data and predicate constraints along every execution path from the entry point to sinks and attach them (excluding the trigger constraint) to rule conditions. We establish data constraints from value assignment statements. Specifically, we modify the compiler to handle the 38 expression types defined in Groovy's documentation [29]. On the other hand, we also build predicate constraints from conditional statements, i.e., each boolean expression in an `if` statement or each case expression in a `switch` statement is translated into a constraint. We handle the ternary expressions by breaking each of them into two branches.

API modeling. A main challenge in this work is to deal with the closed-source APIs provided by SmartThings. We first model the 10 SmartApp APIs that can schedule the method executions based on their arguments and functionalities. For instance, `runIn(delay, method)` delays the execution of `method` by a specified time delay. We attach the delay information to `runIn` and continue to trace into the scheduled method to identify sinks. The successive sinks are also attached with the delay. The delay is eventually inserted into the `when` field (see Listing 1) of the extracted rule.

Table III: One of the extracted rules from the code in Listing 2.

Trigger	Condition	Action
subject: mSensor **attribute**: motion **constraint**: mSensor.motion ==active	**data constraints**: t = tSensor.temperature, tSensor.temperature=#DevState **predicate constraints**: t < threshold, fan.switch == on	**subject**: fan **command**: off **paras**: [ ] **data constraints**:[ ] **delay**: null **when**: null **period**: null

To handle APIs involved in constraint construction, we model objects, methods and object property accesses by reviewing the SmartThings developer documentation [30]. The return values of these methods and object property accesses 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 from expressions that contain API calls.

Compiler customization. To build the symbolic executor, we implement a compilation customizer and add it to the compiler configuration, which is supported by Groovy to allow developers to modify the compilation process. We choose to work at the semantic analysis phase where the compiler creates a class node for each element (variable, method, expression, statement), and we write a set of visit methods that follow the generic Visitor pattern [31] to specify how the compiler processes these class nodes.

2) Configuration Information Collection: Recall that we identify symbolic inputs from `input` methods, which are rendered as graphical interface elements by SmartThings for users to configure apps. To detect CAI threats in a specific home, we need to know configuration information, i.e., symbolic input values. Configuration is not available until apps are installed; thus, *it cannot be obtained through static code analysis*. In other words, so far the extracted rule semantics only contain variable names rather than concrete values. For example, in Table III a reference "`mSensor`" rather than the globally unique identifier of the granted device is extracted and "`threshold`" is not concretized to a value (e.g., $70°F$). Without such information, the extracted rules are incomplete and CAI detection becomes imprecise.

Solution. There are no APIs available to query configuration information from SmartThings. To address the problem, we collect configuration information by instrumenting SmartApps. Code instrumentation has been used in previous

Listing 3: Code snippet showing how the app in Listing 2 is instrumented. Unaltered lines are omitted. Lines 3–7 and 11–19 are the inserted code.

```
//...                                                          1
def updated() {                                               2
    // collecting information                                 3
    def appname = "ComfortTemp"                               4
    def devices = [[devRefStr:"mSensor",                      5
         devRef:mSensor], [devRefStr:"tSensor",
         devRef:tSensor], [devRefStr:"fan",
         devRef:fan], [devRefStr:"ac", devRef:ac]]
    def values = [[varStr:"threshold", var:threshold]]        6
    collectConfigInfo(appname, devices, values)               7
    //...                                                     8
}                                                             9
//...                                                         10
def collectConfigInfo(appname, devices, values) {             11
    def params //Set the cloud messaging server, which        12
         relays messages to HomeGuard frontend app
    def config=["appname":appname, "devices":[:],             13
         "variables":[:]]
    devices.each { dev ->                                     14
      config["devices"][dev.devRefStr]=dev.devRef.getId()}    15
    values.each { val->                                       16
      config["variables"][val.varStr]=val.var }               17
    sendConfig(params, config) // Send the                    18
         configuration to the relay server by calling
         API httpPoseJson
}                                                             19
```

researches [9], [12], [19], [27]. In our case, instrumentation is only to gather configuration during app installation, so it introduces negligible complexity and overhead. We automate instrumentation with a Groovy script. Listing 3 shows the instrumented version of the app in Listing 2.

The lifecycle method `updated` is invoked when the app is installed or re-configured. The `appname` can be obtained from metadata. In each item of the lists `devices` and `values`, `devRefStr` and `varStr` are variable names defined in `input` methods, and `devRef` and `var` denote real values specified by users. The Groovy script reuses the code for symbolic input identification (Section VI-B1) to identify `appname`, `devRefStr`, and `varStr`. The `collectConfigInfo` method (Line 11) assembles a JSON object `config` that stores the app name, mappings between each device variable name `devRefStr` and the unique 128-bit ID of the configured device (`devRef.getId()`), and mappings between each variable name and its specified value. `sendConfig` sends the collected information from the cloud (where the SmartApp runs) to the user's smartphone via a relay server.

3) Rule Assembly: We combine the rule semantics with configuration from the same app to complete rule extraction. Upon receiving configuration, we parse key-value pairs in `config` and construct a constraint for each pair; that is, each key `dev` in `config["devices"]` or each key `var` in `config["variables"]` generates a constraint in the form of `dev = config["devices"][dev]` or `var = config["variables"][var]`, respectively. The derived constraints are inserted to certain fields of all rule semantics defined by the app; specifically, the constraints are appended to the constraint field of rule trigger, and the data constraints fields of condition and action, respectively. Thus, all the device references and variables in the extracted rule semantics are concretized (e.g., the device IDs of `mSensor`, `tSensor`, `fan` and the variable values of `threshold1` in Table III) , and the rules become complete.

C. CAI Threat Detection

Whenever a user installs a new app, HOMEGUARD detects CAI threats between each rule from the new app and every existing rule and between different rules in the new app to see whether any pair meets patterns shown in Table II under the current configuration. In general, the pattern evaluation has two steps: *candidate filtering* and *overlapping-condition detection*. Candidate filtering is performed first to avoid unnecessary computation for overlapping-condition detection.

1) Detecting Action-Interference Threats: Action Interference is commutative, so the detection of rules R_1 and R_2 is performed once. *Candidate filtering* verifies whether R_1 and R_2 satisfy two constraints: $A_1 = \neg A_2$ and $T_1 = T_2$. To evaluate $A_1 = \neg A_2$, we first examine if the actions of R_1 and R_2 issue contradictory commands, or issue the same command with conflicting parameters; either situation indicates $A_1 = \neg A_2$. If $A_1 = \neg A_2$ holds, the evaluation proceeds; otherwise, R_1 and R_2 do not have Action Interference. Next, if R_1 and R_2 have the same trigger ($T_1 = T_2$), they are an **A.1** candidate; otherwise, they are an **A.2** candidate.

To determine whether the candidate really causes an Action-Interference threat, we need to know if they could be executed under the same condition, i.e., *overlapping-condition detection* ($C_1 \wedge C_2$). The overlapping-condition detection is transformed into a *constraint satisfaction* problem by merging all constraints in the conditions of the two rules. If the problem is solvable ($C_1 \wedge C_2$ holds), the candidate is confirmed to cause an **A.1** or **A.2** threat. Our implementation chooses the Java Constraint Programming (JaCoP) library as the solver, which is efficient and open-source.

2) Detecting Trigger-Interference Threats: Trigger-Interference is not commutative so the detection of two rules R_1 and R_2 should be performed in both directions. Without loss of generality, we discuss one direction here. There are two ways that R_1's action triggers R_2 ($A_1 \mapsto T_2$): (a) R_1's action (e.g., turning on a switch) causes a state change of an actuator such that this event (`off`→`on`) triggers R_2; (b) the actuator controlled by R_1 (e.g., turning on a heater) changes an environment channel (e.g., temperature) that changes a sensor (e.g., temperature sensor) measurement subscribed by R_2.[1] First, we follow the two ways above to determine if $A_1 \mapsto T_2$ holds. If $A_1 \mapsto T_2$ holds, we further evaluate if $A_2 \mapsto T_1$ and $A_1 = \neg A_2$ hold; based on the result, R_1 and R_2 are considered as a candidate of **T.1**, **T.2**, **T.3** or **T.4** depending on which auxiliary pattern in Table II the pair satisfies. Next, the overlapping-condition

[1] Determining case (b) precisely requires detailed knowledge about the *in-situ* interactions between actuators and sensors. The techniques for learning the knowledge, e.g., data mining, are out of the scope of this paper.

Table IV: The effects of CAI threats on rule actions. See Table II for CAI definitions and notations.

CAI Type	A.1	A.2	T.1	T.2	T.3	T.4	C.1	C.2	C.3	C.4
Action A_1	−	−		−	+	○				
Action A_2	−	−	+	+	+	○	+	+	−	−

detection result in Action-Interference detection is reused to finalize the detection: if $C_1 \wedge C_2$ holds, a candidate is confirmed to cause a Trigger-Interference threat.

3) Detecting Condition-Interference Threats: The detection of Condition-Interference is anticommutative and we present one direction. To detect whether R_1 has Condition-Interference with R_2, we first evaluate whether $A_1 \Rightarrow C_2$ or $A_1 \not\Rightarrow C_2$ holds. Similar to Trigger-Interference threats, there are two ways that R_1 can affect R_2's condition: (1) R_1 changes the state of an actuator, which changes the satisfaction of R_2's condition directly (e.g., R_1 turns on a heater and R_2 checks if the heater's state is `on`); and (2) R_1 affects an environment channel by controlling an actuator, which changes the satisfaction of R_2's condition (e.g., R_1 turns on the heater and the condition of R_2 involves the room temperature). We verify if R_1's action affects R_2's condition in either way; if so, the detection proceeds.

Next, we distinguish whether R_1 enables ($A_1 \Rightarrow C_2$) or disables ($A_1 \not\Rightarrow C_2$) R_2's condition. To this end, we create an *effect constraint* to denote the effect of A_1. For instance, if A_1 locks a door (`door1`), we generate a 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.temp>=T`. We then merge the effect constraint with R_2's condition and solve the new constraint satisfaction problem. If the problem is solvable, $A_1 \Rightarrow C_2$ holds, and the two rules are mapped to **C.1** or **C.2** depending on whether $T_1 = T_2$ holds; otherwise, $A_1 \not\Rightarrow C_2$ holds, and the two rules are mapped to **C.3** or **C.4** in the same way.

D. Risk Ranking

As discussed in Section V, the outcome of CAI threats ranges from security threats to user-desired features. Notifying users of every CAI instance equally increases user efforts and might annoy users, making them tend to underestimate or even ignore the notifications. We propose a user-friendly risk ranking model to help users evaluate notifications.

Observation 1: In general, each CAI threat type (Table II) has specific impacts on the involved rules, i.e., promoting, suppressing or looping the rule actions. For example, in **A.1**, rules R_1 and R_2 perform conflicting actions (A_1 and A_2, respectively) on the same actuator, whose final state thus unpredictably violates the intention of either R_1 or R_2; in other word, **A.1** suppresses the actions of R_1 and R_2 (we consider that both actions are suppressed due to the unpredictability). Likewise, in **T.1**, R_1's action A_1 triggers the execution of R_2 to take action A_2; i.e., **T.1** promotes A_2. In **T.4**, R_1 and R_2 trigger each other and perform conflict actions alternately on the same actuator (e.g., turn on and off a switch); in this way, **T.4** loops A_1 and A_2. We use $+$, $-$, $\circ$ to denote promoting, suppressing and looping, respectively, and summarize the effect of all CAI threat types on the involved rule actions in Table IV.

Observation 2: The risk implication of promoting, suppressing or looping a rule's action depends on both the functionality of the rule and the sensitivity of the action. Consider a safety-critical rule R_1 ("*unlock the door and open the window when smoke is detected*") and a non-safety rule R_2 ("*open the window when air quality is low*"). Given a CAI threat that *prevents* opening the window, it imposes a higher risk to R_1's action A_1 than to R_2's action A_2 since users might install R_1 for security or safety functionality but install R_2 for comfort or convenience. On the other hand, while *promoting* opening the window or unlocking the door has a low impact on the functionality of both R_1 and R_2, it is risky in common cases based on the safety nature of windows and doors. Therefore, we take both *rule functions* and *device control sensitivity* into consideration for evaluating the risk of a CAI instance.

We formally define the risk of a CAI threat instance $I(type, R_1, R_2)$ as $Risk(I)$, where $R_1 = (T_1, C_1, A_1)$ and $R_2 = (T_2, C_2, A_2)$ are the two involved rules (see Table II for notations). $Risk(I)$ has three possible values {1, 0, -1} (interpreted as {high, medium, low}). From Observation 1 and 2, we know that a CAI instance has distinct effects and therefore imposes different risks on the involved rules. The risk $risk_i$ on each rule R_i is calculated as a function of CAI effect e_i on R_i (see Table IV), the functionality category c_i of rule R_i, the device dev_i and command cmd_i of R_i's action:

$$risk_i = \max(M_1(e_i, c_i), M_2(e_i, dev_i, cmd_i))$$

where $M_1(e_i, c_i)$ computes a risk value by factoring the *rule function* of R_i and $M_2(e_i, dev_i, cmd_i)$ computes another risk value by factoring the *device control sensitivity* of R_i. In SmartThings, each app is assigned a `category` field (e.g., *Safety & Security*, *Convenience*, *Energy Management*) in its source code that specifies its functionality. In our risk model, the functionality category c_i of a rule R_i is "safety" if the `category` of the app that defines R_i is "Safety & Security"; otherwise c_i is "non-safety". Thus, $M_1(e_i, c_i)$ produces a risk value by looking up the pre-defined mapping in Table V, which shows how the M_1 risk level is determined by the effect of the threat on a rule and the rule category.

To model the general sensitivity of controlling a device dev_i with command cmd_i, we analyze the 146 official SmartApps given that the automation rules in these apps provide information about how IoT devices are supposed to be controlled for specific functionalities. For example, automatically locking a door is typically considered a safe operation, since "safety" rules usually lock (rather than unlock) a door to ensure safety. Thus, `(+)lock.lock()` (promote locking a door lock) has a

Table V: The risk level of a CAI threat instance I on a rule R_i, given the effect e_i of I on R_i and the functionality category c_i of R_i.

Category	Effect(+)	Effect(−)	Effect(o)
safety rules	low	high	high
non-safety rules	low	medium	medium

low risk, `(-)lock.lock()` (suppress locking a door) has a high risk and `(o)lock.lock()` (alternately lock and unlock a door) has a high risk. Automatically turning on a light is regarded as a low-risk operation since "non-safety" rules usually turn on lights for convenience. Thus, `(+)light.on()` has a low risk, `(-)light.on()` has a medium risk and `(o)light.on()` has a medium risk (annoying users). Note that SmartThings uses *capabilities* to model different device types and commands. Based on the above idea, we build a general *risk knowledge model KM* for each capability-supported command under different CAI effects (i.e., $+$, $-$, o), by analyzing all rules in the 146 SmartApps as shown in Algorithm 1; in total, we obtain the *KM* of 46 capability-supported commands (Table VI). Next, $M_2(e_i, dev_i, cmd_i)$ uses dev_i and cmd_i to get a capability-supported command *capCmd* and then use *capCmd* and e_i to retrieve a risk value from *KM*.

Algorithm 1: The algorithm for extracting device control sensitivity under different CAI effects from SmartApps

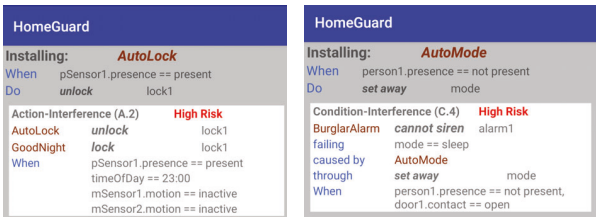
```
   Input  : Apps ← the source code of all SmartApps
   Output: Risk knowledge model of capability-supported commands KM
1  foreach app ∈ Apps do
2      Rules ← ExtractRuleSemantics (app)
3      catetory ← ExtractCategory (app)
4      foreach rule ∈ Rules do
5          capability ← FindCapability(rule.action.device)
6          cmd ← rule.action.command
7          capCmd ← Concatenate(capability, cmd)
8          oppCapCmd ← FindOppositeCmd(capCmd)
9          if category is "Safety & Security" then
10             count[capCmd]['+']['low']++
               count[capCmd]['-']['high']++
               count[capCmd]['o']['high']++
               count[oppCapCmd]['+']['high']++
               count[oppCapCmd]['-']['low']++
               count[oppCapCmd]['o']['high']++
11         else
12             count[capCmd]['+']['low']++
               count[capCmd]['-']['medium']++
               count[capCmd]['o']['medium']++
13 foreach capCmd ∈ count.keys() do
14     foreach e ∈ {'high', 'medium', 'low'} do
15         KM[capCmd][e]= max(count[capCmd][e]['high'],
           count[capCmd][e]['medium'],
           count[capCmd][e]['low'])
```

E. HomeGuard Frontend App

HOMEGUARD frontend bridges the detection system and smart home users. A *rule interpreter* component translates newly installed rules into a human-readable form and displays them via a *user interface*, such that users can check if the rules match their intention. A *threat interpreter* displays the detected CAI threats to users in a readable manner, allowing them to decide whether to uninstall some app(s) or whether to re-configure the involved app(s). Fig. 5 shows screenshots of interfaces provided by the frontend app.

Table VI: The output *KM* of running Algorithm 1 on 146 SmartApps, showing the risk level of a CAI threat instance I on capability-supported commands under different effect e_i. H: high, M: medium, L: low. Partial results (out of 46) are listed due to space limits.

Capability.command	Effect(+)	Effect(−)	Effect(o)
alarm.off	H	L	H
alarm.siren	L	H	H
light.off	L	M	M
location.setLocationMode	L	M	M
lock.lock	L	H	H
lock.unlock	H	M	H
switch.on	L	M	M
valve.close	L	H	H
valve.open	H	L	H



(a) (b)

Figure 5: Screenshots that show the HOMEGUARD frontend app interface when (a) installing `AutoLock` when `GoodNight` has been already installed; (b) installing `AutoMode` when `BurglarAlarm` has been installed. `AutoLock`, `GoodNight`, `AutoMode` and `BurglarAlarm` are SmartApps that define **Rule 1, 2, 5, 6** in Fig. 2, respectively.

VII. EVALUATION

HOMEGUARD is evaluated on a Dell desktop (rule extraction) with 3.4GHz Intel Core i7 CPU-6700 and 8GB memory and a Samsung Galaxy S8 smartphone (CAI threat detection and configuration collection) with Android OS 8.1.0. We study whether CAI threats can be identified by HOMEGUARD from real market apps.

A. Test Cases

We create two test suites. The first set is used to explore the status *in quo* of CAI threats in market apps and evaluate the performance of HOMEGUARD in a large scale. The second set is a subset of the first one and used to validate the detected CAI threats in a real-world environment.

Market apps. We collect 146 out of 182 SmartApps from the public repository [32], removing 36 web service SmartApps which do not implement any rules but just expose web endpoints for device or service integration [30]. We use this set to (1) exhaustively identify CAI threats in market apps and learn the distribution of risk levels among the discovered instances, and (2) evaluate the performance of HOMEGUARD at a large scale. In this experiment, it is impossible to iterate over all configuration possibilities without concrete user input; instead, we confine the detection based on the following configuration situations that CAI detection really cares about: (1) whether two apps work with the same device(s) (if they request the same device type) and (2) whether one app really affects another one through physical channels (for example, turning on/off a light might

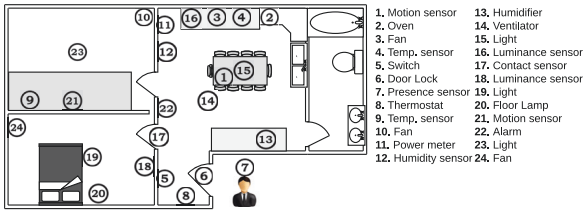
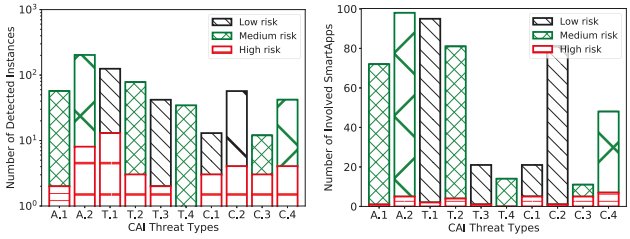


Figure 6: Devices and their layout in the real-world deployment.



(a) (b)

Figure 7: Statistics of detection results on 146 SmartApps: (a) the number and risk ranking distribution of detected instances for each CAI threat type; (b) the number and risk ranking distribution of the SmartApps that are vulnerable to each CAI threat type. See Table II for the threat acronyms.

or might not affect the measurement of an illuminance sensor, based on their positions). Thus, we address the infinite configuration issue by enumerating the binary answers (YES or NO) to the two configuration questions above.

Real-world deployment for validation. To validate the detected CAI threats by HOMEGUARD in real deployments, we select 18 market apps that are found cause threats from the second test case and configure them to work with 24 typical IoT devices (Fig. 6); these apps generate all CAI threat types (see Table VII for the selected apps, app-device bindings, and CAI threats to validate). We run these apps to verify the detection results of HOMEGUARD.

B. Correctness of Rule Extraction

We first evaluate *rule extractor*'s ability to extract rule semantics from the test suite. We manually review the code and record the rules in the first test suite. To avoid human errors, we also run these apps with simulated devices to verify the correctness. Finally, we obtain 1107 rules in total. The manual analysis results are used as *ground truth*.

We encountered several exceptions due to a lot of unforseen code dynamics and variations. For example, `Feed My Pet` uses `device.petfeedershield` in the `input` method instead of a regular capability; `Camera Power Scheduler` uses a public API `runDaily` which was not documented by SmartThings. We fine-tune the rule extractor by, e.g., adding the nonstandard device types into the capability list and modeling the undocumented APIs we encountered. Eventually, the rule extractor extracts all rules from the market apps precisely and completely.

To test the rule extraction speed, we extract rules from 146 market apps and get an average execution time of 1341ms per app. Rule extraction can be performed only once for market apps and the result can be shared via public databases, so it is a one-time effort and can be done offline.

C. CAI Threat Detection

Detecting threats from market apps. We perform CAI threat detection over the 146 market apps in the second test set and record the results. We identify 663 CAI threat instances in total and find that 101 out of 146 apps are susceptible to at least one type of CAI threat. The statistics of the detection results and vulnerable apps are shown in Fig. 7. Fig. 7(a) shows that the total instance number and the high/medium/low-risk instance number over each CAI type have different distributions. Among all CAI types, **A.2** has the most instances while **T.1** has the most high-risk instances. From Fig. 7(b), we can see that fewer apps are involved in high-risk instances than those in medium/low-risk instances. Therefore, the risk ranking reduces user burden significantly on making decisions if they are only concerned about high-risk threats and also provides flexibility for advanced users to eliminate problematic app interactions by also looking at medium- and low-risk threats.

Validation of the CAI threats detected from market apps. We validate CAI threats in real-world settings; specifically, we bind 24 typical devices (Fig. 6) to 18 martket apps selected from involved market apps (see Table VII for details). By experiments, we confirm that all the listed threats, except for Set 7, occur as indicated by our detection results. In **C.1** (or **C.3**), two rules R_1 and R_2 are triggered simultaneously; thus, whether R_1's action that leads to SmartThings updating its database actually interferes with R_2's condition checking cannot be determined by pattern-proving but depends on how SmartThings handles the execution of simultaneously triggered apps, which is a blackbox to us.

Our observation is that **C.1** in Set 7 does not occur but **C.1** and **C.3** in Set 9 always occur. In `Forgiving Security`, the condition checking (if home is in "Away" mode) is deferred by a scheduling API `runIn(delay, method)`, allowing `Rise and Shine`'s changing mode to take effect before `method` (where condition checking is performed) is called. We verify the influence of `runIn(delay, method)` by (1) setting `delay` to 0 and (2) deleting `runIn` and instead running `method` without delay. **C.1** and **C.3** always occur in case (1) but never occur in case (2) (same as Set 7). The result shows that **C.1** and **C.3** always occur when the condition checking (which may be interfered with) is scheduled by the platform runtime in a waiting queue and otherwise never occur. Thus, theorem-proving based CAI threat detection could get rid of some false positives by taking this platform-specific feature into consideration when detecting **C.1** and **C.2**.

Detection speed. To evaluate the efficiency of CAI detection, we test the averaged execution time for detecting each CAI threat type between two rules on a Samsung

Table VII: The apps and app-device bindings for constructing CAI threats.

App Name	Rule and Configuration	Set #	CAI Type
`CurlingIron` `Virtual Thermostat`	When motion (1) detected, turn on oven (2) and fan (3) for 30 minutes. When motion (1) detected, if temperature (4) is lower than 72°F, turn off fan (3).	1	**A.1**
`NFCTagToggle` `LockItWhenILeave`	When the user touches on mobile app, toggle switch (5) and toggle door lock (6). When presence sensor (7) becomes "not present", lock door (6).	2	**A.2**

`CurlingIron`
`SwitchChangesMode`
`MakeItSo`
	When motion (1) detected, turn on oven (2) and fan (3).
	When oven (2) is turned on, set home to "party" mode.
	When changed to "Party" mode, unlock door (6) and turn on thermostat (8).
		3	**T.1**

`It'sTooHot` `EnergySaver`	When temperature (9) exceeds 80°F, turn on fan (10). When power usage (11) exceeds 3000 W, turn off fan (10).	4	**T.2**
`SmartHumidifier` `HumidityAlert!`	When humidity (12) is below 30%, turn on humidifier (13); when humidity (12) exceeds 50%, turn off humidifier (13). When humidity (12) exceeds 50%, turn on ventilator (14); when humidity (12) is below 30%, turn off ventilator (14).	5	**T.3**
`LightUptheNight`	When illuminance (16) exceeds 50 lx, turn off light (15); when illuminance (16) gets below 30 lx, turn on light (15).	6	**T.4**
`Brighten Dark Places` `LetThereBeDark`	When door (17) is opened, if illuminance (18) is below 10 lx, turn on light (19). When door (17) is opened, turn off lights (20); when door (17) is closed, restore the state of lights (20).	7	**C.1**
`Forgiving Security` `Scheduled Mode Change`	When motion sensor (1) or (21) becomes "active", if the home is in "Away" mode, siren alarm (22). Set home to "Away" mode at 10 am and set home to "Night" mode at 6pm.	8	**C.2**, **C.4**
`Forgiving Security` `Rise and Shine`	When motion sensor (21) becomes "active", if home is in "Work" mode, turn on light (23) after 1 second. When motion sensor (21) becomes "active", set home to "At-Home" mode ("Work" mode).	9	**C.3 (C.1)**

`GoodNight`
`Once a Day`
`MakeItSo`
	When motion (1)(21) detected, if switches (2)(3)(5)(10)(13)(14)(15)(19)(20)(23)(24) are all off, set home to "sleep" mode.
	Turn on fan (24) at 11 pm and turn off fan (24) at 12 am.
	When changed to "sleep" mode, lock door (6).
		10	**C.4**

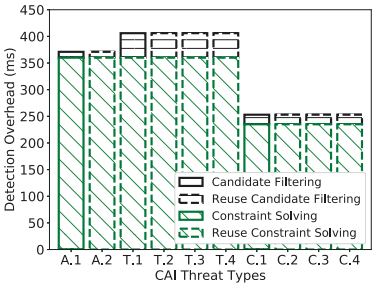


Figure 8: CAI detection overhead for a pair of rules. Green dotted lines mean the constraint solving for detecting **A.2**, **T.1**,**T.2**,**T.3**,**T.4** threats can reuse the solving result of **A.1** and the constraint solving for **C.2**, **C.3**, **C.4** can reuse that of **C.1**. The threat acronyms are defined in Table II.

Galaxy S8 smartphone. As shown in Fig. 8, the most time-consuming operation is constraint solving. The constraints solving overhead in Condition-Interference Threats is lower since the involved number of constraints is about half of that in Action-Interference Threats. 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 across detecting different threats. For an arbitrary pair of rules, the maximum total time for detecting all CAI threats is 671 ms. The actual detection time is usually much shorter since two rules rarely fit all threat patterns and may fail partial or all candidate filtering operations; thus, constraint solving overhead is avoided.

VIII. Discussion

User effort. Users need extra operations to download instrumented apps before installation. A mitigation solution is to automate this process by running a script. We have demonstrated this solution by developing a Python script with `Selenium` webdriver to automatically obtain the source code from SmartThings Web IDE [33], run the instrumentation script (in Section VI-B2), and install instrumented apps in the web IDE. Users only need to provide the SmartThings IDE log-in account to the script.

Multi-Platform Applicability. The rule extractor and configuration collector in HomeGuard are platform-specific since platforms use different programming languages and APIs. When source code [34] or bytecode [35] is available, engineering effort for code analysis and instrumentation could be made to support another platform. Besides apps, some platforms (e.g., IFTTT) define rules on mobile or web user interfaces (UIs). Rules can be extracted by crawling free texts from UIs and parse the texts with natural language processing (NLP) techniques [13], [24], [36].

IX. Conclusion

We comprehensively categorized CAI threats, and designed and built a system HomeGuard to address the problem. HomeGuard 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, without needing users to specify security goals. 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 and risk ranking on the user's smartphone. We evaluated HomeGuard using real SmartApps from the app store and validated the detection results in real-world environments. The evaluation results show that HomeGuard is effective, efficient, and precise.

Acknowledgment

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-1828363, CNS-1564128, CNS-1824440 and CNS-1856380. The authors would like to thank the anonymous reviewers and our shepherd, Dr. Yennun Huang.

REFERENCES

[1] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," *arXiv preprint arXiv:1808.02125*, August 2018.

[2] "SmartThings," https://www.smartthings.com/, 2018.

[3] "Apple HomeKit," https://www.apple.com/ios/home/, 2019.

[4] "IFTTT," https://ifttt.com, 2017.

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

[6] R. Xu, Q. Zeng, L. Zhu, H. Chi, X. Du, and M. Guizani, "Privacy leakage in smart homes and its mitigation: IFTTT as a case study," *IEEE Access*, vol. 7, pp. 63 457–63 471, 2019.

[7] H. Chi, Q. Zeng, X. Du, and L. Luo, "PFirewall: Semantics-aware customizable data flow control for home automation systems," *arXiv preprint arXiv:1910.07987*, 2019.

[8] Q. Zeng, J. Su, C. Fu, G. Kayas, L. Luo, X. Du, C. C. Tan, and J. Wu, "A multiversion programming inspired approach to detecting audio adversarial examples," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019*.

[9] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[10] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, and J. Kim, "FACT: Functionality-centric access control system for iot programming frameworks," in *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies (SACMAT), 2017*.

[11] A. Rahmati, E. Fernandes, K. Eykholt, and A. Prakash, "Tyche: Risk-based permissions for smart home platforms," *arXiv preprint arXiv:1801.04609*, 2018.

[12] 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.

[13] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "HoMonit: Monitoring smart home apps from encrypted traffic," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2018*.

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

[15] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in iot apps," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2018*.

[16] W. Ding and H. Hu, "On the safety of iot device physical interaction control," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2018*.

[17] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in *Usenix Security Symposium*, 2018.

[18] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "IotSan: fortifying the safety of iot systems," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2018*.

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

[20] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes," in *Proceedings of the 26th International Conference on World Wide Web (WWW), 2017*.

[21] C.-J. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu, "SIFT: building an internet of safe things," in *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN), 2015*.

[22] J. L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, and M. Sridharan, "IoTA: a calculus for internet of things automation," in *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2017*.

[23] K.-H. Hsu, Y.-H. Chiang, and H.-C. Hsiao, "SafeChain: Securing trigger-action programming from attack chains," *IEEE Transactions on Information Forensics and Security*, 2019.

[24] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2019*.

[25] 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.

[26] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized android application repackaging detection using logic bombs," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2018*.

[27] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

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

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

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

[31] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *IEEE Computer Software and Applications Conference (COMPSAC)*, 1998.

[32] "SmartThings public github repository," https://github.com/SmartThingsCommunity/SmartThingsPublic, 2018.

[33] "SmartThings Groovy IDE," https://graph.api.smartthings.com/, 2018.

[34] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2008*.

[35] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of java bytecode," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010.

[36] I. Hwang, M. Kim, and H. J. Ahn, "Data pipeline for generation and recommendation of the iot rules based on open text data," in *the 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 2016.