Scalable Analysis of Interaction Threats in IoT Systems

Mohannad Alhanahnah*†
mohannad@huskers.unl.edu
University of Nebraska–Lincoln
Computer Science and Engineering
Lincoln, Nebraska, USA

Clay Stevens*
clay.stevens@huskers.unl.edu
University of Nebraska–Lincoln
Computer Science and Engineering
Lincoln, Nebraska, USA

Hamid Bagheri
bagheri@unl.edu
University of Nebraska–Lincoln
Computer Science and Engineering
Lincoln, Nebraska, USA

ABSTRACT

The ubiquity of Internet of Things (IoT) and our growing reliance on IoT apps are leaving us more vulnerable to safety and security threats than ever before. Many of these threats are manifested at the interaction level, where undesired or malicious coordinations between apps and physical devices can lead to intricate safety and security issues. This paper presents IotCom, an approach to automatically discover such hidden and unsafe interaction threats in a compositional and scalable fashion. It is backed with automated program analysis and formally rigorous violation detection engines. IotCom relies on program analysis to automatically infer the relevant app's behavior. Leveraging a novel strategy to trim the extracted app's behavior prior to translating them to analyzable formal specifications, IotCom mitigates the state explosion associated with formal analysis. Our experiments with numerous bundles of real-world IoT apps have corroborated IotCom's ability to effectively detect a broad spectrum of interaction threats triggered through cyber and physical channels, many of which were previously unknown, and to significantly outperform the existing techniques in terms of scalability.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Interaction Threats, IoT Safety, Formal Verification

ACM Reference Format:
Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable Analysis of Interaction Threats in IoT Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3395363.3397347

*Equal contribution
†Now at University of Wisconsin-Madison.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397347

1 INTRODUCTION

Internet-of-Things (IoT) ecosystems are becoming increasingly widespread, particularly in the context of the smart home. Industry forecasts suggest the average smart home will end the current year with as many as 50 connected devices [32]. The race to configure, control, and monitor these devices produces a web of different platforms all operating within the same cyber-physical environment. These platforms also allow users to install third-party software *apps*, which can intricately interact with each other, allowing complex and varied automations. Such diversity enhances the user's experience by delivering many options for automating their home, but it comes at a price; it also escalates the attack surface for safety and security threats. The increased complexity produced by the coordination and interaction of these apps subjects the system to the risk of undesired behavior, whether by misconfiguration, developer error, or malice. For example, an app that unlocks the door when a user returns home may be subverted—either accidentally or intentionally—to unlock the door when the user is not actually present. This risk is exacerbated by the unpredictable nature of IoT environments, as it is not known *a priori* which apps and devices will be installed in tandem. The increased impact of these physical risks makes identification of risky interactions even more important.

In this context, the safety implications and security risks of IoTs have been a thriving subject of research for the past few years [3, 15, 17, 19, 20, 24, 26, 33, 38, 39, 47, 57, 58, 64] These research efforts have scrutinized deficiencies from various perspectives. However, existing detection techniques target only certain types of inter-app threats [20, 38, 47] and do not take into account *physical channels*, which can underpin risky interactions among apps. Moreover, the state-of-the-art techniques suffer from acknowledged missing of interaction threats, as they require manual specification of the initial configuration for each app to be analyzed. This, in turn, results in missing potentially unsafe behavior if it appears from different configurations [20, 26, 38, 47]. Additionally, these analyses have been shown to experience scalability problems when applied on large numbers of IoT apps [20, 38, 47].

To address this state of affairs, this paper presents a novel approach and accompanying tool suite, called IotCom, for compositional analysis of such hidden and unsafe interaction threats in a given bundle of cyber and physical components co-located in an IoT environment. IotCom first utilizes a path-sensitive static analysis to automatically generate an inter-procedural control flow graph (ICFG) for each app. It then applies a novel graph abstraction technique to model the behavior relevant to the devices connected to the app as a *behavioral rule graph (BRG)*, which derives rules

from IoT apps via linking the triggers, actions, and logical conditions of each control flow in each app. Unlike prior techniques, our approach respects the conditions along each branch rather than only the triggers and actions. IotCom then automatically generates formal app specifications from the BRG models. Lastly, it uses a lightweight formal analyzer [37] to check bundles of those models for violations of multiple safety and security properties arising from interactions among the apps rules.

IotCom has several advantages over existing work. First, unlike prior work, IotCom supports the detection of violations occurred through physically mediated interactions by explicitly modeling a mapping of each device capability to the pertinent physical channels. Second, while prior approaches require manual specification of the initial configuration, IotCom exhaustively identifies all initial configurations, which in turn enables automatically checking them for potential interaction violations with no need for any manual configuration. Third, our novel BRG abstraction technique optimizes the performance of our analysis and markedly improves our scalability over state-of-the-art techniques by effectively trimming the automatically-extracted ICFGs, eliding all nodes and edges irrelevant to the app's behavior from the analysis.

Using a prototype implementation of IotCom, we evaluated its performance in detecting prominent classes of IoT coordination threats among thousands of publicly-available real-world IoT apps developed using diverse technologies. Our results corroborate IotCom's ability to effectively detect complex coordinations among apps communicating via both cyber and physical means, many of which were previously unreported. We also demonstrate IotCom's significantly improved scalability when compared to existing IoT threat detection techniques.

We further compare the precision of IotCom to the other approaches using a set of benchmark IoT apps, developed by the other research groups [20, 26]. IotCom is up to 68.8% more successful in detecting safety violations. Additionally, compared to the state-of-the art techniques in detecting safety and security violations in IoT environments, IotCom reduces the violation detection time by 92.1% on average and by as much as 99.5%. To summarize, this paper makes the following contributions:

- *Classification of interaction threats between IoT apps.* We identify and rigorously define seven classes of multi-app interaction threats between IoT apps over physical and cyber channels both within and among apps.
- *Formal model of IoT systems.* We develop a formal specification of IoT systems, respecting *cyber and physical channels* and representing the behavior of IoT apps apropos the detection of safety and security vulnerabilities. We construct this specification as a reusable Alloy [1] module to which all extracted app models conform.
- *Automated analysis.* We show how to exploit the power of our formal abstractions by building a modular model extractor that uses static analysis techniques to automatically extract the precise behavior of IoT apps into a trimmed *behavioral rule graph*, respecting the *logical conditions* that impact the behavior of the app rules, which is then captured in a format amenable to formal analysis.
- *Experiments.* We evaluate the performance of IotCom against real-world apps developed for *multiple IoT platforms* (SmartThings and IFTTT), corroborating IotCom's ability in effective compositional analysis of IoT apps interaction vulnerabilities in the order of minutes. We make research artifacts, including the entire Alloy specifications, and the experimental data available to the research and education community [35].

2 BACKGROUND AND MOTIVATION

Smart home IoT platforms are cyber-physical systems comprising both virtual elements, such as software, and physical devices, like sensors or actuators. In popular smart home platforms, such as SmartThings [49], Apple's HomeKit [5], GoogleHome [29], Zapier [63] and MicrosoftFlow [44], physical devices installed in the home are registered with virtual proxies in a cloud-based backend. Each proxy tracks the state of the device via one or more *attributes*, which can assume different *values*. The backend also allows the user to install software *apps*, which automate the activities of these devices by applying custom *rules* that act on the virtual proxies. These rules adopt a *trigger-condition-action* paradigm:

Triggers: Cyber or physical events reported to the smart home system by the devices, such as a motion sensor being activated, trigger the rules.

Conditions: Logical predicates defined on the current state of the devices determine if the rule should execute. For example, a rule might only execute if the system is in "home" mode.

Actions: If the conditions are met, the rule changes the state of one or more devices, which could result in a physical change like activating a light switch.

The safety and security of these systems is a major concern [17, 38, 47], particularly regarding software apps. Users can install multiple, arbitrary apps which can interact not only with physical devices in the smart home, but also with each other. *Multi-app coordination threats among smart home apps arise when two or more app rules interact to produce a surprising, unintended, or even dangerous result in the physical environment of the smart home.* Apps can interact over *cyber* channels such as shared device proxies, global settings, or scheduled tasks. We refer to coordination over cyber channels as *direct* coordination. They also interact over *physical* channels [26] via a shared metric acted upon by an *actuator* and monitored by a *sensor*; this is termed *indirect* coordination.

To make the idea concrete, consider three publicly-available apps, i.e., the SmartThings app *MultiSwitch* and the IFTTT automations *If It's Bright Turn Off the Light* and *Living Room Lamp On*. Figure 1 shows an example configuration of these apps along with their devices, which can enter into an infinite loop. *If It's Bright* watches a light sensor. When the light reaches a user-defined level, it turns off the *MultiSwitch*. *MultiSwitch* forwards that command to both the overhead light and the lamp. *Living Room Lamp On* responds to the overhead light going off by turning on the lamp. That, in turn, activates the light sensor through luminance physical channel, which initiates the same chain of events again.

The above example points to one of the most demanding issues in the smart home IoT ecosystem, i.e., detection of multi-app coordination threats. What is required is a system-wide reasoning—that determines how those individual rules could impact one another

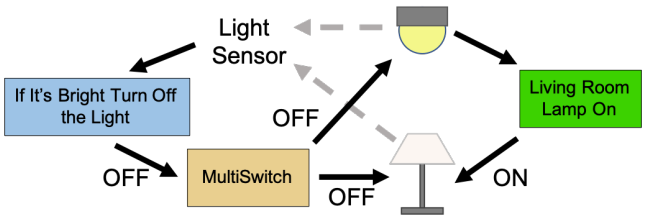


Figure 1: Example infinite actuation loop. The apps turn the living room lights on and off again repeatedly due to a misconfiguration.

when the corresponding apps are installed together—not comfortably attainable through conventional analysis methods such as static analysis or testing, which are more suited for identifying issues in individual parts of the system. In the next sections, we first provide a classification of various multi-app coordination threats, and then present a formal modeling approach to address these issues.

3 MULTI-APP COORDINATION THREATS

In this section, we present seven classes of potential multi-app coordination threats that can arise due to interactions between IoT app rules. As defined earlier, a *rule* comprises a set of *triggers*, *conditions*, and *actions*. More formally, an app rule ρ is a tuple $\rho = \langle T_\rho, C_\rho, A_\rho \rangle$, where T_ρ, C_ρ, and A_ρ are sets of triggers, conditions, and actions for rule ρ, respectively. Each *component* (trigger, condition, or action) can be described as a triple of a device, an attribute, and a set of values. The sets of devices, attributes, and values associated with a component α are denoted as $\mathbb{D}(\alpha)$, $\mathbb{A}(\alpha)$, and $\mathbb{V}(\alpha)$, respectively.

3.1 Direct Coordination

Two rules R_i and R_j directly coordinate if an *action* from R_i influences the devices and attribute associated with a component of R_j. As they act on a shared physical environment, co-located apps can coordinate via both cyber and physical means. In cyber coordination, the devices and attribute of an action of R_i must *match* those referenced by some component of R_j:

Definition 3.1 (match). The relation *match* defines an intersection between the devices and attribute of a component α from an IoT app rule R_i and a component β from another IoT app rule R_j:

$$match(\alpha, \beta) \equiv \big(\mathbb{D}(\alpha) \cap \mathbb{D}(\beta) \neq \emptyset\big) \wedge \big(\mathbb{A}(\alpha) = \mathbb{A}(\beta)\big)$$

Physical coordination is mediated by some physical *channel*. We define the set of physical channels as Channels, with Actuators(γ) and Sensors(γ) denoting the sets of devices that can either act upon or sense changes to a channel $\gamma \in$ Channels, respectively.

Definition 3.2 (same_channel). The relation *same_channel* defines physically-mediated interaction via channel γ between the devices of an action α from an IoT app rule R_i and a trigger β from another IoT app rule R_j:

$$same\_channel(\alpha, \beta) \equiv \exists \gamma \in \textsc{Channels}.((\mathbb{D}(\alpha) \subseteq \textsc{Actuators}(\gamma)) \wedge (\mathbb{D}(\beta) \cap \textsc{Sensors}(\gamma) \neq \emptyset))$$

The most common type of coordination—which is a core feature of IoT automation systems—is *(T1) action-trigger* coordination, where the value of one of R_i's actions matches a value in or actuates a channel sensed by one of R_j's triggers, as depicted in Figure 2. This coordination is often intended but can also lead to an unintended activation of subsequent rules if misused.

Definition 3.3 ((T1) Action-trigger). Action a of rule R_i activates trigger t of rule R_j either directly (by involving overlapping devices, attributes, and values) or mediated by some physical channel actuated by the devices of a and sensed by the devices of t.

$$T1(R_i, R_j) \equiv \big((\forall a \in A_{R_i}, c \in C_{R_j}.(match(a, c) \Rightarrow (\mathbb{V}(a) \subseteq \mathbb{V}(c))) \wedge (\exists a \in A_{R_i}, t \in T_{R_j}.((match(a, t) \wedge (\mathbb{V}(a) \subseteq \mathbb{V}(t))) \vee (same\_channel(a, t))))\big)$$

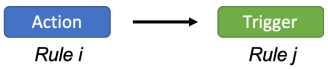


Figure 2: (T1) Action-trigger

Action-condition coordination may be less obvious to the user; in this case, R_i either (*T2*) enables or (*T3*) disables R_j depending on whether or not the values of the action and the related condition match, as shown in Figures 3a and 3b.

Definition 3.4 ((T2) Action-condition (match)). Rule R_i executes action a that changes the system to satisfy condition c of rule R_j.

$$T2(R_i, R_j) \equiv \big(\forall a \in A_{R_i}, c \in C_{R_j}.(match(a, c) \Rightarrow (\mathbb{V}(a) \subseteq \mathbb{V}(c))\big)$$

Definition 3.5 ((T3) Action-condition (no match)). Rule R_i executes action a that changes the system to no longer satisfy condition c of rule R_j.

$$T3(R_i, R_j) \equiv \big(\exists a \in A_{R_i}, c \in C_{R_j}.(match(a, c) \wedge (\mathbb{V}(a) \cap \mathbb{V}(c) = \emptyset)\big)$$

(a) (T2) match **(b) (T3) no match**

Figure 3: Action-condition

3.2 Chain Coordination

The first three classes define ways for two rules to coordinate directly, but apps may also coordinate via a chain of rules—each coordinating with another via action-trigger (T1) coordination—or via a relationship between their triggering event(s). We refer to two rules that can be triggered by the same event as *siblings* if their conditions are not mutually exclusive.

Definition 3.6 (sibling). Relation *sibling* holds between two rules R_i and R_j if and only if there is a trigger t_1 in R_i that overlaps via devices, attributes, and values with a trigger t_2 in R_j; none of the conditions in R_i violate any condition of R_j; and vice versa.

$$sibling(R_i, R_j) \equiv \big((\forall c_1 \in C_{R_i}, c_2 \in C_{R_j}. (match(c_1, c_2) \Rightarrow (\mathbb{V}(c_1) \cap \mathbb{V}(c_2) \neq \emptyset))) \wedge (\exists t_1 \in T_{R_i}, t_2 \in T_{R_2}.(match(t_1, t_2) \wedge (\mathbb{V}(t_1) \cap \mathbb{V}(t_2) \neq \emptyset)))\big)$$

Chain coordination may lead to the scenario where the action of a chain of rules (or a single rule itself) triggers one of the rules previously involved in the same chain. We define this class of multi-app coordination threats as *(T4) self coordination* (shown in Figure 4):

Definition 3.7 ((T4) Self coordination). Rule R_i—either directly or through a transitive chain of intermediate rules connected by the $T1$ relation, denoted $T1^+$—triggers itself.

$$T4(R_i) \equiv T1^+(R_i, R_i)$$

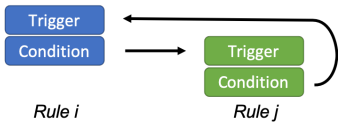


Figure 4: (T4) Self coordination

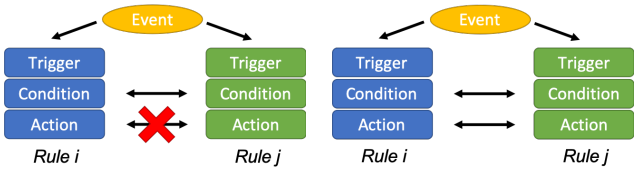
Action-action coordination occurs when two distinct rules act upon the same attribute of the same device. If the rules are triggered by unrelated events, there is no coordination between the two rules. If the triggering events are the same, then the rules may coordinate to produce an undesired result such as a race condition or additional wear on a given device (cf. Figures 5a-5b):

Definition 3.8 ((T5) Action-action (conflict)). Two rules R_i and R_j that are each triggered by the same event—either directly or through a chain of coordinating rules—each has an action with the same device and attribute but different values. ($T1^*$ denotes the reflexive transitive closure of the $T1$ relation).

$$T5(R_i, R_j) \equiv (\exists a_1 \in A_{R_i}; a_2 \in A_{R_j}; R_m, R_n \in R.(match(a_1, a_2) \wedge (\mathbb{V}(a_1) \neq \mathbb{V}(a_2)) \wedge T1^*(R_m, R_i) \wedge T1^*(R_n, R_j) \wedge ((R_m = R_n) \vee sibling(R_m, R_n))))$$

Definition 3.9 ((T6) Action-action (repeat)). Two rules R_i and R_j that are each triggered by the same event—either directly or through a chain of coordinating rules—each has an action with the same device, attribute, and value.

$$T6(R_i, R_j) \equiv (\exists a_1 \in A_{R_i}; a_2 \in A_{R_j}; R_m, R_n \in R.(match(a_1, a_2) \wedge (\mathbb{V}(a_1) = \mathbb{V}(a_2)) \wedge T1^*(R_m, R_i) \wedge T1^*(R_n, R_j) \wedge ((R_m = R_n) \vee sibling(R_m, R_n))))$$



(a) (T5) conflict (b) (T6) repeat

Figure 5: Action-action

The prior chain coordinations result from a single initiating event; it is also possible for rules to be configured such that a given outcome is guaranteed even in the face of mutually-exclusive events. We define a seventh class of coordination—*(T7) exclusive event coordination*—to detect this case, shown in Figure 6.

Definition 3.10 ((T7) Exclusive event coordination). Two rules R_i and R_j that would be triggered by *mutually exclusive* events—which share a device and attribute but different values—have actions that share the same device, attribute, and value.

$$T7(R_i, R_j) \equiv (\exists a_1 \in A_{R_i}; a_2 \in A_{R_j}; R_m, R_n \in R; t_1 \in T_{R_m}; t_2 \in R_n. (match(a_1, a_2) \wedge (\mathbb{V}(a_1) = \mathbb{V}(a_2)) \wedge T1^*(R_m, R_i) \wedge T1^*(R_n, R_j) \wedge match(t_1, t_2) \wedge (\mathbb{V}(t_1) \cap \mathbb{V}(t_2) \neq \emptyset))$$

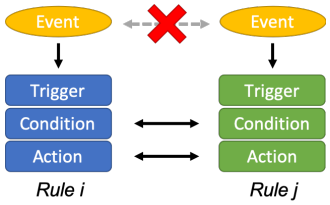


Figure 6: (T7) Exclusive event

In the rest of the paper, we show how, through a practical combination of static analysis and a lightweight formal method, IotCom can automatically discover such unsafe and intricate interaction threats in a compositional and scalable fashion.

4 APPROACH OVERVIEW

This section introduces IotCom, a technique that automatically determines whether the interactions within an IoT environment could compromise the safety and security thereof. Figure 7 illustrates the architecture of IotCom and its two major components, described in detail in the following two sections:

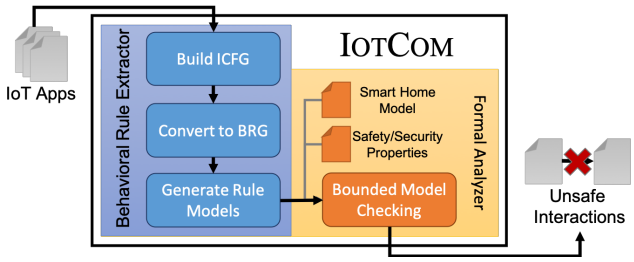


Figure 7: IotCom System Overview.

(1) Behavioral Rule Extractor (Section 5): The *Behavioral Rule Extractor* component automatically infers models of the apps behavior using a novel graph abstraction technique. The component first performs static analysis on each app to generate an interprocedural control flow graph (ICFG). It then creates a *behavioral rule graph* containing only the flows pertinent to the events and commands forwarded to/from physical IoT devices, along with any conditions required for those actions. Each flow is then automatically transformed into a formal model of the app.

(2) Formal Analyzer (Section 6): The *Formal Analyzer* component is then intended to use lightweight formal analysis techniques to verify specific properties (i.e., IoT coordination threats) in the extracted specifications. IotCom uses three formal specifications:

(1) a *base model of smart home IoT systems* that defines foundational rules for cyber and physical channels, IoT apps, how they behave, and how they interact with each other, (2) assertions for *safety and security properties*, and (3) the *IoT app behavioral rule models* automatically generated by the previous component for each app. Those specifications are then checked together for detecting violations of the properties.

5 BEHAVIORAL RULE EXTRACTOR

The Behavioral Rule Extractor executes three main steps to automatically infer the behavior of individual IoT apps: (1) build an *inter-procedural control flow graph (ICFG)*; (2) convert the ICFG to a *behavioral rule graph (BRG)*; and (3) generate formal models for the behavioral rules.

5.1 Building ICFG

The Behavioral Rule Extractor first generates an inter-procedural control flow graph for each app. It analyzes the abstract syntax tree of the given app to build a *call graph* of local and API-provided methods as well as a *control flow graph* for each local method. Each graph is generated using a path-sensitive analysis [41] to preserve the logical conditions along each control flow. To capture the precise behavior of IoT apps, it is essential to extract the predicates that will influence the actions performed by IoT apps. Path-sensitivity supports this goal. Many of the triggered methods in the IoT apps are calling other methods that will perform certain actions. As such, an inter-procedural analysis is required to consider the calling context when analyzing the target of a triggered method. This, in turn, improves the precision of our approach compared to the state-of-the-art techniques, which do not consider the conditions when identifying interactions between apps [26]. The Behavioral Rule Extractor then combines each control flow graph with the call graph to construct an ICFG starting at each *entry method* in the graph. The details of generating the ICFG depend on how apps are defined for each platform.

IFTTT applets are reactive rules that interact with REST services exposed by service providers [34]. Each applet consists of a single trigger-action pair. The Behavioral Rule Extractor treats each applet as a standalone IoT app defining exactly one rule. It performs string analysis [22] to extract an ICFG comprising one entry node for the trigger and one "method call" invoking a device API for the action. For instance, IFTTT applet "If I arrive at my house then open my garage door" [28] would result in an ICFG with an entry node for "arrive at my house" and a method call node for "open my garage door". Automatically identifying the corresponding keywords specified in triggers and actions, in turn, allows IotCom to detect the associated capability, attribute, and value in the next steps.

SmartThings apps are written as Groovy programs, allowing for multiple rules and more extensive logic. To generate an ICFG for a SmartThings app, the Behavioral Rule Extractor first extracts principal information regarding: (1) the devices and attributes used in the app, (2) the user's configuration of the app, (3) any global variables defined in the SmartThings documentation [49] or set using the `state` object, and (4) the entry methods of the app's triggers. The SmartThings platform defines entry methods using calls to specific API methods: `subscribe`, `schedule`, `runIn`, and `runOnce`. Next, the Behavioral Rule Extractor creates a control flow graph for each of those entry methods. These graphs are combined to generate an inter-procedural control flow graph for the IoT app. Note that existing state-of-the-art analysis techniques lack support for direct program analysis of Groovy code. By performing the analysis directly on the Groovy code, IotCom avoids the pitfalls (and cost) of translating the code into some intermediate representation. As a concrete example, Listing 1 shows the Groovy code defining the *MultiSwitch* app described in Section 2. The calls to `subscribe` on Lines 11-12 define two entry points—`switchOn` and `switchOff`. Each of those methods would comprise two nodes in the ICFG corresponding to the logging line and API call contained in each.

Listing 1 Groovy code for MultiSwitch app

```
1  preferences {
2     section("When this switch is toggled...") {
3        input "master", "capability.switch", title: "Where?" }
4     section("Turn on/off these switches...") {
5        input "switches", "capability.switch", multiple: true } }
6  def installed() {
7     subscribe(master, "switch.On",  switchOn)
8     subscribe(master, "switch.Off", switchOff) }
9  def updated() {
10    unsubscribe()
11    subscribe(master, "switch.On",  switchOn)
12    subscribe(master, "switch.Off", switchOff) }
13 def switchOn(evt)  {
14    log.debug "Switches on"
15    switches?.on()  }
16 def switchOff(evt) {
17    log.debug "Switches off"
18    switches?.off() }
```

5.2 Generating Behavioral Rule Graph

The Behavioral Rule Extractor next tailors the ICFG into a succinct, annotated graph representing the relevant behavior of the IoT app—a *behavioral rule graph (BRG)*. By eliding all edges and nodes from the ICFG that do not impact the app's behavior with respect to physical devices, the BRG makes it easier to infer the behavior defined in the app, optimizing the performance of our analysis. To construct the BRG from the ICFG, the nodes in the ICFG are traversed starting from each entry method, generating nodes in the BRG as follows:

- **Trigger:** Entry method nodes from the ICFG are propagated to the BRG as trigger nodes.
- **Condition:** Control statements such as `if` blocks generate condition nodes in the BRG.
- **Action:** Any node that invokes a device API method creates an action node in the BRG.
- **Method Call:** Method calls to other local methods produce method call nodes in the BRG, as the called method may include relevant app behavior.

Continuing the example (Listing 1) from Section 5.1, the Behavioral Rule Extractor converts the ICFG for *MultiSwitch* into a BRG, starting with the entry point function `switchOn`. The next node in the ICFG is a call to a logging API, which has no bearing on the app's behavior and is not included in the BRG. The node following the logging is an API call to `on`, which translates directly to an action node in the BRG. The resulting BRG captures all actions that affect the devices in the environment. If the example contained any

conditional statements, the BRG would also maintain the control flow reflecting the conditions for each action.

After creating the BRG, the statements corresponding to each node are converted to ⟨*device*, *attribute*, *value*⟩ tuples. If a value in any of the nodes does not correspond directly to a member of one of those sets, we perform backward inter-procedural data flow analysis [45] to resolve the dependency.

5.3 Generating Rule Models

The final component of the Behavioral Rule Extractor generates formal models of each app's rules based on the BRG. As described in Section 2, the behavior of an IoT app consists of a set of rules R, where each rule is a tuple of triggers, conditions, and actions.

In order to tie the behavior of these rules back to the physical devices in the smart home, the elements of T, C, and A are each formalized as sets of tuples of ⟨*device*, *attribute*, *value*⟩. Each type of device is assumed to have its own set of device-specific attributes, and each attribute constrains its own allowed values according to the device manufacturer's specifications. For example, a smart lock *device* may have a "locked" *attribute* to indicate the state of the lock, which accepts *values* of "locked" or "unlocked". An action to unlock a specific lock (*TheLock*) would contain a tuple composed of those elements, e.g., ⟨*TheLock*, *locked*, *unlocked*⟩.

To generate the models from the BRG, IotCom starts from each trigger node (which is used as the Trigger for the rule) and traverses the graph to find the action nodes; every rule must have at least one Action. From each action node, it performs a reverse depth-first search back to the trigger, collecting the tuples for each condition node encountered along the path as the Conditions of the rule.

6 FORMAL ANALYZER

This section describes the *Formal Analyzer* component of IotCom, which takes as input the behavioral rule models generated by the *Behavioral Rule Extractor*. These formal models are verified against various safety and security properties using a bounded model checker to exhaustively explore every interaction within a defined scope. This allows IotCom to automatically analyze each bundle of apps without manual specification of the initial system configuration, which is required for comparable state-of-the-art techniques [20, 47]. We use Alloy [37] to demonstrate our approach for several reasons. First, it provides a concise, simple specification language suitable for declarative specification of both IoT apps and safety and security properties to be checked. In particular, Alloy includes support for modeling transitive closure, which is essential to analyze complex, chained interactions. Second, it provides a fully-automated analyzer, shown to be effective in exhaustively analyzing specifications in various domains [7–13, 43].

The bounded model checking relies on three sets of formal specifications, as shown in Figure 7: (1) a base *smart home model* describing the general entities composing a smart home environment; (2) the app-specific *behavioral rule models* generated by the *Behavioral Rule Extractor*; and (3) formal assertions for our *safety and security properties*. Complete Alloy models are available online at our project site [35].

Listing 2 Excerpt of base smart home Alloy model.

```
abstract sig Device    { attributes : set Attribute }
abstract sig Attribute { values : set Value }
abstract sig Value     { }
abstract sig IoTApp { rules : set Rule }
abstract sig Rule {
  triggers   : set Trigger,
  conditions : set Condition,
  actions    : some Action }
// Trigger, Condition, and Action contain
// similar tuples
abstract sig Trigger {
  devices    : some Device,
  attribute  : one  Attribute,
  values     : set  Value }
abstract sig Condition { ... }
abstract sig Action    { ... }
```

Listing 3 Excerpt of Environment model.

```
abstract sig Channel {
  sensors   : set Capability,
  actuators : set Capability }
one sig ch_temperature extends Channel {} {
  sensors   = cap_temperatureMeasurement
  actuators = cap_switch + cap_thermostat + cap_ovenMode }
one sig ch_luminance extends Channel {} {
  sensors   = cap_illuminanceMeasurement
  actuators = cap_switch + cap_switchLevel }
one sig ch_motion extends Channel {} {
  sensors   = cap_motionSensor + cap_contactSensor
  actuators = cap_switch }
```

6.1 Smart Home Base Model

The overall smart home system is modeled as a set of Devices and a set of IoTApps, as shown in Listing 2. Each IoTApp contains its own set of Rules. Each Device has some associated state Attributes, each of which can assume one of a disjoint set of Values. Recall from Section 4, each rule contains its own set of Triggers, Conditions, and Actions. Each individual trigger, condition, and action is modeled as a tuple of one or more Devices, the relevant Attribute for that type of device, and one or more Values that are of interest to the trigger, condition, or action. Defined in Alloy, each of the listed entities is an abstract signature which is extended to a concrete model signature for each specific type of device, attribute, value, IoT app, behavioral rule, etc.

Environment Modeling. Apps can communicate both *virtually* within the cloud backend and *physically* via the devices they control. Virtual interactions fall into two main categories: (1) direct mappings, where one app triggers another by acting directly on a virtual device/variable watched by the triggered app; or (2) scheduling, where one rule calls—e.g., using the runIn API from SmartThings—to invoke a second rule after a delay. Physically mediated interactions occur indirectly via some physical *channel*, such as temperature. Our model—in contrast to others [20, 47]—directly supports detection of violations mediated via physical channels (cf. Listing 3). As part of our model of the overall SmartThings ecosystem, we include a mapping of each device to one or more physical Channels as either a sensor or an actuator.

6.2 Extracted IoT App Behavioral Rule Models

The second set of specifications required by the *Formal Analyzer* is the models automatically extracted from each individual IoT app. These specifications extend the base specifications described in Section 6.1 with specific relations for each individual IoT app.

Listing 4 Excerpts from the generated specification for MultiSwitch (Listing 1)

```
1  one sig MultiSwitch extends IoTApp {
2    master   : one  Switch,
3    switches : some Switch }
4      { rules = r0 + r1 }
5
6  one sig r0 extends Rule {}{
7    triggers  = r0_trig0
8    no conditions
9    actions   = r0_act0 }
10 one sig r0_trig0 extends Trigger {} {
11   devices   = MultiSwitch.master
12   attribute = Switch_State
13   value     = ON }
14 one sig r0_act0 Action {} {
15   devices   = MultiSwitch.switches
16   attribute = Switch_State
17   value     = ON }
18
19 one sig r1 extends Rule {}{ ... }
```

Listing 4 partially shows the Alloy specification generated for the MultiSwitch app from Section 2. First, the new signature MultiSwitch extends the base IoTApp by adding fields for some Switch devices–a single master and multiple others–as well as constraining the inherited rules field to contain the two rules, r0 and r1, defined on Lines 6 and 19 as extensions of Rule. As described in Section 5, the *Behavioral Rule Extractor* generates the tuples for the triggers, conditions, and actions of each app's rules from the behavioral rule graph. In this case, the entry point node corresponding to the `switchOn` method is translated into the r0_trg0 signature (Line 10) while the action node of the BRG generates r0_act0 (Line 14). The bundle of these specifications define all apps co-installed in the system.

6.3 Safety/Security Properties

In Section 3, we present seven classes of multi-app coordination threats that violate safety. In this section, we describe Alloy assertions drawn from the logical definitions of those threat classes. These assertions express the threats as *safety properties* that are expected to hold in the specifications extracted from each individual IoT app. We also draw upon prior work [20, 26, 47] to define specific unsafe or undesirable behaviors that may result from chain triggering. Table 1 summarizes A set of safety and security properties caused due to action-trigger coordination (T1), derived from the literature [20, 26, 47] and formally verified by IotCom. In total, we consider 36 safety and security properties, 7 generic coordination threats introduced Section 3) and 29 properties listed in Table 1[1].

As a concrete example of the seven coordination threats, the assertion *T4* in Listing 5 corresponds to threat *T4*, presented in Section 3. In this snippet, we define the predicate are_connected (Lines 7-11) which encodes the relation specified in Def. 3.3 for threat *(T1)*. It relies on the two other predicates, match (Lines 2-3) and same_channel (Lines 4-6), which correspond to the logical relations of the same names defined in Defs. 3.1 and 3.2 from Section 3. The fact on Line 13 converts are_connected predicate to a field connected on the Rule signature. Line 17 then defines the assertion, which ensures that no rule is found in the transitive closure of it's own connected rules, matching the definition of the threat class from Def. 3.7.

[1]The complete specifications of these properties are available online at our project site [35].

Table 1: A sample of safety and security properties caused due to *action-trigger coordination*, derived from the literature [20, 26, 47] and formally verified by IotCom.

Property	Description
P.1	DON'T turn on the AC WHEN mode is away
P.2	DON'T turn on the bedroom light WHEN door is closed
P.3	DON'T turn on dim light WHEN there is no motion
P.4	DON'T turn on living room light WHEN no one is home
P.5	DON'T turn on dim light WHEN no one is home
P.6	DON'T turn on light/heater WHEN light level changes
P.7	DON'T turn off heater WHEN temperature is low
P.8	DON'T unlock door WHEN mode is away
P.9	DON'T turn off living room light WHEN someone is home
P.10	DON'T turn off AC WHEN temperature is high
P.11	DON'T close valve WHEN smoke is detected
P.12	DON'T turn off living room light WHEN mode is away
P.13	DON'T turn off living room light WHEN mode is vacation
P.14	DO set mode to away WHEN no one is home
P.15	DO set mode to home WHEN someone is home
P.16	DON'T turn on heater WHEN mode is away
P.17	DON'T open door/window WHEN smoke is detected or mode is away
P.18	DON'T turn off security system WHEN no one is home
P.19	DON'T turn off the alarm WHEN smoke is detected
P.20	DON'T unlock the door WHEN light level changes
P.21	DON'T lock the door WHEN smoke is detected
P.22	DON'T open the door WHEN smoke is detected and heater is on
P.23	DON'T unlock the door WHEN smoke is detected and heater is on
P.24	DON'T open the door WHEN motion is detected and fan is on
P.25	DON'T unlock the door WHEN motion is detected and fan is on
P.26	DON'T open the door/window WHEN temperature changes
P.27	DON'T set mode WHEN temperature changes
P.28	DON'T set mode WHEN smoke is detected
P.29	DON'T set mode WHEN motion is detected and alarm is sounding

As a concrete example of a safety properties caused due to action-trigger coordination (T1), Listing 6 illustrates the Alloy assertion for one of the fine-grained safety properties analyzed by IotCom (P.8 in Table 1). This assertion states that no rule (r) should have an action (a, Line 2) that results in a contact sensor (i.e., the door) being opened (Lines 4-5) while the home mode is Away (Lines 10-11).

Listing 5 Example coordination threats (T1) and (T4) defined as assertions in Alloy. The are_connected predicate and connect attribute encode the logical definition of T1.

```
1  // example relations defining coordination (cf. Section 3)
2  pred match[a : Action, t : Trigger] {
3     (some t.devices & a.devices) and (a.attribute = t.attribute)}
4  pred same_channel[a : Action, t : Trigger] {
5     (some c : Channel | (a.devices in c.actuators) and
6        (some t.devices & y.sensors))}
7  pred are_connected[r,r' : Rule] {
8     (some a : r.actions, t : r'.triggers {
9        (match[a, t] and (a.value in t.value)) or same_channel[a, t]})
10    all a : r.actions, c : r'.conditions {
11       (match[a,c]) => (a.value in c.value) } }
12 // defines the 'connected' field so we can use transitive closure
13 fact { all r,r' : Rule | (r' in r.connected) <=> are_connected[r,r'] }
14 // action-trigger coordination
15 assert t1 { no Rule.connected }
16 // self coordination
17 assert t4 { no r : Rule | r in r.^connected }
```

The property check is formulated as a problem of finding a valid trace that satisfies the specifications but violates the assertion. The returned solution encodes the sequence of rule activations leading to the violation. Given our running example (cf. Figure 1), the analyzer automatically detects a violation scenario, a visualization of which is shown in Figure 8, where the four rules form a cycle.

Listing 6 Example Alloy assertion for the property DON'T unlock door WHEN location mode is Away.

```
assert P8 {
  no r : IoTApp.rules, a : r.actions {
    // DON'T open the door...
    a.attribute = CONTACT_SENSOR_CONTACT_ATTR
    a.values    = CONTACT_SENSOR_OPEN
    // ... WHEN ...
    ((some r' : r.*are_connected, t : r'.triggers {
      (some r' : r.*are_connected, a' : r'.actions {
      // ...mode is away
      a'.attribute = MODE_ATTR
      a'.values    = MODE_AWAY })) }}
```

IotCom's ability to detect violations in complex chains of interaction across both cyber and physical channels sets it apart from other research in the area.

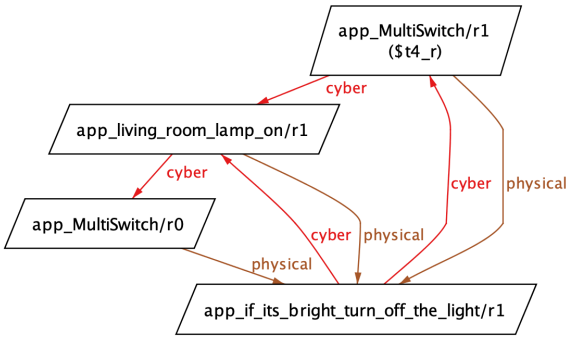


Figure 8: An automatically detected violation scenario for our running example (cf. Fig. 1), where the four rules form a cycle.

7 EVALUATION

This section presents our experimental evaluation of IotCom, addressing the following research questions:

- **RQ1:** What is the overall accuracy of IotCom in identifying safety and security violations compared to other state-of-the-art techniques?
- **RQ2:** How well does IotCom perform in practice? Can it find safety and security violations in real-world apps?
- **RQ3:** What is the performance of IotCom's analysis realized atop static analysis and verification technologies?

Experimental subjects. Our experiments are all run on a multi-platform dataset of 3732 smart home apps drawn from two sources: *(1) SmartThings apps:* We gathered 404 SmartThings apps from the SmartThings public repository [51]. These apps are written in Groovy using the SmartThings Classic API platform. *(2) IFTTT applets:* We used the IFTTT dataset provided by Bastys et al. [14]. This dataset is in JSON format, with each object defining an IFTTT applet.

Safety and Security Properties. We use a set of 36 safety and security properties for all of our experiments, each encoded as an Alloy assertion as described in Section 6.3.

We performed the experiments on a MacBook Pro with a 2.2GHz 2-core Intel i7 processor and 16GB RAM, with the exception of the real-world app analysis in Section 7.2. Those experiments were performed as distributed jobs on a local cluster of 2 CPU/16 core Intel Xeon processors, with each job assigned up to 32GB RAM. We used Alloy 4.2 for model checking for all experiments.

Table 2: Safety violation detection performance comparison between SOTERIA, IoTSAN and IotCom. True Positive (TP), False Positive (FP), and False Negative (FN) are denoted by symbols ☑, ☒, ☐, respectively. (X#) represents the number # of detected instances for the corresponding symbol X.

Test Cases	SOTERIA*	IoTSAN	IotCom
Individual Apps			
ID1BrightenMyPath	☑	☑	☑
ID2SecuritySystem	☑	☐†	☑
ID3TurnItOnOffandOnEvery30Secs	☑	☐	☑
ID4PowerAllowance	☑☐	(☐2)	(☑2)
ID5.1FakeAlarm	☐	☐	☐
ID6TurnOnSwitchNotHome	☑	☑	☑
ID7ConflictTimeandPresenceSensor	☑	☐‡	☑
ID8LocationSubscribeFailure	☑	☑	☑
ID9DisableVacationMode	☒	☐	☑
Bundles of Apps			
Application Bundle 1	☑	☑	☑
Application Bundle 2	☑	☐†	☑
Application Bundle 3	☑	☐†	☑
Application Bundle 4#	☐	☐‡	☑
Application Bundle 5#	☐	☐	☑
Application Bundle 6#	☐	☐	☑
Precision	90%	100%	**100%**
Recall	66.7%	25%	**93.8%**
F-measure	76.6%	40%	**96.8%**

* results obtained from [21]
† IoTSAN did not generate the Promela model
‡ SPIN crashing
# Benchmarks involving physical channels related violations.

7.1 Results for RQ1 (Accuracy)

To evaluate the effectiveness and accuracy of IotCom and compare it against other state-of-the-art techniques, we used the IoTMAL [36] suite of benchmarks. This dataset contains custom SmartThings Classic apps, for which all violations, either singly or in groups, are known in advance—establishing a ground truth. As IotCom identifies safety/security violations arising from interactions of conflicting rules. Such rules can be defined within the scope of one app or multiple apps. Therefore, to perform a fair comparison with the state-of-the-art, we used benchmarks which incorporate violations in both individual apps and bundles of apps.

We faced two challenges while evaluating the accuracy of IotCom against the state-of-the-art: (1) Most analysis techniques—including HOMEGUARD [24], SOTERIA [20], and iRULER [57]—are not available; IoTSAN [47] was the lone exception. We also were not able to run IoTMON because only one component thereof is publicly accessible; its Groovy parser [2] is available, but the channels discovery and NLP components are not. SOTERIA [20] was evaluated using the IoTMAL dataset, but the tool is not publicly available. Therefore, we rely on the results provided in the technical report [21]. (2) The

[2] https://github.com/nsslabcuus/IoTMon

violations in the IoTMAL dataset do not involve physical channels. For evaluating this capability of the compared techniques, we developed three bundles, B4–B6, available online from the project website [35].

Table 2 summarizes the results of our experiments for evaluating the accuracy of IotCom in detecting safety violations compared to the other state-of-the-art techniques. IotCom succeeds in identifying all 9 known violations out of 10 in the individual apps, and all violations in 6 bundles of apps. Furthermore, IotCom identifies two violations in the test case *ID4PowerAllowance*–namely, *(T6) repeated actions* and *(T5) conflicting actions*. Different from SOTERIA and IoTSAN, IotCom captures schedule APIs; thus, it can identify the conflicting actions violation that was not detected by the other techniques. IotCom misses only a single violation, in test case *ID5.1FakeAlarm*. This app generates a fake alarm using a smart device API not often used in SmartThings apps. Neither SOTERIA nor IoTSAN detected this violation.

IotCom also successfully identifies potential safety and security violations arising from interactions between apps. Test bundles $B1 - B3$ exhibit such violations using only virtual channels of interaction. Bundles $B4 - B6$ define violations due to *physical* interactions between apps. For example, B4 contains an interaction violation over the temperature channel that can result in the door being unlocked while the user is not present, violating one of the specific properties belonging to class *(T1) chain triggering*, while B5 and B6 contain unsafe behavior and infinite actuation loop, respectively. SOTERIA and IoTSAN cannot detect such violations that involve interactions over physical channel.

7.2 Results for RQ2 (Real-World Apps)

We further evaluated the capability of IotCom to identify violations in real-world IoT apps. We randomly partitioned the subject systems of 3732 real-world SmartThings and IFTTT apps into 622 non-overlapping bundles, each comprising 6 apps, in keeping with the sizes of the bundles used in prior work [33, 47]. This partitioning simulates a realistic usage of IoT apps, where no restrictions prevent a user from installing any combination of IoT apps. The resulting bundles enabled us to perform several independent experiments. We evaluated each bundle against (1) the seven classes of interaction threats introduced in Section 3 and (2) a set of specific *action-trigger* coordinations adopted from the literature [20, 26, 47] (listed in Table 1), which we used to assess the capability of our approach to accommodate and detect scenario-based violations. More details on these properties, along with the specifications thereof are available from the project's website [35]. Overall, IotCom detected 2883 violations across the analyzed bundles of real-world IoT apps, with analysis failing on one of the 622 bundles.

Figure 9 illustrates how the detected violations were distributed among the seven classes of multi-app coordination, presented in Section 3. Threats *(T1) Action-trigger*, *(T4) Self coordination*, and *(T6) Action-action (repeat)* were the most prevalent. These threats also appear in the motivating example from Section 2. The fact that action-trigger (T1) and action-action (T5, T6) coordination classes were the most frequently detected violations indicates that race conditions among actions would be the most likely consequence of multi-app coordination. Out of the 29 specific safety properties from

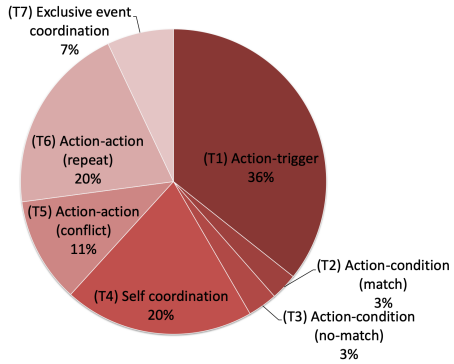


Figure 9: Distribution of the detected violations across seven classes of multi-app coordination.

class *(T1) Action-trigger*, IotCom detects violations of 15 properties, where 72.3% (450 out of 621) of the bundles violate at least one property. In the following, we describe some of our findings.

7.2.1 Violation of (T1) Action-trigger. Figure 10 depicts a chain of virtual interactions that could lead to a door being left unlocked if misconfigured. The SmartThings app *LockItWhenILeave* locks the door when the user leaves the house, as detected by a presence sensor. The lock action triggers the IFTTT applet *Unlock Door*, which unlocks the door again. This violates one of our specific, scenario-based, action-trigger coordination properties.

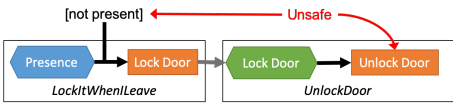


Figure 10: Example violation of *T1*: Cyber coordination between apps may leave the door unlocked when no one is home. The first rule is guarded by a condition that the home owner not be present.

This example also demonstrates IotCom's unique ability to consider logical conditions when evaluating interactions. The code of *LockItWhenILeave* does not specify a particular value for the presence sensor in the trigger for its rule; the entry method is invoked by any change to the presence sensor. Instead, the rule uses a condition to ensure it is only invoked when the user is *not* present. Other techniques, particularly those that require manual specification of the initial system configuration for analysis, may miss this violation by only considering the interaction when the user is present. IotCom does not have such a limitation, and correctly identifies the violation.

7.2.2 Violation of (T4) Self Coordination via Physical Channel. The chain of interactions shown in Figure 11 results in a loop that could continually turn a switch on and off, similar to our example from Section 2. This violation represents the self coordination threat (cf. *T4*). The loop involves three SmartThings apps: *RiseAndShine*, *TurnItOnXMinutesIfLightIsOff*, and *LightsOnWhenIArriveInTheDark*. *RiseAndShine* contains a rule activating some switch when motion

is detected. *LightsOnWhenIArriveInTheDark* controls a group of switches based on the light levels reported by light sensors. *TurnItOnXMinutesIfLightIsOff* switches a switch on for a user-specified period, then turns it back off.

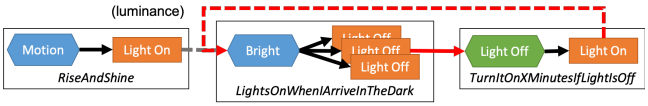


Figure 11: Example violation of *T4*: Lights continually turn off and on. Dashed line represents coordination via the *luminance* physical channel.

When *RiseAndShine* activates its switch, it could trigger *LightsOnWhenIArriveInTheDark* via the *luminance* physical channel, switching all connected lights off. This event triggers *TurnItOnXMinutesIfLightIsOff*, which may re-enable one of the lights. This changes the luminance level, entering into an endless loop between *LightsOnWhenIArriveInTheDark* and *TurnItOnXMinutesIfLightIsOff*. IoTCom is uniquely capable of detecting this violation due to our support of physical channels, scheduling APIs, and arbitrarily long chains of interactions among apps.

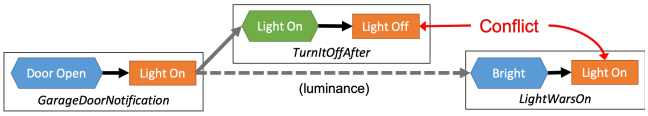


Figure 12: Example violation of *T5*: Both "on" and "off" commands sent to the same light due to the same event. Dashed line represents coordination via the *luminance* physical channel.

7.2.3 Violation of (T5) Action-action (conflict) via Physical Channel. The three apps shown in Figure 12 lead to potentially unpredictable behavior due to competing commands to the same device, violating *T5*. They also interact in part over a physical channel that could not be detected by approaches that only consider virtual interaction between apps. The IFTTT applet *GarageDoorNotification* activates a switch when the garage door is opened. This triggers the action of SmartThings app *TurnItOffAfter*, which will turn off the light after a predefined period. At the same time, *GarageDoorNotification* may also have triggered the IFTTT applet *LightWarsOn* via a light sensor, interacting over the physical *luminance* channel. *LightWarsOn* would attempt to turn the light back on, producing an unpredictable result—a race condition—depending on which rule was executed first.

We also manually evaluated the accuracy of IoTCom on a sample of the real-world bundles. We first randomly selected 30 of the 622 bundles of six applications (approx. 5%), and acquired the source code for the apps in those bundles. We then manually examined the Groovy source or IFTTT definition of each app in each bundle to find violations of the seven classes of multi-app coordination presented in Section 3. The precision, recall, accuracy, and F-measure derived from that random bundles are summarized in Table 3.

Table 3: Results obtained through a manual analysis of a sample of approx. 5% (30) of the 622 real-world bundles.

Precision	Recall	Accuracy	F-measure
47.86%	94.92%	65.71%	62.64%

Note that the low precision (47.86%) is in part due to a quirk of IFTTT. IFTTT applets specify a "channel" to signify the device for the trigger and the action of each rule, identified by an ID value assigned by the platform and passed to the REST interface of the service invoked by the applet. The service controlling Android phones re-uses the same channel ID for multiple tasks; for example, detecting a connection to a particular WiFi network and changing the volume on the phone both correspond to the same channel ID. Our reference implementation of IoTCom interprets both of those to be the same device, so a rule that, for example, muted the phone when connecting to a particular network would be flagged as a violation of both T1 and T4, even though the actual action taken by the rule would not result in any coordination.

7.3 Results for RQ3 (Performance and Timing)

The last evaluation criteria are the performance benchmarks of model extraction and formal analysis of IoTCom on real-world apps drawn from the SmartThings and IFTTT repositories.

Figure 13 presents the time taken by IoTCom to extract rule models from the Groovy SmartThings apps and IFTTT applets. The scatter plot shows both the analysis time and the app size. According to the results, our approach statically analyzes and infer specifications for 98% of apps in less than one second.

We also measured the verification time required for detecting safety/security violations and compared the analysis time of IoTCom against that required by IoTSAN [47]. We checked all 36 safety and security properties against the app bundles. IoTSAN requires the initial configuration for each app in the bundle as part of the model to be analyzed. IoTCom, however, exhaustively examines all configurations that fall within the scope of the app model. To perform a fair comparison between the two approaches, we generated initial configurations for 11 bundles of apps and converted them into

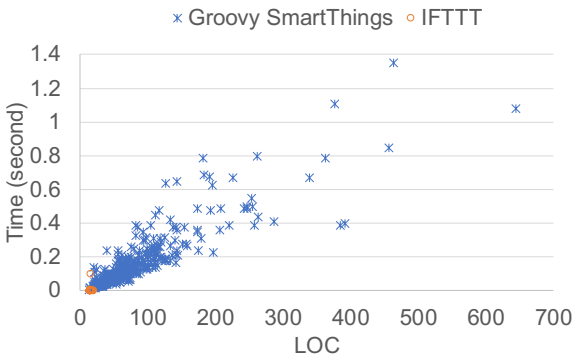


Figure 13: Scatter plot representing analysis time for behavioral rule extraction of IoT apps using IoTCom.

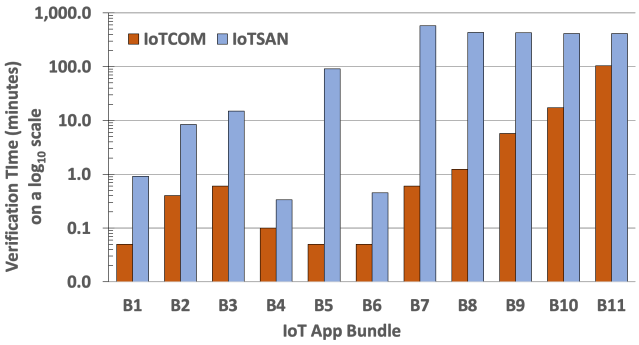


Figure 14: Comparing verification time by IotCom and IoTSAN to perform the same safety violation detection across 11 bundles of real-world apps (in minutes, displayed on a $\log_{10}$ scale). IotCom remarkably outperforms IoTSAN (by 92.1% on average), without sacrificing the detection capability.

a format supported by IoTSAN. We then ran the two techniques considering all valid initial configurations to avoid missing any violation.

Figure 14 depicts the total time taken by each approach to analyze *all* relevant configurations (rather than a single, user-selected configuration). Note that the analysis time is portrayed in a logarithmic scale. The experimental results show that the average analysis time taken by IotCom and IoTSAN per bundle is 11.9 minutes (ranging from 0.05 to 104.78 minutes) and 216.9 minutes (ranging from 0.33 to 580.91 minutes), respectively. Overall, IotCom remarkably outperforms IoTSAN in terms of the time required to analyze the same bundles of apps by 92.1% on average and by as much as 99.5%. The fact that IotCom is able to effectively perform safety/security violation detection of real-world apps in just a few minutes (on an ordinary laptop), confirming that the presented technology is indeed feasible in practice for real-world usage.

7.4 Threats to Validity

The major external threat to the validity of our results involves the study of a small set of benchmark programs developed and released by prior research work [20], so that we can directly compare our results with their previously reported results. To mitigate this threat and help determine whether our results may generalize, we conducted additional studies using 3732 real-world SmartThings and IFTTT apps. This enabled us to assess the capabilities of IotCom in real-world scenarios and capture violations that have not been discovered by prior work. The primary threat to internal validity involves potential errors in the implementations of IotCom and the infrastructure used to run IoTSAN. To overcome this threat, we extensively validated all of our tool components and scripts over the same baseline and configurations that have been used by prior work. The experimental data is also publicly available for external inspection. The main threat to construct validity relates to the fact that IotCom identifies all potential safety violations that can occur but do not yet assess whether these violations can happen in reality (e.g., when the heater is switched on, it is difficult to identify the time that will be taken until the temperature reaches a certain value because other factors can affect this process such as the size of the room).

8 DISCUSSION AND LIMITATIONS

For the sake of the feasibility demonstration of the presented ideas, this paper provides substantial supporting evidence for analyzing two of the most prominent IoT platforms, i.e., SmartThings home automation platform and IFTTT trigger-action platform. It would be interesting to see how IotCom fares when applied to safety and security analysis of other IoT platforms, such as HealthSaaS [31], Microsoft Flow [44] and Zapier [63]. Given that we developed IotCom based on the general trigger-condition-action paradigm, upon which these other IoT platforms are by and large relying, we believe the current analysis technique can be naturally extended to include such platforms. This forms a thrust of our future work.

Regarding the efforts required by end-users, IotCom uses static analysis to automatically infer apps behavior, captured in BRG models. It then automatically translates such behavioral models into formal specifications in the Alloy language. The formal analysis part is also conducted automatically. However, the specifications for the base smart home Alloy model and the safety properties are manually developed once and can be reused by others. Thus, it poses a one-time cost to develop new properties to be analyzed. Each app specification is automatically extracted, independent of the other apps. So if a new app is added, the only thing the user needs to do is to run IotCom's behavioral rule extractor to automatically extract the specification for the new app without changing the other apps. Note that we also automatically identified device-specific attributes by parsing the documentation of SmartThings to extract capabilities and actions of devices.

Similar to any approach that relies on static analysis, IotCom is subject to false positives. A fruitful avenue of future research is to strengthen IotCom by incorporating dynamic analysis techniques. In principle, it should be possible to leverage dynamic analysis techniques to automatically confirm some of the violations, and potentially enforce the required safety policies, further lessening the manual analysis effort. Moreover, dynamic analysis can address dynamic features in Groovy apps, as they support reflection and can make HTTP requests at runtime [18].

9 RELATED WORK

IoT safety and security has received a lot of attention recently [2, 4, 6, 16, 17, 23, 25, 30, 40, 42, 46, 48, 52–56, 58–62, 65]. Here, we provide a discussion of the related efforts in light of our research. As depicted in Table 4, we compare IotCom with the other existing approaches over various features provided by such analyzers. ContexIoT [38] analyzes individual IoT apps to prevent sensitive information leakage at run-time. However, it does not support the analysis of risky interactions among multiple apps. Soteria [20] and HOMEGUARD [24] are static analysis tools for detecting violations in multiple IoT apps. However, these techniques do not take into account physical channels, which can carry perilous interactions among apps, such as violations reported in section 7.2 as detected by IotCom. Moreover, none of these techniques can handle the interactions between IoT apps and trigger-action platform services.

Table 4: Comparing IotCom with the state-of-the-art IoT analysis approaches.

	ContexIoT [38]	SOTERIA [20]	HOMEGUARD [24]	IoTSAN [47]	IoTMON [26]	IotCom
Physical channel analysis	✗	✗	✗	✗	✓	✓
Trigger-action applet analysis	✗	✗	✗	✓	✗	✓
Multi-app analysis	✗	✓	✓	✓	✓	✓
Entire config. space analysis	✗	✗	✗	✗	✗	✓

Along the same line, IoTSAN [47] detects violations in bundles of more than two apps. However, IoTSAN [47] first translates the Groovy code of the SmartThings apps to Java, limiting its analysis to just less than half of the devices supported by SmartThings [50]. In contrast, IotCom directly analyzes Groovy code, supports large app bundles and all SmartThings device types, and is completely automated. IoTSAN [47], similar to Soteria [20] and HOMEGUARD [24], cannot detect violations mediated by physical channels.

IoTMON [26] is a purely static analysis technique that analyzes rules based solely on triggers, neglecting the conditions for specific actions. In contrast, IotCom validates the safety of app interactions with more precision by effectively capturing logical conditions influencing the execution of app rules through a precise control flow analysis. Moreover, IoTMON, similar to many other techniques we studied, does not support the analysis of interactions between IoT apps and trigger-action platform services. To the best of our knowledge, IotCom is the first IoT analysis technique for automated analysis of the entire configuration space, broadening the scope of the analysis beyond certain initial system configurations, required to specify in other existing techniques manually.

Other researchers have evaluated the security of IFTTT applets [3, 14, 27, 33]. Fernandes et al. [27] studied OAuth security in IFTTT, while Bastys et al. [14] used information flow analysis to highlight possible privacy, integrity, and availability threats. However, none of the studies examined the aforementioned IoT safety and security properties. In contrast, IotCom performs large scale safety and security analysis, examining interactions between tens of IFTTT smart home applets. IotCom also analyses bundles comprising both SmartThings Classic apps and IFTTT applets, demonstrating its unique cross-platform analytical capability.

10 CONCLUSION

This paper presents a novel approach for compositional analysis of IoT interaction threats. Our approach employs static analysis to automatically derive models that reflect behavior of IoT apps and interactions among them. The approach then leverages these models to detect safety and security violations due to interaction of multiple apps and their embodying physical environment that cannot be detected with prior techniques that concentrate on interactions within the cyber boundary. We formalized the principal elements of our analysis in an analyzable specification language based on relational logic, and developed a prototype implementation, IotCom, on top of our formal analysis framework. The experimental results of evaluating IotCom against prominent IoT safety and security properties, in the context of thousands of real-world apps, corroborates its ability to effectively detect violations triggered through both virtual and physical interactions.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by awards CCF-1755890 and CCF-1618132 from the National Science Foundation.

REFERENCES

[1] 2012. *D. Jackson, Software Abstractions, 2nd ed. MIT Press, 2012.* MIT Press.
[2] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and A Selcuk Uluagac. 2018. Peek-a-Boo: I see your smart home activities, even encrypted! *arXiv preprint arXiv:1808.02741* (2018).
[3] Ioannis Agadakos, Chien-Ying Chen, Matteo Campanelli, Prashant Anantharaman, Monowar Hasan, Bogdan Copos, Tancrède Lepoint, Michael Locasto, Gabriela F. Ciocarlie, and Ulf Lindqvist. 2017. Jumping the Air Gap: Modeling Cyber-Physical Attack Paths in the Internet-of-Things. In *Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and PrivaCy (CPS '17)*. ACM, New York, NY, USA, 37–48.
[4] Bako Ali and Ali Awad. 2018. Cyber and physical security vulnerability assessment for IoT-based smart homes. *Sensors* 18, 3 (2018), 817.
[5] Apple Homekit 2018. Apple HomeKit. https://www.apple.com/ios/home/.
[6] Noah Apthorpe, Dillon Reisman, and Nick Feamster. 2017. Closing the blinds: Four strategies for protecting smart home privacy from network observers. *arXiv preprint arXiv:1705.06809* (2017).
[7] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2018. A formal approach for detection of security flaws in the android permission system. *Formal Asp. Comput.* 30, 5 (2018), 525–544. https://doi.org/10.1007/s00165-017-0445-z
[8] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand Behrouz, and Sam Malek. 2016. Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 514–525. https://doi.org/10.1109/DSN.2016.53
[9] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Trans. Software Eng.* 41, 9 (2015), 866–886. https://doi.org/10.1109/TSE.2015.2419611
[10] Hamid Bagheri and Kevin J. Sullivan. 2013. Bottom-up model-driven development. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 1221–1224. https://doi.org/10.1109/ICSE.2013.6606683
[11] Hamid Bagheri and Kevin J. Sullivan. 2016. Model-driven synthesis of formally precise, stylized software architectures. *Formal Asp. Comput.* 28, 3 (2016), 441–467. https://doi.org/10.1007/s00165-016-0360-8
[12] Hamid Bagheri, Chong Tang, and Kevin J. Sullivan. 2014. TradeMaker: automated dynamic analysis of synthesized tradespaces. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 106–116. https://doi.org/10.1145/2568225.2568291
[13] Hamid Bagheri, Chong Tang, and Kevin J. Sullivan. 2017. Automated Synthesis and Dynamic Analysis of Tradeoff Spaces for Object-Relational Mapping. *IEEE Trans. Software Eng.* 43, 2 (2017), 145–163. https://doi.org/10.1109/TSE.2016.2587646
[14] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What?: Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1102–1119.
[15] Jorge Blasco, Thomas M. Chen, Igor Muttik, and Markus Roggenbach. 2018. Detection of app collusion potential using logic programming. *J. Network and Computer Applications* 105 (2018), 88–104. https://doi.org/10.1016/j.jnca.2017.12.008
[16] Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N Asokan. 2015. Smart and secure cross-device apps for the internet of advanced things. In *International Conference on Financial Cryptography and Data Security*. Springer, 272–290.
[17] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking

in Commodity IoT. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1687–1704.
[18] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2018. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *arXiv preprint arXiv:1809.06962* (2018).
[19] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel. 2018. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *CoRR* abs/1809.06962 (2018). arXiv:1809.06962 http://arxiv.org/abs/1809.06962
[20] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 147–158.
[21] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. arXiv:cs.CR/1805.08876
[22] Z. Berkay Celik, Gang Tan, and Patrick McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[23] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.
[24] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. *CoRR* abs/1808.02125 (2018).
[25] Chad Davidson, Tahsin Rezwana, and Mohammad A. Hoque. 2019. Smart Home Security Application Enabled by IoT:. In *Smart Grid and Internet of Things*, Al-Sakib Khan Pathan, Zubair Md. Fadlullah, and Mohamed Guerroumi (Eds.). Springer International Publishing, Cham, 46–56.
[26] Wenbo Ding and Hongxin Hu. 2018. On the Safety of IoT Device Physical Interaction Control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 832–846.
[27] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized action integrity for trigger-action iot platforms. In *22nd Network and Distributed Security Symposium (NDSS 2018)*.
[28] Garageio. 2019. IFTTT applet: If I arrive at my house then open my garage door. https://ifttt.com/applets/213296p.
[29] Google home 2018. Google Home. https://store.google.com/us/product/google$_{h}$ome?hl=en-US.
[30] Arnoud Goudbeek, Kim-Kwang Raymond Choo, and Nhien-An Le-Khac. 2018. A Forensic Investigation Framework for Smart Home Environment. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August 1-3, 2018*. IEEE, 1446–1451. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00201
[31] HEALTHSAAS 2020. HEALTHSAAS: THE INTERNET OF THINGS (IOT) PLATFORMFOR HEALTHCARE. https://www.healthsaas.net/..
[32] Households 2019. Households have 10 connected devices now, will rise to 50 by 2020, IT News, ET CIO. https://cio.economictimes.indiatimes.com/news/internet-of-things/households-have-10-connected-devices-now-will-rise-to-50-by-2020/53765773.
[33] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. 2019. SafeChain: Securing Trigger-Action Programming From Attack Chains. *IEEE Trans. Information Forensics and Security* 14, 10 (2019), 2607–2622. https://doi.org/10.1109/TIFS.2019.2899758
[34] IFTTT Documentation 2019. IFTTT Platform Documentation. https://platform.ifttt.com/docs.
[35] IotCom project website 2019. IotCom project website. https://sites.google.com/view/iotcom/home.
[36] iotmal 2019. IoTMAL benchmark app repository. https://github.com/IoTBench/IoTBench-test-suite/tree/master/smartThings/smartThings-Soteria.
[37] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290.
[38] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'17)*. San Diego, CA.
[39] Sylvain Kubler, Kary Främling, and Andrea Buda. 2015. A standardized approach to deal with firewall and mobility policies in the IoT. *Pervasive and Mobile Computing* 20 (2015), 100 – 114. https://doi.org/10.1016/j.pmcj.2014.09.005
[40] Brent Lagesse, Kevin Wu, Jaynie Shorb, and Zealous Zhu. 2018. Detecting Spies in IoT Systems using Cyber-Physical Correlation. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 185–190.
[41] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.* 88 (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001
[42] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically debugging IoT control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM, 133–142.
[43] Niloofar Mansoor, Jonathan A. Saddler, Bruno Silva, Hamid Bagheri, Myra B. Cohen, and Shane Farritor. 2018. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 785–790. https://doi.org/10.1145/3236024.3275534
[44] microsoftFlow 2020. microsoftFlow. https://flow.microsoft.com.
[45] Eugene M. Myers. 1981. A Precise Inter-procedural Data Flow Algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. ACM, New York, NY, USA, 219–230.
[46] Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. 2017. IOTA: A Calculus for Internet of Things Automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 119–133.
[47] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the Safety of IoT Systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, New York, NY, USA, 191–203.
[48] Amir Rahmati, Earlence Fernandes, Kevin Eykholt, and Atul Prakash. 2018. Tyche: A risk-based permission model for smart homes. In *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 29–36.
[49] SmartThings 2018. SmartThings Classic Documentation. https://docs.smartthings.com/en/latest/ref-docs/reference.html.
[50] SmartThings capabilities reference 2018. SmartThings Classic capabilities reference. https://docs.smartthings.com/en/latest/capabilities-reference.html.
[51] smartthings github 2018. SmartThings Community repository. https://github.com/SmartThingsCommunity/SmartThingsPublic.
[52] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1501–1510.
[53] A. Tekeoglu and A. Ş. Tosun. 2016. A Testbed for Security and Privacy Analysis of IoT Devices. In *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. 343–348.
[54] Himanshu Thapliyal, Nathan Ratajczak, Ole Wendroth, and Carson Labrado. 2018. Amazon Echo Enabled IoT Home Security System for Smart Home Environment. In *IEEE International Symposium on Smart Electronic Systems, iSES 2018 (Formerly iNiS), Hyderabad, India, December 17-19, 2018*. IEEE, 31–36. https://doi.org/10.1109/iSES.2018.00017
[55] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. 2017. Smartauth: User-centered Authorization for the Internet of Things. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Berkeley, CA, USA, 361–378.
[56] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 3227–3231.
[57] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1439–1453. https://doi.org/10.1145/3319535.3345662
[58] Qi Wang, Wajih Ul Hassan, Adam M. Bates, and Carl A. Gunter. 2018. Fear and Logging in the Internet of Things. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018$_{0}$1A-2$_{W}$ang$_{p}$aper.pdf
[59] Judson Wilson, Riad S Wahby, Henry Corrigan-Gibbs, Dan Boneh, Philip Levis, and Keith Winstein. 2017. Trust but verify: Auditing the secure internet of things. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 464–474.
[60] Fu Xiao, Le-Tian Sha, Zai-Ping Yuan, and Ru-Chuan Wang. 2017. VulHunter: A Discovery for unknown Bugs based on Analysis for known patches in Industry Internet of Things. *IEEE Transactions on Emerging Topics in Computing* PP (09 2017), 1–1. https://doi.org/10.1109/TETC.2017.2754103
[61] Muneer Bani Yassein, Wail Mardini, and Ashwaq Khalil. 2016. Smart homes automation using Z-wave protocol. *2016 International Conference on Engineering*

& MIS (ICEMIS) (2016), 1–6.

[62] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets-XIV)*. Article 5, 7 pages.

[63] zapier 2020. zapier. https://zapier.com/.

[64] Bruno Bogaz Zarpelão, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, and Sean Carlisto de Alvarenga. 2017. A survey of intrusion detection in Internet of Things. *J. Netw. Comput. Appl.* 84 (2017), 25–37. https://doi.org/10.1016/j.jnca.2017.02.009

[65] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. HoMonit: Monitoring Smart Home Apps from Encrypted Traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 1074–1088.