IoTCom: Dissecting Interaction Threats in IoT Systems

Mohannad Alhanahnah, Clay Stevens, Bocheng Chen, Qiben Yan, *Senior Member, IEEE*, and Hamid Bagheri, *Senior Member, IEEE*

Abstract—Due to the growing presence of Internet of Things (IoT) apps and devices in smart homes and smart cities, there are more and more concerns about their security and privacy risks. IoT apps normally interact with each other and the physical world to offer utility to the users. In this paper, we investigate the safety and security risks brought by the interactive behaviors of IoT apps. Two major challenges ensue in identifying the interaction threats: i) how to discover the threats across both cyber and physical channels; and ii) how to ensure the scalability of the detection approach. To address these challenges, we first provide a taxonomy of interaction threats between IoT apps, which contains seven classes of coordination threats categorized based on their interaction behaviors. Then, we present IoTCom, a compositional threat detection system capable of automatically detecting and verifying unsafe interactions between IoT apps and devices. IoTCom applies static analysis to automatically infer relevant apps' behaviors, and uses a novel strategy to trim the extracted app's behaviors prior to translating them into analyzable formal specifications, mitigating the state explosion associated with formal analysis. Our experiments with numerous bundles of real-world IoT apps have corroborated IoTCom's ability to effectively identify a broad spectrum of interaction threats triggered through cyber and physical channels, many of which were previously unknown. Finally, IoTCom uses an automatic verifier to validate the discovered threats. Our experimental results show that IoTCom significantly outperforms the existing techniques in terms of the computational time, and maintains the capability to perform its analysis across different IoT platforms.

Index Terms—Interaction threats, IoT safety, formal verification

1 Introduction

Internet-of-Things (IoT) ecosystems are becoming increasingly widespread, particularly in the smart home space. Industry forecasts suggest that 77 million users in the US smart home market by 2025 [1]. The race to configure, control, and monitor these IoT 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 adaptive automations. Such diversity enhances the user's experience by delivering many options for automating their home, but it comes at a price; it also expands the attack surface and results in safety and security threats. The increased complexity produced by the coordination and interaction of these apps subjects the system to the risk of undesirable behaviors due to 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 the 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 [2], [3], [4], [5] These research efforts have scrutinized deficiencies from various perspectives. However, existing detection techniques only target certain types of inter-app threats [3], [4], [6] 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 [3], [4], [5], [6]. Additionally, these analyses have been shown to experience scalability problems when applied to large numbers of IoT apps [3], [4], [6].

To address the foregoing challenges, this paper presents a novel technique, dubbed 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

- *Mohannad Alhanahnah is with the Department of Computer Science, University of Wisconsin-Madison, Madison, WI 53706 USA. E-mail: mohannad@cs.wisc.edu.*
- *Clay Stevens and Hamid Bagheri are with the School of Computing, University of Nebraska-Lincoln, Lincoln, NE 68588 USA. E-mail: clay.stevens@huskers.unl.edu, bagheri@unl.edu.*
- *Bocheng Chen and Qiben Yan are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA. E-mail: {chenboc1, qyan}@msu.edu.*

Manuscript received 13 Sept. 2021; revised 25 May 2022; accepted 26 May 2022. Date of publication 31 May 2022; date of current version 18 Apr. 2023.
This work was supported by National Science Foundation under Grants CCF-1755890, CCF-1618132, CCF-2139845, and CNS-1950171.
(Corresponding author: Mohannad Alhanahnah.)
Recommended for acceptance by D. Bianculli.
Digital Object Identifier no. 10.1109/TSE.2022.3179294

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 by linking the triggers, actions, and logical conditions of each control flow in each app. Unlike prior techniques, our approach respects the conditions (i.e., path sensitive) along each branch rather than only the triggers and actions. IoTCom then automatically generates formal app specifications from the BRG models. Therefore, the novelty of IoTCom 's static analysis resides in the holistic approach combining path sensitivity analysis, ICFG analysis, and BRG construction, allowing for a comprehensive analysis of IoT systems. In addition, the BRG abstraction step, unique in this work, optimizes the performance of the required model extraction. Lastly, it uses a lightweight formal analyzer [7] to check bundles of those models for violations of multiple safety and security properties arising from interactions among the apps rules.

Using a prototype implementation of IoTCom, we evaluated its ability to detect prominent classes of IoT coordination threats among thousands of publicly-available real-world IoT apps developed using diverse technologies. We further compare the precision of IoTCom against the other approaches using a set of benchmark IoT apps, developed by the other research groups [3], [5]. IoTCom is remarkably more successful in detecting safety violations. 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 compared to existing IoT threat detection techniques.

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.

This paper describes several new non-trivial extensions to the preliminary version of our work described in [8]. (1) We have added support for automatically verifying the safety and security violations discovered by IoTCom's analysis engine via checking the possibility of triggering the identified interactions threats. (2) We have expanded the capability of the Behavioural Rule Extractor by developing a dedicated translator that generates the formal specification for various formats of IoT applications (i.e., both code-based and text-based forms). (3) We report on additional and never-published empirical evaluations to further assess IoTCom's capabilities in detecting IoT interaction threats across new IFTTT dataset consisting of 989 apps, plus improving the statistical strength of the old results through additional experimentation. (4) We also present novel case studies to show the power of IoTCom.

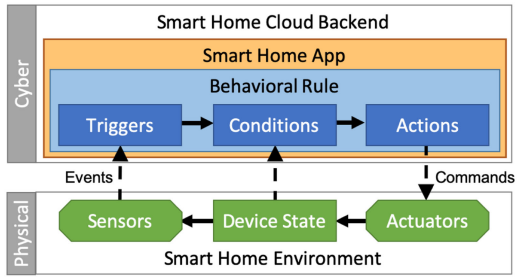


Fig. 1. Smart Home Automation Model. Apps installed in the cloud backend are triggered by physical sensors and conditions on the device state. Actions defined in the app are sent as commands to the physical actuators of the system.

On top of these technical contributions, the paper provides an expanded approach section to capture an in-depth description of IoTCom's static analysis component and a revamped discussion of IoTCom in the context of existing research.

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 [9] 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.
- *Tool implementation.* We develop IoTCom as a standalone analysis tool, backed by our formal analysis engine, for compositional analysis of IoT interaction threats. We make IoTCom research artifacts available to the research and education community, including the tool, all specifications, and experimental data. (details provided in Section 8.7).
- *Experimental evaluation.* We present our experiences with a thorough evaluation of IoTCom in the context of real-world IoT apps, developed for *multiple IoT platforms*.

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, as shown in Fig. 1. In popular smart home platforms, such as SmartThings [10], Apple's HomeKit [11], GoogleHome [12], Zapier [13] and MicrosoftFlow [14], physical devices installed in the home are registered with virtual proxies in a cloud-based

backend. Each proxy tracks the state of the device through one or more *attributes*, which can assume different *values*. The back-end also allows the user to install software *apps*, which automates 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 [4], [6], [15], particularly regarding software apps. Users can install multiple, arbitrary apps that can interact with not only 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 [5] via a shared metric acted upon by an *actuator* and monitored by a *sensor*; this is termed *indirect* coordination.

3 Taxonomy of Interaction Threats

In this section, we present a taxonomy of interaction threats between IoT apps. The main goal of this taxonomy is not to provide a conclusive list of all types of interaction threats, instead, it aims to underpin the understanding of models for tackling this intricate problem. The taxonomy comprises seven classes of potential multi-app coordination threats. We then provide the formal definition of each class. These classes capture the interaction behaviors of two IoT apps executing in the same environment. Since an IoT app consists of trigger-condition-action, its execution indicates a set of conditions are satisfied and specific actions are performed, ultimately affecting other IoT apps' triggers and conditions. Consequently, the main criteria for designing the classification in our taxonomy are to capture the different types of influences of satisfied conditions and actions of one rule/app on the triggers, conditions, and actions of other rules/apps.

We classify interaction threats into two main categories: *Direct Coordination* and *Chain Coordination*. Then, under each category, we consider a set of interaction threats as depicted in Fig. 2.

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.

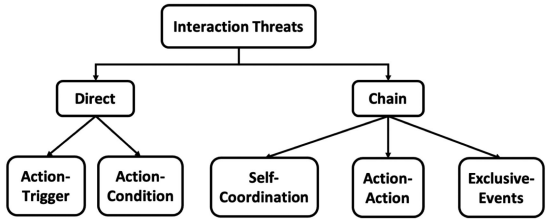


Fig. 2. Taxonomy of interaction Threats.

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 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*:*

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

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 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*:*

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

(T1) action-trigger coordination occurs when 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. This coordination is a core feature of IoT automation systems and is thus very common and often intended behavior. It can, however, lead to an unintended activation of subsequent rules if misused. Section 8.3.1 provides a real-world example based for this violation.

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. Sections 8.3.2 and 8.3.3 provide real-world examples of (*T2*) and (*T3*) violations, respectively.

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). If a rule is connected via action-trigger coordination to another rule, we refer to the triggered rule as a *descendant*.

Definition 3 (*descendant*). *Relation descendant holds between rules R_i and R_j if and only if there is an action a_i in R_i and a trigger t_j in R_j such that (a) a_i and t_j overlap via devices, attributes, and values or (b) a device in t_j is a sensor for a channel actuated by a_i and if none of the conditions in R_i violate any conditions in R_j.*

$$descendant(R_i, R_j) \equiv \Big((\exists a_i \in A_{R_i}, t_j \in T_{R_j} . match(a_i, t_j)) \vee same\_channel(R_i, R_j)\Big) \wedge (\forall c_1 \in C_{R_i}, c_2 \in C_{R_j}. (match(c_1, c_2) \Rightarrow (\mathbb{V}(c_1) \cap \mathbb{V}(c_2) \neq \emptyset)))$$

We refer to two rules that can be triggered by the same event as *siblings* if their conditions are not mutually exclusive.

Definition 4 (*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.*

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

(T4) self coordination: 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. Section 8.3.4 provides a real-world example based for this violation.

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.

(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. Section 8.4.1 provides a real-world example based for this violation.

(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. Section 8.4.2 provides a real-world example based for this violation.

(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. Section 8.3.6 provides a real-world example based for this violation.

4 Approach Overview

This section introduces IoTCom, a system that automatically determines whether the interactions within an IoT environment could compromise the safety and security thereof. Fig. 3 illustrates the architecture of IoTCom and its three major components, described in detail in the following sections:

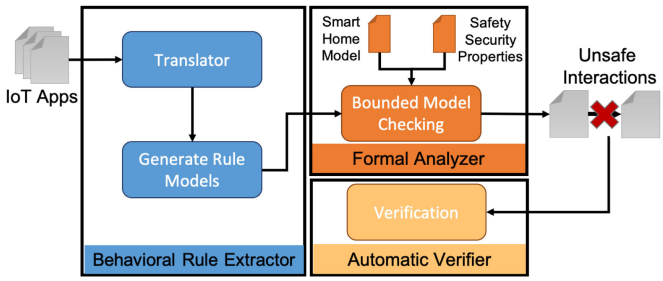


Fig. 3. IoTCom system overview.

(1) *Behavioral Rule Extractor (Section 5):* The *Behavioral Rule Extractor* component automatically infers models of the apps behavior using a novel translator that can handle various formats of IoT apps. The translator component performs static analysis, based on the format of the IoT app to generate the corresponding formal specification model. The translator then creates a behavioural rules 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. The importance of this component is threefold: (1) supporting cross-platform analysis by formalizing IoT apps in a unified format regardless of the underlying architecture, (2) enhancing the performance by eliminating irrelevant details, and (3) boosting the precision by capturing all trigger and condition requirements.

(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 as a whole are then checked for detecting violations of the properties.

(3) *Automatic Verifier (Section 7):* This component leverages SmartThings Emulator to verify the violations discovered by the Formal Analyzer. We automatically export the apps involved in the violations and the settings that led to each violation.

5 Behavioral Rule Extractor

The Behavioral Rule Extractor executes two main steps to automatically infer the behavior of individual IoT apps: (1) perform static analysis to infer behavioural rules. (2) generate formal models for the behavioral rules.

5.1 Translator

This component applies various static analysis techniques to infer the behavioural rules of IoT apps. It supports multiple formats of IoT app definitions, as different IoT platforms specify apps in different ways. For example, Samsung SmartThings

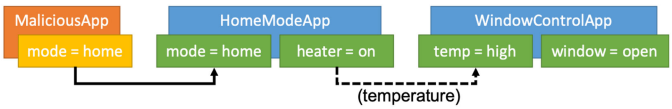


Fig. 4. An example of malicious IoT apps interaction.

Classic apps are *code-based* and written in Groovy, while IFTTT applets are *text-based* rules. Therefore, our translator handles these different formats without the need to perform any additional conversion, which might reduce the accuracy.

Listing 1. Malicious IoT App activates Home Mode

```
1:  preferences {
2:   section("Select Mode:")
   { input "HomeMode","mode"}}
3:   section("
   Presence sensor") { input "presence",
   "capability.presenceSensor"} }
4: def initialize() {
5:   subscribe(presence,
    "presence", presenceHandler)}
6: def presenceHandler(evt) {
7:   def evtValue = evt.value
8:   if (evtValue == "not present"){
9:    if (state.sunMode == "sunset")
      changeMode()
10:   else state.sunMode = "sunrise"
11:  } else {
12:   def timeStamp = new Date()
13:   log.debug
      "$timeStamp: status is $evtValue"} }
14: def changeMode() {
15:  if (location.mode != HomeMode) {
16:   setLocationMode(HomeMode)
17:  } else {
18:   mode = location.mode
19:   log.debug "Current mode is: $mode" } }
```

5.1.1 Code-Based Translator

As shown in Fig. 4 (adapted from [5], [16]), the example comprises three IoT code-based apps—one malicious (MaliciousApp) and two benign. HomeModeApp turns the heater on based on the "mode" of the smart home system. The mode is a general, customizable setting used for automation that generally tracks whether the user is home or away or if it is day or night. WindowControlApp opens the window when the temperature is higher than a certain threshold. MaliciousApp represents a third-party app that pretends to perform a benign activity but instead modifies the smart home's mode, unbeknownst to the home owner. Specifically, the detrimental interaction occurs when MaliciousApp switches the smart home mode to home after verifying that the user is away or not present (Line 8 in Listing 1). This triggers HomeModeApp, which turns on the heater. The heater may in turn activate the temperature sensor, triggering WindowControlApp to open the window. Together, the interaction of the apps compromises the safety of the home by *opening the window when no one is at home*.

Listing 1 shows the Groovy code defining the malicious app. Algorithm 1 describes the steps performed by the *Behavioral Rule Extractor* , detailed below, to derive models of a code-based IoT apps, starting with building the ICFG (lines 5-16).

Algorithm 1. Translator of Code-Based IoT Apps

INPUT: IoT App
OUTPUT: App: set of behavioral rules
1: App ← < {} >
2: ICFG ← {}, CFGs ← {}, CG ← {}, BRG ← {}
3: DevicesCap ← {}, Triggers ← {}
4: UserInput ← {}, GlobalVar ← {}
5: **// Step 1 : Generating ICFG**
6: **// Step 1.1: Entity Extraction**
7: DevicesCap ← extractDevicesCap(app)
8: UserInput ← extractUserInput(app)
9: GlobalVar ← extractGlobalVar(app)
10: Triggers ← extractTriggers(app)
11: **// Step 1.2: Generating ICFG**
12: **for each** $trigger \in Triggers$ **do**
13: CFGs ← constructCFG(trigger.entryMethod)
14: CG ← updateCG()
15: **end for**
16: ICFG ← constructICFG(CG, CFGs)
17: **// Step 2: Converting ICFG to BRG**
18: BRG ← constructBRG (ICFG, Triggers, DevicesCap, UserInput, GlobalVar)
19: **// Step 3: Generating Behavioral Rules**
20: **for each** $trigger \in Triggers$ **do**
21: App.R ← constructRules(BRG)
22: **end for**

To generate an ICFG, the Behavioral Rule Extractor first performs *Entity Extraction* to extract the information required to infer the rules in the app (Lines 6-10). Next, it creates a control flow graph for each trigger's entry method (Lines 11-15). The call graph will be updated while processing the entry method. An edge will be created between the caller (i.e., entry method) and the callee (i.e., local method). These graphs are combined to generate an inter-procedural control flow graph for the IoT app. Following is a detailed discussion of each step in Algorithm 1

Entity Extraction: This step extracts the following details: (1) the smart home devices and attributes altered/queried by this app; (2) any configuration values specified by the user, such as a desired setting for some device attributes; (3) any global variables used in the app; and (4) any events that trigger actions from the app, signified by use of certain APIs, and the methods invoked by those triggers.

The extraction algorithm traverses all statements in the AST, extracting the attached devices and user input from the `preferences` block (Listing 1, lines 1-3).

IoT apps are event-driven, so each subscription or scheduled call defines a distinct entry point. Triggers and entry methods are thus extracted by traversing the AST for calls to the `subscribe` , `schedule` , `runIn` , or `runOnce` API methods. For instance, a contact sensor device—identified in SmartThings by the ***contactSensor*** capability [17]—has a ***contact*** attribute representing the state of the sensor. The attribute can take two values, either ***open*** or ***closed***. Depending on the value, such a device can be formalized as $\langle$***contactSensor***, ***contact***, *closed*$\rangle$, or $\langle$***contactSensor***, ***contact***, ***open***$\rangle$. The extracted tuples are stored for later use in building the behavioral rule graph.

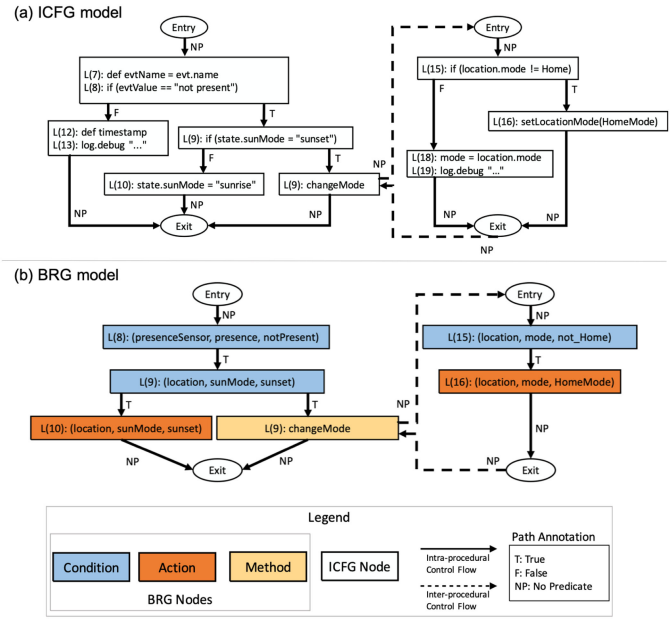


Fig. 5. Extracted models for *MaliciousApp* , described in Listing 1, at different steps of analysis.

Generating ICFG: In conjunction with Entity Extraction, the Behavioral Rule Extractor also generates a *call graph* and *control flow graph* for each user-defined method using a path-sensitive analysis. To construct an ICFG, each control flow graph is incorporated with the call graph at each trigger's entry point. Fig. 5a shows the ICFG corresponding to the malicious app code shown in Listing 1. The ICFG mode includes the CFG of the entry method `presenceHandler` (Fig. 5a left side), and the CFG of the local method `changeMode` (Fig. 5a right side). 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.

The holistic approach adopted by IoTCom via combining path sensitivity analysis, ICFG analysis, and BRG construction represents the novelty of the code-based translator, allowing for a comprehensive analysis of IoT systems. In addition, the BRG abstraction step, unique in this work, achieves performance optimization in the required model extraction.

5.1.2 Text-Based Translator

IFTTT applets are reactive rules that interact with REST services exposed via IFTTT's Connect API by third-party service providers [18]. Each applet consists of a single trigger-action pair, each of which provides a text-based service id and trigger/action id to describe the endpoints connected by the applet. The applets provided by each service can be queried via a GET request to the service's Connect API endpoint, providing a JSON response containing the trigger and action definitions for each applet.

The Behavioral Rule Extractor treats each applet as a standalone IoT app defining exactly one rule. It performs string analysis on the JSON defining the applet [19] to (a) extract the devices and attributes used by the applet from the ids of the trigger and action and (b) construct a rule based on the description of the applet itself. Specifically, the text-based translator relies on a set of heuristics to perform the translation process. For each trigger and action, the translator maintains a set of keywords extracted automatically from SmartThings. It starts by inspecting *triggerTitle* and *triggerChannelId* for constructing trigger nodes. Similarly, it inspects *actionTitle* and *actionChannelId* for constructing action nodes. Listing 2 illustrates the corresponding trigger and action nodes.

The overall process is defined in detail in Algorithm 2. Line 2 extracts the trigger information from the text of the applet via the "getTrigger" method, then Line 3 extracts the service id and trigger id from the trigger-specific text with "getIds". Lines 4-5 use the service and trigger id query a global map of devices and attributes, respectively, to find the device/attribute used by the trigger, or create a new device/attribute if the corresponding component has not yet been mapped (performed by the "getOrCreateX" methods). Lines 6-10 perform a similar set of operations for the action of the applet rather than the trigger. Finally, Line 12 generates the simple, three node BRG using triples containing the device and attribute found for each, and Line 14 constructs a behavioral rule for the BRG using the method described in Section 5.2.

Algorithm 2. Translator for text-based apps

INPUT: app: textual app definition
OUTPUT: $rule$: behavioral rule
1: **// Step 1.1: Get device/attribute from the trigger**
2: $trigger \leftarrow$ getTrigger(app)
3: $service\_id, trigger\_id \leftarrow$ getIds($trigger$)
4: $trigger\_dev \leftarrow$ getOrCreateDevice($service\_id$)
5: $trigger\_attr \leftarrow$ getOrCreateAttr($trigger\_dev, trigger\_id$)
6: **// Step 1.2: Get device/attribute from the action**
7: $action \leftarrow$ getAction(app)
8: $service\_id, action\_id \leftarrow$ getIds($action$)
9: $action\_dev \leftarrow$ getOrCreateDevice($service\_id$)
10: $action\_attr \leftarrow$ getOrCreateAttr($action\_dev, action\_id$)
11: **// Step 2. Create a BRG for the applet**
12: $BRG \leftarrow$ constructBRG($\langle trigger\_dev, trigger\_attr, true\rangle$, $\langle action\_dev, action\_attr, true\rangle$)
13: **// Step 3. Create the rule from the BRG**
14: $rule \leftarrow$ constructRules(BRG)

As a concrete example, the applet called "Open garage door when presence detected" [20] and its components are illustrated in Listing 2. The translator would use these various components of the IFTTT rule 2 to synthesize a rule initiated by the event "presence detected"—signified by a change to the "presences" attribute of the "presence sensor"—with an edge to the action "Switch on", invoking the "open_garage_door" action on the "switch" device.

Listing 2. Partial Specification of an IFTTT Applet, Displaying the IFTTT Rule Components

```
1:      {
2:    "actionChannelTitle":"SmartThings",
3:    "actionTitle":"Switch on",
4:    "title":"Open garage door
  when presence detected",
5:    "triggerChannelTitle":"SmartThings",
6:    "triggerTitle":"Presence detected",
7:  }
```

5.2 Generating Rule Models

The second 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 $\langle device, attribute, value \rangle$. 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., $\langle TheLock, locked, unlocked \rangle$. The devices, attributes, and values for Samsung SmartThings Classic apps are all defined as part of the SmartThings Classic API [17], allowing IoTCom to include pre-made values for each of the tuples used to model behavioral rules for those devices. For IFTTT applets, the devices, attributes, and values are instead extracted from the JSON response for each individual service, with the service id specified in each trigger/action taken to represent the device, and the trigger/action id representing an attribute of that device as in Algorithm 2.

To generate the models from the behavioural rules, IoTCom starts from each trigger component (which is used as the Trigger for the rule) and identifies corresponding action components; every rule must have at least one Action. From each action component, we identify the relevant condition to each pair of trigger-action.

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 the manual specification of the initial system configuration, which is required for comparable state-of-the-art techniques [3], [4]. We use Alloy [7] 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 [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33].

The bounded model checking relies on three sets of formal specifications, as shown in Fig. 3: (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*.

We make the complete Alloy models publicly available (see Section 8.7).

Listing 3. Excerpt of Base Smart Home Alloy Model

```
1:  abstract sig Device { attributes : set Attri-
    bute }
2:  abstract sig Attribute { values : set Value }
3:  abstract sig Value { }
4:  abstract sig IoTApp { rules : set Rule }
5:  abstract sig Rule {
6:  triggers : set Trigger,
7:  conditions : set Condition,
8:  actions : some Action }
9:  // Trigger, Condition, and Action contain
10: // similar tuples
11: abstract sig Trigger {
12: devices : some Device,
13: attribute : one Attribute,
14: values : set Value }
15: abstract sig Condition { ... }
16: abstract sig Action { ... }
```

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 3. 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.

Listing 4 Excerpt of Environment Model

```
1:  abstract sig Channel {
2:  sensors : set Capability,
3:  actuators : set Capability }
4:  one sig ch_temperature extends Channel {} {
5:  sensors = cap_temperatureMeasurement
6:  actuators = cap_switch + cap_thermostat +
    cap_ovenMode }
7:  one sig ch_luminance extends Channel {} {
8:  sensors = cap_illuminanceMeasurement
9:  actuators = cap_switch + cap_switchLevel }
10: one sig ch_motion extends Channel {} {
11:       sensors = cap_motionSensor +
          cap_contactSensor
12: actuators = cap_switch }
```

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 [3], [4]—directly supports detection of violations mediated via physical channels (cf. Listing 4). 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 5. Excerpts From the Generated Specification for MaliciousApp (Listing 1)

```
1:  one sig MaliciousApp extends IoTApp {
2:  presence : one PresenceSensor,
3:  location : one Location }
4:  { rules = r0 }
5:  one sig r0 extends Rule {}{
6:  triggers = r0_trg0
7:  conditions = r0_cnd0 + r0_cnd1
8:  actions = r0_act0 }
9:  one sig r0_trg0 extends Trigger {} {
10: devices = MaliciousApp.presence
11: attribute = PresenceSensor_Presence
12: no values }
13: one sig r0_cnd0 extends Condition {} {
14: devices = MaliciousApp.location
15: attribute = Location_Mode
16: values = Location_Mode.values - Location_Mo-
    de_Home }
17: one sig r0_cnd1 extends Condition {} { ... }
18: one sig r0_act0 extends Action {} {
19: devices = MaliciousApp.location
20: attribute = Location_Mode
21: values = Location_Mode_Home }
```

Listing 5 partially shows the Alloy specification generated for the MaliciousApp First, the new signature MaliciousApp extends the base IoTApp by adding fields for a PresenceSensor device and a Location as well as constraining the inherited rules field to contain only r0, defined on Line 5 as an extension 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 `presenceHandler` method is translated into the r0_trg0 signature (Line 9), while the condition nodes correspond with r0_cnd0 and r0_cnd1 (Lines 13, 17). Lastly, the action node from that path of the BRG generates r0_act0 (Line 18). Each of the apps analyzed would be translated into a similar specification; the bundle of these specifications define all apps in the system, analyzed by the bounded model checker. 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, security, and functionality properties. 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 [3], [4], [5] to define specific unsafe or undesirable behaviors that may result from chain triggering. In total, we consider 36 safety and security properties, 7 generic coordination threats introduced in Section 3) and 29 properties provided (see Section 8.7)

Listing 6. 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_con-
    nected[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 }
```

Listing 7 Example Alloy Assertion for the Property DON't Unlock Door/Window WHEN Location Mode is Away

```
1:  assert P8 {
2:  no r : IoTApp.rules, a : r.actions {
3:  // DON't open the door/window...
4:  a.attribute = CONTACT_SENSOR_CONTACT_ATTR
5:  a.values = CONTACT_SENSOR_OPEN
6:  // ... WHEN ...
7:  (some r' : r.*are_connected,
8:  a : r'.(triggers + conditions + actions) {
9:  // ...mode is away
10: a'.attribute = MODE_ATTR
11: a'.values = MODE_AWAY }) }}
```

As a concrete example of the seven coordination threats, the assertion $T4$ in Listing 6 corresponds to threat $T4$, presented in Section 3. In this snippet, we define the predicate are_connected (Lines 7-11) which encodes the relation 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. 1 and 2 from Section 3. The fact on Line 13 converts are_connected predicate to a field connected on the Rule signature. Line 17

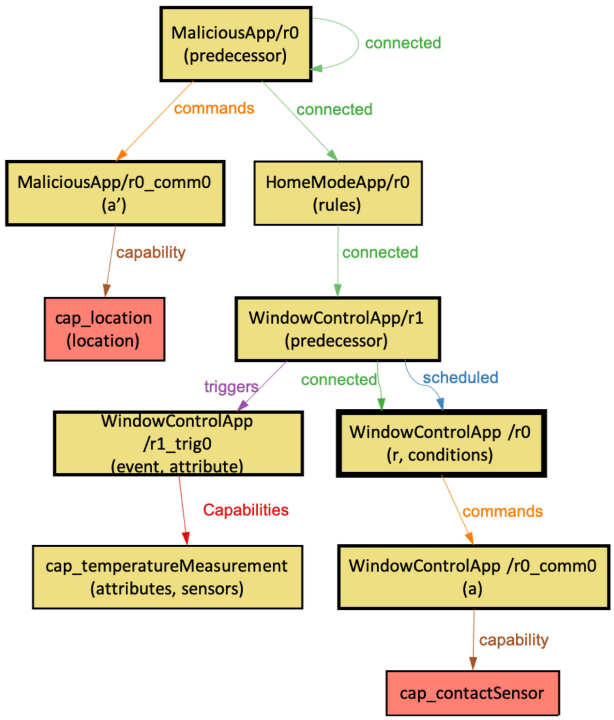


Fig. 6. An automatically detected violation scenario for our running example (cf. Fig. 4).

then defines the assertion, which ensures that no rule is found in the transitive closure of it's own connected rules.

As a concrete example of a safety properties caused due to action-trigger coordination (T1), Listing 7 illustrates the Alloy assertion for one of the fine-grained safety properties analyzed by IoTCom (P.8 in [8]). 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).

The property check is formulated as a problem of finding a valid trace that satisfies the specifications but violates the assertion. If the analyzer finds such a trace, it returns a *counterexample* for the property defining the specific assignments of tuples made for each relation, encoded as an XML document. In particular, the returned counterexample contains the sequence of rule activations leading to the violation as well as the initial configuration for each app, if applicable. Given our running example (cf. Fig. 4), the analyzer automatically detects a violation scenario, a visualization of which is shown in Fig. 6, where the four rules result in an unsafe physical state.

7 Automatic Verifier

This component verifies the discovered violations. It operates over two steps: *violation extraction* and *interaction simulator*.

7.1 Violation Extraction

To simulate the interactions among apps that lead to each violation, the Automatic Verifier requires two pieces of information: (a) the list of apps involved in the violation, and (b) the device capabilities of devices used within each app. The Automatic Verifier extracts these information from the XML generated by the Formal Analyzer for each detected violation scenario. Each recorded violation scenario contains a trace of rule activations that result in the violation, as well as potentially other rule activations that are not directly tied to the violation itself. To limit the scope of the verification, the Automatic Verifier first builds a directed graph of all rule activations in the trace, and then uses Dijkstra's algorithm to find the shortest path between the rules directly involved in the violation. We then identify the apps that define the rules along that shortest path, the collection of which form the list of apps whose interactions must be simulated by the verifier. The device capabilities are then extracted from each app involved in the violation and used to decide the device interaction category required for the verifier to check the possible violation.

7.2 Interaction Simulator

This step simulates the interactions between the IoT apps according to the exported settings in the previous step. We feed these settings and the violated property to our simulator to verify the violations that can be triggered. We leverage the simulator framework of IoTCheck [34]. This framework contains virtualized devices (i.e., device handlers) for the majority of the devices used by our dataset. It provides an execution model for SmartThings platform by supporting a set of virtual device handlers and necessary external environment settings. It simulates the execution of IoT apps across various physical and logical channels. Java PathFinder is used to identify violations during interactions between IoT applications. We feed the Groovy IoT apps to the simulator, which forms pairs of Groovy apps and automatically generates the necessary configuration setting based on the required device capabilities. Finally, we simulate and monitor the execution of SmartThings apps within a virtual machine to capture existing violations.

8 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:* Can IoTCom be extended to handle apps belong to parties other than SmartThings?
- *RQ4:* What is the performance of IoTCom's analysis realized atop static analysis and verification technologies?
- *RQ5:* What is the performance of the automatic verifier component of IoTCom?

Experimental Subjects. We used the IoTMAL suite of benchmarks [35], developed by other researchers, to perform a fair comparison of the existing analysis techniques. To further evaluate the implications of our tool in practice, we collected 3732 smart home apps drawn from two different repositories: *(1) SmartThings apps:* We gathered 404 SmartThings apps from the SmartThings public repository [36]. These apps are written in Groovy using the SmartThings Classic API platform. *(2) IFTTT applets:* We used the entire IFTTT dataset

provided by Bastys *et al.* [37]. This dataset is in JSON format, with each object defining an IFTTT applet. We further performed experiments on a newly obtained IFTTT dataset to supplement the results. This dataset contains 989 new IFTTT. This dataset is crawled from the IFTTT website [38].

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

8.1 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 [35] 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 that incorporate violations in both individual apps and bundles of apps. We also randomly partitioned the real-world IoT apps into 6 non-overlapping bundles, simulating a collection of apps installed on an end-user device. The bundles enabled us to perform several independent experiments.

We faced two challenges while evaluating the accuracy of IoTCom against the state-of-the-art: (1) Most analysis techniques—including HOMEGUARD [40], SOTERIA [3], and iRULER [41]—are not available; IoTSAN [4] was the lone exception. We also were not able to run IoTMON because only one component thereof is publicly accessible; its Groovy parser [1] is available, but the channels discovery and NLP components are not. IoTCheck [34] does not have the ability to identify safety violations in each individual app, since it is designed to only detect the violations across multiple apps. SOTERIA [3] was evaluated using the IoTMAL dataset, but the tool is not publicly available. Therefore, we rely on the results provided in the technical report [39].

(2) The violations in the IoTMAL dataset do not involve physical channels. To evaluate this capability of the compared techniques, we developed three bundles, B4–B6, available online (see Section 8.7).

Table 1 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[2] 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 (i.e., communicates the device status over http) which is not often used in SmartThings apps. Also, neither SOTERIA nor IoTSAN could detect this violation. IoTCom further outperforms IoTCheck because IoTCom can handle violation detection in each individual app and detect the conflicts between apps interacting via the physical channels, while IoTCheck cannot.

TABLE 1
Safety Violation Detection Performance Comparison Between SOTERIA, IoTSAN and IoTCom

Test Cases	SOTERIA*	IoTSAN	IoTCheck@	IoTCom
Individual Apps				
ID1BrightenMyPath	☑	☑	N/A	☑
ID2SecuritySystem	☑	☐†	N/A	☑
ID3TurnItOnOffandOnEvery	☑	☐	N/A	☑
ID4PowerAllowance	☑☐	(☐2)	N/A	(☑2)
ID5.1FakeAlarm	☐	☐	N/A	☐
ID6TurnOnSwitchNotHome	☑	☑	N/A	☑
ID7ConflictTimeandPresence	☑	☐‡	N/A	☑
ID8LocationSubscribeFailure	☑	☑	N/A	☑
ID9DisableVacationMode	☒	☐	N/A	☑
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%	**100%**
Recall	66.7%	25%	16.6%	**93.8%**
F-measure	76.6%	40%	27.6%	**96.8%**

**Results obtained from [39].*
@IoTCheck does not handle individual Apps (ID1-ID9).
†IoTSAN did not generate the Promela model.
‡SPIN crashing.
#Benchmarks involving physical channels related violations.

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. IoTcheck can identify apps that have device interactions, but it only has a limited support for a small set of device handlers. Therefore, IoTCheck's performance is unsatisfactory, since it cannot detect any of the violations that involve unsupported devices.

8.2 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 [4], [42]. 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 inde-

1. https://github.com/nsslabcuus/IoTMon
2. Although the number of individual apps is 9, but the app *ID4PowerAllowance* has two known violations.

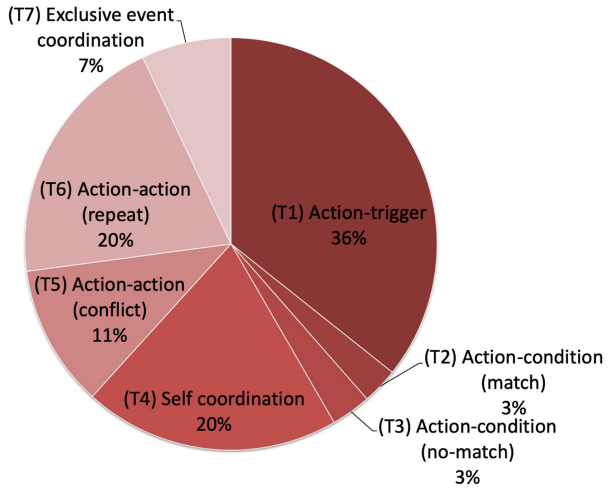


Fig. 7. Distribution of the detected violations across seven classes of multi-app coordination.

pendent 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 [3], [4], [5], 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 (see Section 8.7). Overall, IoTCom detected 2883 violations across the analyzed bundles of real-world IoT apps, with analysis failing on one of the 622 bundles.

Fig. 7 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 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.

8.3 Manual Analysis

We manually inspect a sample of the real-world bundles from Section 8.2 to verify the reality of the detected interaction threats. We first randomly selected 30 of the 622 analyzed bundles (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.

Our manual analysis corroborates the recognition rate from the automatic verification, with a recall rate of 94.92% on the 30 bundles drawn from those used in Section 8.2. However, the precision (47.86%) and F1-measure (62.64%) were low, in part due to re-use of identifiers in IFTTT "channel" definitions as well as overloading of "switch" and "notification" devices/channels. For example, the service controlling Android phones re-uses the same channel

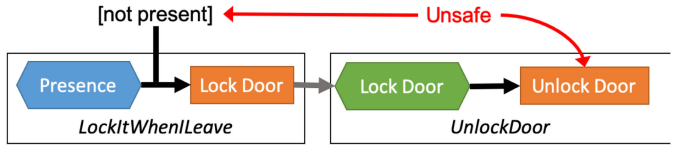


Fig. 8. Example violation of *T1*: Cyber coordination between apps may leave the door unlocked when no one is home. The first rule is guarded by a condition that the home owner not be present.

ID for multiple tasks, including detecting a connection to a particular WiFi network, changing the volume on the phone, and sending a notification, among things. Conversely, there are numerous different brands of smart switch devices, each of which has their own identifier but all share "switch" devices. Our reference implementation of IoTCom identifies devices based in part on those identifiers—both in distinguishing them from one another and in identifying members of shared device types—so some rule activations may be flagged as a violation even though the actual action taken by the rule would not result in any coordination.

As an example case, we performed manual verification on a bundle identified as AeFLPm, for which IoTCom reported violations of T1, T4, and T6. Of those three, our manual analysis determined that T1 and T6 were in fact violated, whereas T4 was not (a false positive). For T1, IFTTT applet "Check if rain then send Android notication" was determined to be a valid trigger for applet "When it starts raining, basement dehumidifier goes on", which is activated by a notification such as that sent by the first app. The T6 violation came from what is likely a spam applet called "l o ve=V!A!s!H!iK!a!r!a!n SPECIALIST molvi ji in ENGLAND 08107470632"; this applet contained a variety of rules which sent the same (spam) notification repeatedly, violating T6 (same command sent multiple times). The false positive for T4 was also related to the general "notification" device type. IoTCom detected a self-activation loop in the applet "Check if rain then send Android notication" due to the notifications sent by other applets in the bundle, but failed to recognize that the offending applet is only triggered by notifications from "Weather Underground", rather than generic Android notifications. Thus, this bundle does not present a real violation of T4.

In the rest of this section, we describe a few representative findings for each interaction threat described in Section 3.

8.3.1 Violation of (T1) Action-trigger

Fig. 8 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. Meanwhile, 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.

8.3.2 Violation of (T2) Action-condition-match

Action-condition coordinations where commands and conditions match may result in outcomes that run contrary to expectations. For example, the app *smoke_alert* includes a

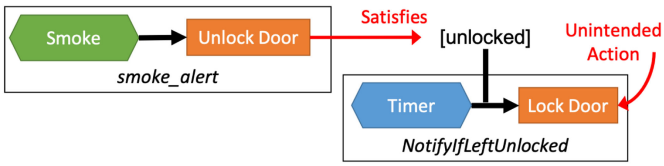


Fig. 9. Example violation of *T2*: *smoke_alert* app enables the condition for *NotifyIfLeftUnlocked*, which unintentionally undoes the action of *smoke_alert*.

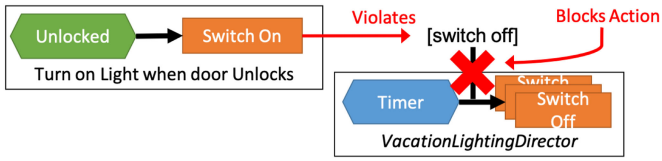


Fig. 10. Example violation of *T3*: "Turn on Light when door Unlocks" applet may unexpectedly block the actions of *VacationLightingDirector* app.

rule that unlocks a door if smoke is detected to allow easy exit/entry in case of fire. If incorrectly configured, however, this could satisfy a condition for one of the rules in app *NotifyIfLeftUnlocked*, which—upon identifying the lock is unlocked—would re-lock the door. This goes contrary to the intent of the first rule, and would likely signal a misconfiguration, as shown in Fig. 9.

8.3.3 Violation of ($T3$) Action-condition-not-match

Action-conditions coordination where the results do not match may also lead to unexpected outcomes, as in the following example from our real-world results. The app *VacationLightingDirector* contains a rule that randomly turns on a light only if a given set of switches are off; the IFTTT applet "Turn on Light when door Unlocks" may turn one of those switches *on*, which may unexpectedly prevent *VacationLightingDirector* from operating until the switch is turned off (shown in Fig. 10).

8.3.4 Violation of ($T4$) Self Coordination via Physical Channel

The chain of interactions shown in Fig. 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.

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

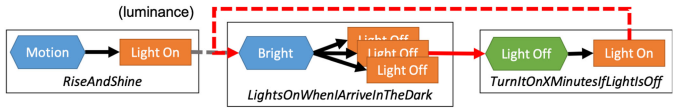


Fig. 11. Example violation of *T4*: Lights continually turn off and on. Dashed line represents coordination via the *luminance* physical channel.

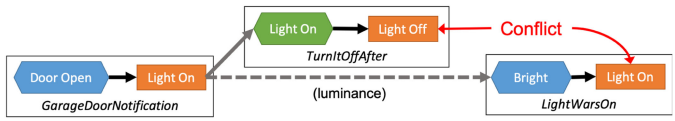


Fig. 12. Example violation of *T5*: Both "on" and "off" commands sent to the same light due to the same event. Dashed line represents coordination via the *luminance* physical channel.

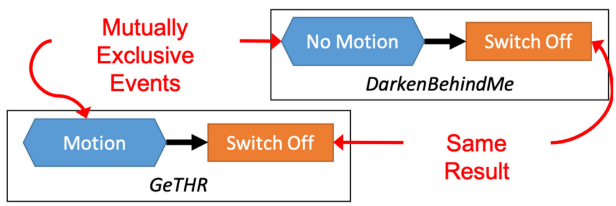


Fig. 13. Example violation of *T7*: Mutually exclusive events lead to the same, guaranteed result.

physical channels, scheduling APIs, and arbitrarily long chains of interactions among apps.

8.3.5 Violation of ($T5$) Action-Action (Conflict) via Physical Channel

The three apps shown in Fig. 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.

8.3.6 Violation of ($T7$) Exclusive-Event-Coordination

This violation (shown in Fig. 13) involves the two Groovy apps *DarkenBehindMe* and *GeTHR*. Both apps share the same device (i.e., motion sensor) and attribute (i.e., motion activity), but they are triggered based on different values: inactive motion triggers *DarkenBehindMe* and active motion triggers *GeTHR*. Since IoTCom detects this violation, this supports the user to avoid situations when two apps are unintentionally triggered based on the activity of one device.

8.4 Results for RQ3 (Extendability)

We extended our safety violation analysis of IoTCom on a new IFTTT dataset by considering IFTTT rules that belong to parties other than SmartThings. This evaluation shows

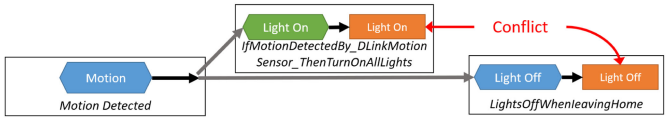


Fig. 14. The detected T5 threat example: *IfmotiondetectedbyDLinkMotionSensorthenturnonAlllights* and *Lightsoffwhenleavinghome* are triggered by a motion when person is leaving and the conflict happens to light.

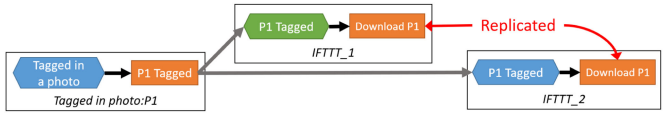


Fig. 15. The detected T6 violation example: IFTTT_1 (*Download_newFacebookPhotos_youretagged_ intoDropbox*) and IFTTT_2 (*Download_allphotos_inwhichIamtagged_ tomyDropbox*) are triggered by *Tagged_one_photo* action and the photo P1 is downloaded twice.

IoTCom is not only limited to analyze the SmartThing apps, but it is also flexible to handle apps that belong to various parties. We used a set of 989 new IFTTT applets crawled from the IFTTT website by Ur *et al.* [38]. Rules and device/attribute definitions were extracted from each applet as described in Algorithm 2 in Section 5.1.2. We randomly divide the existing 989 IFTTT apps into 123 non-overlapping bundles, each containing 8 apps. With the widespread use of SmartThings, more and more users are running multiple applications simultaneously on their phones and we expand the size of the bundle to fit a more general scenario [43]. At the same time, using random segmentation as in previous experiments does not restrict user choices.

IoTCom discovered violations in 22 bundles of real-world IoT apps. Each bundle includes 8 applets, which may result in multiple types of violations. When multiple apps are interacting frequently with each other, the race conditions among the actions arise during their interactions. This, in turn, leads to frequent violations of T6. Such race conditions can also lead to another Threat (T5) action-action (conflict), which can appear along with T6 threats.

8.4.1 Violation of (T5) Action-action (conflict)

Fig. 14 presents another case of violating $T5$. When the smart presence sensor *dlink_motion_sensor* detects a person leaving, it triggers *Lightsoffwhenleavinghome* and turns the switch off after the information is transmitted to hue. However, when the motion sensor catches the movement, it triggers *IfmotiondetectedbyDLinkMotionSensorthenturnonAlllights* and switches on the light controlled by hue. Another $T5$ conflict with such similar approach occurs during the interaction of *Lightsoffwhenleavinghome* and *Turnonwhensunset* . When a person leaves home, the first IFTTT applet is triggered and the lights controlled by hue will turn off. When the sensor detects sunset and triggers the second IFTTT applet, here comes the violation, i.e., the hue receives turn-on light command and a race condition occurs.

8.4.2 Violation of (T6) Action-action (repeat)

Fig. 15 illustrates a safety violation that involves two IFTTT applets, *Download_newFacebookPhotos_youretagged_intoDropbox* and *Download_allphotos_inwhichIamtagged_tomyDropbox* . These two applets execute the download_image_toDropbox action when they receive the trigger of tagged_a_photo. These two rules conform to the same device, attribute, and value. So when both IFTTT rules are used at the same time, the same photo will be downloaded to the Dropbox twice (corresponding to T6 threat), leading to a waste of the storage space.

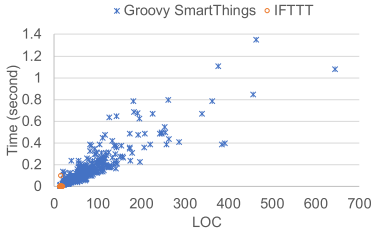
8.5 Results for RQ4 (Performance and Timing)

This section evaluates the analysis time required by different phases of IoTCom. It describes the overhead incurred by the model extraction and formal analysis based on real-world apps drawn from the SmartThings and IFTTT repositories.

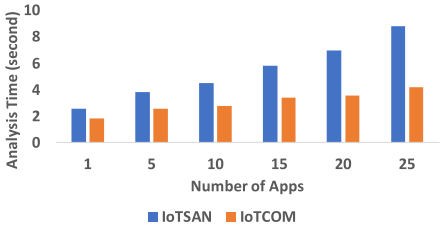
Fig. 16a presents the time taken by IoTCom to extract rule models from the Groovy SmartThings apps and IFTTT applets. The scatter plot shows both the analysis time and the app size. According to the results, our approach statically analyzes and infer specifications for 98% of apps in less than one second.

Fig. 16b presents the comparison results between IoTCom and IoTSAN. In all different bundle sizes. The results show over 52% of reduction in the analysis time in comparison with IoTSAN, because BRG optimizes the performance by trimming the ICFG and eliding the edges and nodes that do not impact the app's behavior apropos of physical devices.

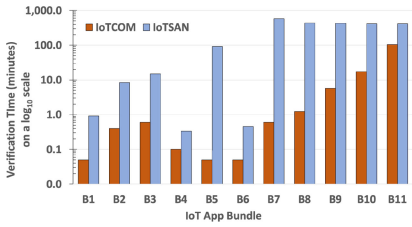
We also measured the verification time required for detecting safety/security violations and compared the analysis time of IoTCom against that required by IoTSAN [4].



(a) Scatter plot representing analysis time for behavioral rule extraction of IoT apps using IoTCom.



(b) Static analysis time (secs) for IoTCom vs. IoTSAN by number of apps analyzed. IoTCom requires less time to generate its behavioral rule models directly from Groovy.



(c) Comparing verification time by IoTCom and IoTSAN to perform the same safety violation detection across 11 bundles of real-world apps (in minutes, on a $\log_{10}$ scale).

Fig. 16. RQ4 results.

TABLE 2
Safety Violation Detection Performance Verified by the Automated Verifier

Bundle#	Verified violations	Description of conflicts verified by IoTcheck
1	√	Executing App1(PortableSwitchturnsonotherdevice) triggers the condition of App2 (LaLaManorAutomation)
2	√	Executing App1(BrightenMyPath) triggers the condition of App2(LightupwhenImgone)
3	×	The simulator doesn't support this capacity
4	√	Executing App1(LightsOffWhenClosed) triggers the condition of App2(Turnonorofftogether)
5	√	Executing App1(TurnOnOnlyIfIArriveAfterSunset) triggers the condition of App2 (UndeadEarlyWarning)
6	√	Executing App1(MySmartAppSensorToLightBulb) triggers the condition of App2(helloworld)
7	√	Executing App1(OpenGarage) triggers the condition of App2(UndeadEarlyWarning)
8	√	Executing App1(TurnItOnWhenItOpens) triggers the condition of App2(MonitoronSense)
9	√	Executing App1(MonitoronSense) triggers the condition of App2 (Turnsomethingoffwhenacontactopens)
10	√	Executing App1(Wireless3Way) triggers the condition of App2(LightsOffWhenClosed)
11	√	Executing App1(DelayedCommandExecution) triggers the condition of App2 (PowerAllowanceSeconds)
12	√	Executing App1(GarageDoorAutomation) triggers the condition of App2(GarageDoor)
13	√	Executing App1(TurnItOnFor5Minutes) triggers the condition of App2 (Turnsomethingoffwhenacontactopens)
14	√	Executing App1(TurnItOffAfter) triggers the condition of App2(LightTurnonClosed)
15	√	Executing App1(StrobeWhenIamHomeandSomeone Arrives) triggers the condition of App2 (LightTurnonClosed)
16	√	App1 triggers the condition making it no longer satisfy the condition of app2
17	√	Executing App1(TurnItOnFor5Minutes) triggers the condition of App2(LightupwhenImgone)
18	√	Executing App1(TurnItOnWhenItOpens) triggers the condition of App2(BrightenMyPath)

The violations found by IoTCom show the details of the threats, while symbols √, × indicate "verified" and "unverified", respectively.

We checked all 36 safety and security properties against the app bundles. IoTSAN requires the initial configuration of 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 a format supported by IoTSAN. We then ran the two techniques considering all valid initial configurations to avoid missing any violation.

Fig. 16c 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.

8.6 Results for RQ5 (Automatic Verification)

Next, we evaluate the automatic verifier component of IoTCom, which validates and reflects the accuracy of IoTCom detection results. Because IoTCheck dynamically detects whether two apps will cause the same device to have the opposite instruction state at a certain moment, we exclude the pair of apps that does not contain device interaction. Moreover, we adjust the parameters according to the corresponding device category of the software. The detection results of IoTCheck mainly involve two types of interaction threats i.e., T5 and T3. Combining the applet pair excluded above, we analyzed 39 groovy applets, which are combined into 18 bundles with different conflicts.

The verification results are presented in Table 2. Based on the experimental results, it can be seen that the static analysis results of IoTCom can be dynamically verified, and the recognition rate reaches 94.4% (17/18) as shown in Table 2. One single violation is not verifiable due to the capacity of the applets inside the bundle not satisfying the simulator requirement.

8.7 Data Availability

The IoTCom implementation is available on GitHub[3] as well as on Zenodo[4]. Both repositories contain the implementation of the Behavioral Rule Extractor and the Formal Analyzer components of IoTCom. The repositories also contain the IFTTT dataset used in this work, the corresponding Alloy models, and the definitions of safety assertions.

9 Discussion and Limitations

Usability of IoTCom. Regarding the efforts required by end-users, IoTCom uses static analysis to automatically infer apps behavior, captured in BRG models. It then autom-

3. https://github.com/Mohannadcse/IoTCOM
4. https://doi.org/10.5281/zenodo.6513163

atically 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.

Translator Correctness. IoTCom can generate false alarms due to two following reasons:

1) Code-based Translator: this translator relies on static analysis; however, it does not handle dynamic features in Groovy apps, such as reflection. This limitation can be addressed by incorporating dynamic analysis.
2) Text-based Translator: this translator is prone to miss IFTTT rules that do not match the set of heuristic rules applied by string analysis. This limits the translator's capability to map triggers and actions in the JSON format of IFTTT rules to the corresponding tuples required by our model. Other techniques like Natural Language Processing (NLP) and lexer grammar can be applied to enhance the performance of the translator.
3) IoTCom models the physical interactions at the application level by defining the mapping between physical channels. This can lead to an over-approximation of physical channels. An interesting avenue for future work is to improve modeling physical interactions by considering the physical properties of actuator devices.

Scope of Properties. Section 3 proposes seven categories of interaction threats, which can lead to undesirable behaviours. Also, in Section 6.3, we describe the capability of IoTCom to handle not only the interaction threats in Section 3 but also a set of safety properties. The safety properties represent concrete examples of the interaction threats. Since IoTCom captures the influence across the components of IoT apps (i.e., triggers, conditions, and actions), IoTCom is not limited to identify safety violations. In fact, IoTCom users can define any undesirable behaviours as Alloy assertions. For example, the violation in Fig. 15 can be considered as a functionality violation that misuses storage because the picture will be stored twice.

10 Related Work

In this section, we provide a discussion of the related efforts in light of our research. IoT safety and security has received a lot of attention recently [15], [38], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65]. Table 3 shows a comparison of IoTCom with the other existing approaches over various features provided by such analyzers. ContexIoT [6] 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 [3] and HOMEGUARD [40] are static analysis tools for detecting violations in multiple IoT apps. Taifu [66] and IoTWatcH [67] apply dynamic testing approach for detecting privacy violations. However, these techniques do not take into account physical channels, which can carry perilous interactions among apps, such as violations reported in Section 8.2 as detected by IoTCom. Although IoTCheck [34] considers conflicted interactions over physical channels. Furthermore, these techniques are vendor-dependent and do not support cross-platform interaction, since they cannot handle the interactions between IoT apps and trigger-action platform services, except Taifu [66] and IoTWatcH [67]. While VISCR [68] aims to detect and resolve conflicts of automation rules by considering cross-vendor interaction, it does not exhaustively explore the configuration-space. Therefore, IoTCom and VISCR are orthogonal to each other: IoTCom can improve VISCR's conflict detection capabilities, while IoTCom can benefit from the resolution component of VISCR. Both IoTCom and VISCR maintain a translator component to provide the cross-vendor interactions, but it is unclear whether VISCR supports: (1) interactions over physical channels and (2) conditions in Groovy apps.

TABLE 3
Comparing IoTCom With the State-of-the-art IoT Analysis Approaches

System	Physical channel analysis	Platform independent	Multi-app analysis	Entire config. space analysis
ContexIoT [6]	✗	✗	✗	✗
SOTERIA [3]	✗	✗	✓	✗
IoTSAN [4]	✗	✓	✓	✗
IoTMON [5]	✓	✗	✓	✗
IoTCheck [34]	✓	✗	✗	✗
VISCR [68]	N/A	✓	✓	✗
IoTCom	✓	✓	✓	✓

Along the same line, IoTSAN [4] detects violations in bundles of more than two apps. However, IoTSAN [4] 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 [17]. In contrast, IoTCom directly analyzes Groovy code, supports large app bundles and all SmartThings device types, and is completely automated. IoTSAN [4], similar to Soteria [3] and HOMEGUARD [40], cannot detect violations mediated by physical channels.

IoTMON [5] 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 [37], [42], [69], [70]. Fernandes *et al.* [69] studied OAuth security in IFTTT, while Bastys *et al.* [37] 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.

11 Conclusion

This paper presents IoTCom, a novel approach for identifying unsafe interactions between IoT apps. 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 and further validate the violations triggered through both virtual and physical interactions and across different IoT platforms.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] "Smart home - United States | statista market forecast," 2021. [Online]. Available: https://www.statista.com/outlook/dmo/smart-home/united-states
[2] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel, "Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities,"2018, *arXiv:1809.06962*.
[3] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 147–158.
[4] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel, "IotSan: Fortifying the safety of IoT systems," in *Proc. 14th Int. Conf. Emerg. Netwo. Experiments Technol.*, 2018, pp. 191–203.
[5] W. Ding and H. Hu, "On the safety of IoT device physical interaction control," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 832–846.
[6] Y. J. Jia *et al.*, "ContexIoT: Towards providing contextual integrity to appified IoT platforms," in *Proc. 21st Netw. Distrib. Syst. Secur. Symp.*, 2017, p. 2.
[7] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002.
[8] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in IoT systems," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 272–285.
[9] D. Jackson, *Software Abstractions*, 2nd ed. Cambridge, MA, USA: MIT Press, 2012.
[10] "SmartThings classic documentation," 2018. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/reference.html
[11] "Apple HomeKit," 2018. [Online]. Available: https://www.apple.com/ios/home/
[12] "Google home," 2018. [Online]. Available: https://store.google.com/us/product/google_home?hl=en-US
[13] "zapier," 2020. [Online]. Available: https://zapier.com/
[14] "microsoftflow," 2020. [Online]. Available: https://flow.microsoft.com
[15] Z. B. Celik *et al.*, "Sensitive information tracking in commodity IoT," in *Proc. 27th USENIX Conf. Secur. Symp.*, 2018, pp. 1687–1704.
[16] C. Stevens, M. Alhanahnah, Q. Yan, and H. Bagheri, "Comparing formal models of IoT app coordination analysis," in *Proc. 3rd ACM SIGSOFT Int. Workshop Softw. Secur. Des. Deployment*, 2020, pp. 3–10, doi: 10.1145/3416507.3423188.
[17] "SmartThings classic capabilities reference," 2018. [Online]. Available: https://docs.smartthings.com/en/latest/capabilities-reference.html
[18] "IFTTT platform documentation," 2019. [Online]. Available: https://platform.ifttt.com/docs
[19] Z. B. Celik, G. Tan, and P. McDaniel, "IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019.
[20] tasi, "IFTTT applet: Open garage door when presence detected," 2021. [Online]. Available: https://ifttt.com/applets/u2rRXLWy-open-garage-door-when-presence-detected
[21] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "COVERT: Compositional analysis of android inter-app permission leakage," *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 866–886, Sep. 2015.
[22] S. G. Brida *et al.*, "Bounded exhaustive search of alloy specification repairs," in *Proc. 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1135–1147.
[23] H. Bagheri, A. Sadeghi, R. J. Behrouz, and S. Malek, "Practical, formal synthesis and automatic enforcement of security policies for android," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016, pp. 514–525.
[24] C. Stevens and H. Bagheri, "Reducing run-time adaptation space via analysis of possible utility bounds," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1522–1534.
[25] H. Bagheri and K. J. Sullivan, "Model-driven synthesis of formally precise, stylized software architectures," *Formal Aspects Comput.*, vol. 28, no. 3, pp. 441–467, 2016.
[26] N. Mansoor, J. A. Saddler, B. V. R. E. Silva, H. Bagheri, M. B. Cohen, and S. Farritor, "Modeling and testing a family of surgical robots: An experience report," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2018, pp. 785–790.
[27] H. Bagheri, C. Tang, and K. J. Sullivan, "TradeMaker: Automated dynamic analysis of synthesized tradespaces," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 106–116.
[28] H. Bagheri and K. J. Sullivan, "Monarch: Model-based development of software architectures," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.*, 2010, pp. 376–390.
[29] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 559–570.
[30] H. Bagheri, J. Wang, J. Aerts, N. Ghorbani, and S. Malek, "Flair: Efficient analysis of android inter-component vulnerabilities in response to incremental changes," *Empir. Softw. Eng.*, vol. 26, no. 3, 2021, Art. no. 54.
[31] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "A formal approach for detection of security flaws in the android permission system," *Formal Aspects Comput.*, vol. 30, no. 5, pp. 525–544, 2018.
[32] H. Bagheri, C. Tang, and K. J. Sullivan, "Automated synthesis and dynamic analysis of tradeoff spaces for object-relational mapping," *IEEE Trans. Softw. Eng.*, vol. 43, no. 2, pp. 145–163, Feb. 2017.
[33] H. Bagheri, J. Wang, J. Aerts, and S. Malek, "Efficient, evolutionary security analysis of interacting android apps," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 357–368.
[34] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, G. H. Xu, and S. Lu, "Understanding and automatically detecting conflicting interactions between smart home IoT applications," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2020, pp. 1215–1227.
[35] "IoTMAL benchmark app repository," 2019. [Online]. Available: https://github.com/IoTBench/IoTBench-test-suite/tree/master/smartThings/smartThings-Soteria
[36] "SmartThings Community repository," 2018. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic

[37] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in IoT apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1102–1119.
[38] B. Ur *et al.*, "Trigger-action programming in the wild: An analysis of 200, 000 IFTTT recipes," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2016, pp. 3227–3231, doi: 10.1145/2858036.2858556.
[39] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," 2018, *arXiv:1805.08876*.
[40] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," 2018, *arXiv:808.02125*.
[41] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action IoT platforms," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1439–1453.
[42] K. Hsu, Y. Chiang, and H. Hsiao, "SafeChain: Securing trigger-action programming from attack chains," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 10, pp. 2607–2622, Oct. 2019.
[43] "Global smart home market revenue 2016–2022," 2021. [Online]. Available: https://www.statista.com/statistics/682204/global-smart-home-market-size
[44] Y. Tian *et al.*, "Smartauth: User-centered authorization for the internet of things," in *Proc. 26th USENIX Conf. Secur. Symp.*, 2017, pp. 361–378.
[45] Q. Wang, W. U. Hassan, A. M. Bates, and C. A. Gunter, "Fear and logging in the Internet of Things," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.
[46] 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 *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 1501–1510.
[47] N. Apthorpe, D. Reisman, and N. Feamster, "Closing the blinds: Four strategies for protecting smart home privacy from network observers," 2017, *arXiv:1705.06809*.
[48] B. Lagesse, K. Wu, J. Shorb, and Z. Zhu, "Detecting spies in IoT systems using cyber-physical correlation," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops*, 2018, pp. 185–190.
[49] A. Rahmati, E. Fernandes, K. Eykholt, and A. Prakash, "Tyche: A risk-based permission model for smart homes," in *Proc. IEEE Cybersecurity Develop.*, 2018, pp. 29–36.
[50] J. Chen *et al.*, "IOTFUZZER: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.
[51] A. Acar *et al.*, "Peek-a-boo: I see your smart home activities, even encrypted!," 2018, *arXiv:1808.02741*.
[52] F. Xiao, L.-T. Sha, Z.-P. Yuan, and R.-C. Wang, "Vulhunter: A discovery for unknown bugs based on analysis for known patches in industry Internet of Things," *IEEE Trans. Emerg. Topics Comput.*, vol. 8, no. 2, pp. 267–279, Jul.–Sep. 2017.
[53] C. Busold, S. Heuser, J. Rios, A.-R. Sadeghi, and N. Asokan, "Smart and secure cross-device apps for the internet of advanced things," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2015, pp. 272–290.
[54] C.-J. M. Liang *et al.*, "Systematically debugging IoT control system correctness for building automation," in *Proc. 3rd ACM Int. Conf. Syst. Energy-Efficient Built Environments*, 2016, pp. 133–142.
[55] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things," in *Proc. 14th ACM Workshop Hot Topics Netw.*, 2015, pp. 5:1–5:7.
[56] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "HoMonit: Monitoring smart home apps from encrypted traffic," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1074–1088.
[57] B. Ali and A. Awad, "Cyber and physical security vulnerability assessment for IoT-based smart homes," *Sensors*, vol. 18, no. 3, 2018, Art. no. 817.
[58] H. Thapliyal, N. Ratajczak, O. Wendroth, and C. Labrado, "Amazon echo enabled iot home security system for smart home environment," in *Proc. IEEE Int. Symp. Smart Electron. Syst.*, 2018, pp. 31–36.
[59] C. Davidson, T. Rezwana, and M. A. Hoque, "Smart home security application enabled by IoT," in *Smart Grid and Internet of Things*, A.-S. K. Pathan, Z. M. Fadlullah, and M. Guerroumi, Eds., Cham, Switzerland: Springer, 2019, pp. 46–56.
[60] A. Goudbeek, K. R. Choo, and N. Le-Khac, "A forensic investigation framework for smart home environment," in *Proc. 17th IEEE Int. Conf. Trust Secur. Privacy Comput. Commun./12th IEEE Int. Conf. Big Data Sci. Eng.*, 2018, pp. 1446–1451.
[61] M. B. Yassein, W. Mardini, and A. Khalil, "Smart homes automation using z-wave protocol," in *Proc. Int. Conf. Eng. MIS*, 2016, pp. 1–6.
[62] J. Wilson, R. S. Wahby, H. Corrigan-Gibbs, D. Boneh, P. Levis, and K. Winstein, "Trust but verify: Auditing the secure internet of things," in *Proc. 15th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2017, pp. 464–474.
[63] A. Tekeoglu and A. S. Tosun, "A testbed for security and privacy analysis of IoT devices," in *Proc. IEEE 13th Int. Conf. Mobile Ad Hoc Sensor Syst.*, 2016, pp. 343–348.
[64] J. L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, and M. Sridharan, "IOTA: A calculus for Internet of Things automation," in *Proc. ACM SIGPLAN Int. Symp. New Ideas New Paradigms Reflections Program. Softw.*, 2017, pp. 119–133.
[65] Y. Chen, M. Alhanahnah, A. Sabelfeld, R. Chatterjee, and E. Fernandes, "Practical data access minimization in Trigger-Action platforms," in *Proc. 31st USENIX Secur. Symp.*, 2022.
[66] K. Mahadewa *et al.*, "Identifying privacy weaknesses from multi-party trigger-action integration platforms," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2021, pp. 2–15.
[67] Z. Celik *et al.*, "Real-time analysis of privacy-(un) aware IoT applications," in *Proc. Privacy Enhancing Technol. Symp.*, 2021, pp. 145–166.
[68] V. Nagendra, A. Bhattacharya, V. Yegneswaran, A. Rahmati, and S. Das, "An intent-based automation framework for securing dynamic consumer IoT infrastructures," in *Proc. Web Conf.*, 2020, pp. 1625–1636.
[69] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action IoT platforms," in *Proc. 22nd Netw. Distrib. Secur. Symp.*, 2018.
[70] I. Agadakos *et al.*, "Jumping the air gap: Modeling cyber-physical attack paths in the Internet-of-Things," in *Proc. Workshop Cyber-Phys. Syst. Secur. PrivaCy*, 2017, pp. 37–48.

Mohannad Alhanahnah received the PhD degree in computer engineering from the University of Nebraska-Lincoln. He is currently a research associate with the Department of Computer Science, University of Wisconsin-Madison. His research interests include spans over the area of software security and adversarial machine learning.

Clay Stevens received the PhD degree in computer science from the University of Nebraska-Lincoln, with an emphasis in software engineering. Prior to starting his PhD, Clay spent 13 years in industry as a professional software engineer and architect. His research interests aims to scientifically study how software engineers and architects work in practice and to improve the scalability of rigorous formal analysis tools to enable their use on large-scale, real-world problems.

Bocheng Chen received the BS degree in electronic science and technology from Shanghai Jiaotong University, Shanghai, China, in 2020. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI. His research interests include mobile security, AI security, and IoT security.

Qiben Yan (Senior Member, IEEE) received the BS and MS degrees in electronic engineering from Fudan University in Shanghai, China, and the PhD degree in computer science from Virginia Tech. He is currently an assistant professor with the Department of Computer Science and Engineering, Michigan State University. He is a recipient of NSF CRII Award in 2016. His current research interests include wireless communication, wireless network security and privacy, mobile and IoT security, and Big Data privacy.

Hamid Bagheri (Senior Member, IEEE) received the PhD degree in computer science from the University of Virginia. He is an associate professor with the School of Computing, University of Nebraska-Lincoln (UNL). He is a co-director of the ESQuaReD Laboratory, UNL. Prior to joining UNL, he was a postdoctoral researcher with the University of California, Irvine and Massachusetts Institute of Technology. His general research interests include the field of software engineering, and to date, his focus has spanned the areas of software analysis and testing, applied formal methods, and dependability analysis. He has received numerous awards for his research contributions, including the EPSCoR FIRST Award and the National Science Foundation CRII Award. He is a member of IEEE, ACM, and ACM SIGSOFT.