# Understanding and Automatically Detecting Conflicting Interactions between Smart Home IoT Applications

Rahmadi Trimananda
University of California, Irvine
USA
rtrimana@uci.edu

Seyed Amir Hossein Aqajari
University of California, Irvine
USA
amiraj.95@uci.edu

Jason Chuang
University of California, Irvine
USA
chuangj6@uci.edu@uci.edu

Brian Demsky
University of California, Irvine
USA
bdemsky@uci.edu

Guoqing Harry Xu
UCLA
USA
harryxu@cs.ucla.edu

Shan Lu
University of Chicago
USA
shanlu@uchicago.edu

## ABSTRACT

Smart home platforms allow developers to write apps to make smart home devices work together to accomplish tasks, *e.g.*, home security and energy conservation. A smart home app typically implements narrow functionality and thus to fully implement desired functionality homeowners may need to install multiple apps. These different apps can conflict with each other and these conflicts can result in undesired actions such as locking the door during a fire.

In this paper, we study conflicts between apps on Samsung SmartThings, the most popular platform for developing and deploying smart home IoT devices. By collecting and studying 198 official and 69 third-party apps, we found significant app conflicts in 3 categories: (1) close to 60% of app pairs that access the same device, (2) more than 90% of app pairs with physical interactions, and (3) around 11% of app pairs that access the same global variable. Our results suggest that the problem of conflicts between smart home apps is serious and can create potential safety risks. We then developed an automatic conflict detection tool that uses model checking to automatically detect up to 96% of the conflicts.

## CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Empirical software validation**.

## KEYWORDS

smart home apps, concurrency, program analysis, model checking

**ACM Reference Format:**
Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, Guoqing Harry Xu, and Shan Lu. 2020. Understanding and Automatically Detecting Conflicting Interactions between Smart Home IoT Applications. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409682

## 1 INTRODUCTION

Smart home devices are widely available commercially. Modern smart home platforms support developers writing apps that implement useful functionality on smart devices. Significant efforts have been made to create integration platforms such as Android Things from Google [43], SmartThings from Samsung [68], and the open-source openHAB platform [61]. All of these platforms allow users to create *smart home apps* that integrate multiple devices and perform more complex routines, such as implementing a home security system.

In this work, we focus on Samsung's SmartThings platform because it is the de-facto smart home development environment and has the most extensive collection of smart home apps, including those officially created by SmartThings [67] and those developed by third-party companies and hobbyists. Homeowners that use SmartThings can install any of these SmartApps and run them simultaneously in their home deployment. Many of these apps each implement a specific functionality, *e.g.*, turn off lights in the absence of motion. Thus, homeowners will likely need to install multiple apps that collectively achieve the desired functionality.

### 1.1 The Problem

***Interactions and Conflicts of Apps.*** The presence of multiple apps that can control the same device creates interactions that can potentially be undesirable (*i.e.*, conflicts). For example a homeowner may install the `FireCO2Alarm` [63] app which, upon the detection of smoke, sounds alarms and *door-unlocks*[1]. The same homeowner may also install the `Lock-It-When-I-Leave` [14] app to *door-lock* automatically when the homeowner leaves the house.

While it may appear that these apps can be safely installed together, closer examination reveals that they can interact in surprising ways. Consider the following scenario. If smoke is detected, `FireCO2Alarm` will door-unlock the door. If someone leaves home with the presence tag, this will make the presence sensor change its state from `"present"` to `"not present"`, causing the `Lock-It-When-I-Leave` app to door-lock the door. This defeats the intended purpose of the `FireCO2Alarm` app. Thus, the two apps *conflict.*

***Data Races, Atomicity Violations.*** Interactions of smart home apps may initially appear similar to those of concurrent programs, including data races [37, 38, 54] and atomicity violations [41, 55, 81].

[1]We use *door-lock* and *door-unlock* to refer to actions on a physical door, and *lock* and *unlock* to refer to synchronizations in concurrent programming.

Data races can be resolved by acquiring locks appropriately, while atomicity violations can be resolved by ensuring that locks are held long enough to guarantee that a thread can finish all operations in a batch without interference from other threads.

Unfortunately, these techniques cannot resolve the above-mentioned conflict. Suppose that we use a lock to guarantee the atomicity of the critical region of the code—the `FireCO2Alarm` app needs to acquire the lock before triggering the alarm and holds the lock while the alarm is sounding. Similar actions need to be taken to door-lock and door-unlock for the `Lock-It-When-I-Leave` app. However, this approach could disable the desirable functionality of the apps. To illustrate, consider a scenario in which the `Lock-It-When-I-Leave` app detects that someone leaves the house. It then acquires the lock before it enters the critical region in which door-lock is performed. It holds the lock to keep the door locked until the person returns. In this period, if the `FireCO2Alarm` app detects smoke/fire and attempts to door-unlock, it will fail because the `Lock-It-When-I-Leave` app holds the lock. We end up in the same situation: *the door is locked during a fire!*

***Feature Interaction.*** Feature interaction considers the problem in which different software features can have negative interactions [26–28, 33, 48, 62]. Our setting differs from most of the previous work in this area in that smart home apps are developed independently and composed by end users. For example, SmartThings apps are distributed through many different channels (including pay for source). Thus, there does not exist a means to detect/resolve/avoid conflicts during development. Feature interactions have also been studied in research prototypes for home automation [50, 64, 77]. These early systems were prototype systems, and presumed much coarser apps (*e.g.*, a single app for lighting) than current smart home apps implement. HCI researchers have shown that feature interactions in IoT systems make it difficult for users to understand the systems' behavior [80]. In rule-based smart home systems, researchers have developed tools for repairing incorrect rules [58].

***Interactions of Mobile Apps.*** Researchers have also studied interactions between Android apps [29, 32, 34, 44, 49, 51, 73]. However, these techniques focus primarily on cross-app information flow/-taint analysis via ICC/IAC mechanisms in Android (*e.g.*, Intents) and thus cannot be used in our setting. In particular, our problem requires checking *execution trace* and its *properties* that such analyses cannot handle (see Section 3.1).

***The Smart Home App Interaction Problem.*** The problem we focus on in this work is *conflict of expectations*. The expected result of the `Lock-It-When-I-Leave` app is that the door should be *locked* when the homeowner leaves, while the expected result for the `FireCO2Alarm` app is that the door should be *unlocked* during a fire. These expectations conflict in certain scenarios. Hence, the fundamental question here is *what should be the expected state of the door when these apps interact*: *locked* or *unlocked*? The potential conflict between the `FireCO2Alarm` and `Lock-It-When-I-Leave` apps is *not* correctable using standard mechanisms for *concurrent accesses to program variables or entities*—using locks to restore atomicity still violates *the integrity of the expected result.*

***State-of-the-art and Our Work.*** The research community has been actively looking into smart home apps. There is a body of work that aims to find bugs and issues that could lead to serious security problems [25, 30, 35, 36, 39, 40, 69, 82]. However, none of these techniques focuses on interactions and conflicts between multiple apps. In the cyber-physical systems community, work has been done to identify and resolve conflicts between smart home apps at the system level, viewing apps as *black boxes* [57, 74, 75, 78, 79]. While such techniques are useful in certain simple scenarios, they are still semantics-agnostic and do not work even for the above-mentioned conflicts. Understanding the semantics of the apps (*e.g.*, scheduling of events) is key to build a tool that can automatically detect the conflicts—these conflicts can typically be exposed only when a specific set of events occur in a certain order.

IA-Graph [52, 53] studies smart-home app conflicts and proposes a lightweight approach to check for conflicts. This work extracts an SMT formula that describes the legal transitions for an app and then uses an SMT solver to detect whether a set of apps has conflicting transitions. As acknowledged in the IA-Graphs paper, IA-Graphs "ignores complicated computations in the app code"—they are, in fact, used either (1) in condition statements, or (2) to update the device state—and hence the patterns it finds are limited. In addition, not all transitions in an app can be expressed in SMT, further limiting the kinds of conflicts IA-Graphs can detect. Another important drawback is IA-Graphs do not check whether a conflicting transition is reachable in an execution and hence can produce many false positives—it may incorrectly label non-conflicting apps as conflicting. Understanding the impact of these issues is quite difficult without accessing their implementation. Since the authors did not release any software, an empirical evaluation is not possible. Furthermore, they did not perform any wide-scale study. Instead, they devised and evaluated their approach only on a relatively small corpus: 22 apps.

On the contrary, our study covers a broader range of interaction patterns in a much larger corpus of apps. Our conflict detection tool, IoTCheck, works for arbitrarily complicated application logic since it model-checks all app pairs directly using the app code.

## 1.2 Our Contributions

The goal of this paper is to understand the nature of the interactions between smart home apps. We have identified the following five research questions to guide our study.

**RQ1: What kinds of *interactions* are there?** We have collected and studied 198 official SmartThings apps and 69 third-party apps. Compared with recent studies of smart home apps [35, 36, 69, 82], we have among the largest app suite. To understand interactions and possible conflicts, we analyzed these apps in pairs (see Section 3) and examined all pairs of apps that can potentially interact. We discovered three main categories of interactions: (1) interactions between apps that access the same device (see Section 4), (2) interactions between apps such that the output from one app interferes with the input of the other app (e.g., via sensors, see Section 5), and (3) interactions between apps accessing global variables, *e.g.*, whether the home is in the *Home* or *Away* mode (see Section 6).

**RQ2: What types of *conflicts* arise between smart home apps?** For an app pair, we first inspected their source code and documentation to understand the intended behavior of each individual app and then reason about possible interactions between them. If there exists an interaction that can compromise the desired functionality of either app, we say that this pair has a *conflict*, *e.g.*,

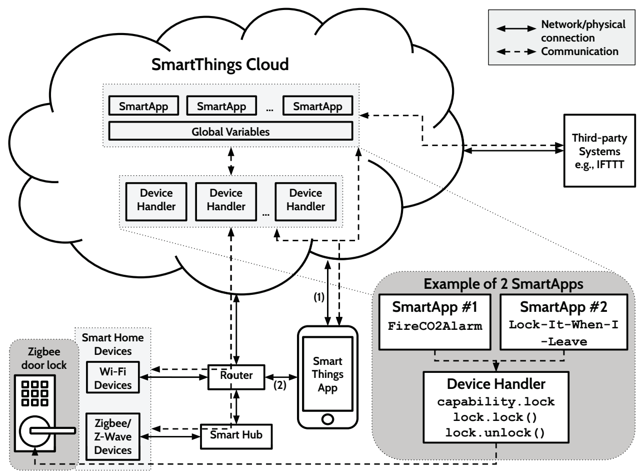

**Figure 1: SmartThings platform with an example of two apps running in parallel.**

the functionality of the `FireCO2Alarm` app to door-lock is compromised by the `Lock-It-When-I-Leave` app. Our goal is to carefully inspect apps that interact, and understand whether they conflict and if they do, why.

**RQ3: How prevalent are these conflicts?** We summarized the results of our study to understand how prevalent the conflicts are. We found that almost 60% of pairs in the first category, more than 90% of pairs in the second category, and around 11% of pairs in the third category have conflicts (see Sections 4.3, 5.2, and 6.3).

**RQ4: Are there common coding patterns that are unsafe in the presence of app interactions?** During our study, we observed several common programming idioms that often result in problematic interactions between apps. Discovering and classifying these idioms can help developers mitigate potential conflicts by avoiding these idioms.

**RQ5: How can we automatically detect conflicts?** Based on our findings, we develop a tool that can automatically detect conflicts (see Section 7). Our tool and dataset are available under an open source license at `http://plrg.ics.uci.edu/iotcheck/` [70–72].

***Implications.*** The implications of this work are two-fold. First, our study opens a new research direction in the area of testing and verification of concurrent programs where the development of different apps are done completely independently. The inability of existing concurrency control mechanisms to resolve smart home apps dictates the need of new techniques (such as IoTCheck) to detect and/or repair these conflicts. Second, for platform vendors such as Google and Samsung, new APIs should be designed and applied to these platforms so that app developers can be directed to make more informed decisions during development even if they are not aware of potential runtime conflicts.

## 2 BACKGROUND

This section provides an overview of SmartThings [68], the de-facto smart home IoT development platform.

***Components.*** Figure 1 shows an overview of the SmartThings platform. There are three main components, as discussed shortly. The network/physical connections between these components are shown in Figure 1 as solid lines, while dashed lines represent communication paths.

**(1) Smart Home Devices:** SmartThings supports both SmartThings-branded and third-party devices as well as a variety of communication protocols, including Wi-Fi, Zigbee, and Z-Wave. While Wi-Fi devices are connected directly to the home router, Zigbee/Z-Wave devices are connected to a SmartThings smart hub through dedicated radios. The smart hub is connected to the home router and relays the communication between the Zigbee/Z-Wave devices and the SmartThings cloud via the router. Classes of devices that are supported by the SmartThings platform include both actuators (*e.g.*, switches, locks, thermostats, lights, or alarms) and sensors (*e.g.*, illuminance, motion, water, or sound sensors).

**(2) SmartThings Cloud:** The SmartThings cloud hosts smart home apps (*i.e.*, SmartApps) and device handlers (*i.e.*, drivers that directly control devices) developed using an event-based programming model in Groovy [42], a managed language running on top of the Java Virtual Machine (JVM). SmartApps implement desired functionalities on smart home devices by accessing global variables and device features through *capabilities* exposed by *device handlers*. For instance, a door lock can be accessed by SmartApps through its device handler that declares lock-related capabilities using `capability.lock`. These capabilities provide access to features such as *door-lock* and *door-unlock* via APIs such as `lock()` and `unlock()`. Third-party systems, *e.g.*, IFTTT (If-This-Then-That) [21], can also connect to the SmartThings cloud and control smart home devices through SmartApps that expose HTTP endpoints as a control interface.

**(3) SmartThings Smartphone App:** Homeowners can use the SmartThings smartphone app to install devices and SmartApps. To communicate with home devices, the smartphone first connects and sends control information to the SmartThing cloud either over the Internet or via the home router, illustrated by arrows (1) and (2) in Figure 1, and then the SmartThings cloud forwards the information to smart home devices via the home router and the smart hub.

***Execution Model.*** SmartThings uses an event-driven execution model and allows multiple SmartApps to run concurrently. Consider for example the `FireCO2Alarm` app [63], which attempts to door-unlock if it detects smoke/fire through a smoke sensor. The app subscribes to the events generated by the sensor's device handler: when the sensor detects smoke/fire, it sends a message to the smart home hub. The smart home hub relays the message to the SmartThings cloud, which in turn runs the sensor's device handler to process the message. The device handler will generate an event and send it to the app's event handler method, which in turn calls another method `takeActions()` to door-unlock. Since multiple apps run concurrently, the two apps `FireCO2Alarm` and `Lock-It-When-I-Leave` share the device handler for the door lock, and thus can both execute `lock()` and `unlock()` at any time on the same device handler. The device handler on the cloud translates each action into device specific commands. The cloud then sends these commands to the local smart hub, which forwards the commands to the door lock.

## 3 METHODOLOGY

This section describes our research methodology. We first define several terms. Next, we discuss our database of smart home apps and

$$\begin{array}{rcl}
\mathcal{X} \in \text{Execution} & = & (\text{Action} \mid \text{Event} \mid \text{Update})^* \\
\mathcal{A} \in \text{Action} & = & \text{read}(\alpha, d, \tau, r) \mid \text{write}(\alpha, d, \tau, r, v) \mid \\
 & & \text{moderead}(\alpha) \mid \text{modewrite}(\alpha, \mu) \mid \\
 & & \text{schedule}(\alpha, t, m) \\
\mathcal{V} \in \text{Event} & = & \text{devEv}(\alpha, d, \tau, r, v) \mid \text{modeEv}(\alpha, \mu) \mid \\
 & & \text{schedEv}(\alpha, m) \\
\mathcal{U} \in \text{Update} & = & \text{devUp}(\alpha, d, \tau, r, v) \mid \text{modeUp}(\alpha, \mu)
\end{array}$$

$$\alpha \in \text{App} \quad d \in \text{DeviceID} \quad \tau \in \text{DeviceType} \quad r \in \text{Feature}$$
$$v \in \text{Value} \quad t \in \text{Time} \quad \mu \in \text{Mode} \quad m \in \text{Method}$$

**Figure 2: SmartThings Execution Traces.**

the way we structure them for the study. Our study focuses on pair-wise interactions. The rationale is that pair-wise interactions are fundamental for understanding multi-app interactions since multi-app interactions can be decomposed to pair-wise interactions for reasoning about. Upon carefully observing how these apps interact in bigger groups, we did not see any new interaction patterns that manifest only when three or more apps are involved.

## 3.1 Definitions

***Execution Traces.*** We first formalize our notion of execution traces for SmartThings in Figure 2. The traces can be generated by one or more apps that run concurrently. An execution $\mathcal{X} \in$ Execution from a set of apps is a sequence of the following:

(1) Action: App $\alpha$ performs an action $\mathcal{A} \in$ Action by executing any of the following set of operations:

- read($\alpha$, $d$, $\tau$, $r$) and write($\alpha$, $d$, $\tau$, $r$, $v$), which read from and write a value $v$ to a feature $r$ of a device with ID $d$ and device type $\tau$, respectively;
- moderead($\alpha$) and modewrite($\alpha$, $\mu$), which read from and write a new mode $\mu$ to the `location.mode` variable, respectively; and
- schedule($\alpha$, $t$, $m$), which schedules a method $m$ to run at time $t$.

(2) Event: An event $\mathcal{V} \in$ Event is either:

- devEv($\alpha$, $d$, $\tau$, $r$, $v$), a device event is delivered to app $\alpha$ from device $d$ to notify the app of device status update;
- modeEv($\alpha$, $\mu$), a mode event is delivered to app $\alpha$ to notify it of a mode change; or
- schedEv($\alpha$, $m$), a schedule event denotes when the framework processes a schedule action and executes the method $m$ in app $\alpha$.

(3) Update: An update $\mathcal{U} \in$ Update is an external input to the smart home. It is either:

- devUp($\alpha$, $d$, $\tau$, $r$, $v$), an update with a new value $v$ generated from a device with ID $d$ and type $\tau$ for feature $r$ and value $v$, i.e., a sensor reading a temperature change; or
- modeUp($\alpha$, $\mu$); an update with a new mode $\mu$, e.g., the homeowner manually setting a new mode.

***Interacts-with Relation.*** We next define a relation *interacts-with* over the domain of Apps × Apps where Apps is the set of all smart home apps. A pair of apps ($\alpha_1$, $\alpha_2$) ∈ *interacts-with* (*i.e.*, $\alpha_1$ *interacts-with* $\alpha_2$) if they interact with each other in one of the three ways:

**(1) Access the same device capability:** Apps $\alpha_1$ and $\alpha_2$ can access a shared device using the same capability; $\alpha_1$ *updates* the device state (*i.e.*, feature $r$ and value $v$) and $\alpha_2$ accesses (*i.e.*, updates or reads) the device state. We refer to this relationship as a *device* interaction. For example, $\alpha_1$ may turn on a switch based on the input of a light/illuminance sensor and $\alpha_2$ may turn off the same switch based on a motion sensor, both calling methods on the same device handler object.

**(2) Physical interaction:** We say that two apps have a *physical-medium* interaction if the output of $\alpha_1$ *physically* becomes an input for $\alpha_2$ and affects the execution of $\alpha_2$. For example, $\alpha_1$ activates a robot vacuum cleaner at a certain time during the day, and the robot's movement becomes the input to a motion sensor that is used by $\alpha_2$.

**(3) Access the same global variable:** Apps $\alpha_1$ and $\alpha_2$ can interact via the same global variable, whose value is stored on the cloud, *e.g.*, $\alpha_1$ updates the variable and $\alpha_2$ accesses it. This is referred to as a *global-variable* interaction. In this study, we focused on the `location.mode` variable because it is the only global variable in the SmartThings platform that allows for both *write* and *read* accesses. `location.mode` has three preconfigured values: *Home*, *Away*, and *Night*. An example scenario is that one app updates `location.mode` based on the input of the presence sensor while the second app reads it to determine whether a door should be locked/unlocked.

***Conflict Relation.*** Apps $\alpha_1$ and $\alpha_2$ *conflict* if they interact (in one of the ways discussed above) and the interaction may compromise the *correctness* of the apps or produce an *unintended outcome*. Although the notion of a conflict is somewhat vague, we found that Definitions 3.1 and 3.2 worked well most of the time in practice.

*Definition 3.1.* **Device/Global-Variable Conflict.** *Two apps $\alpha_1$ and $\alpha_2$ conflict iff there exists an execution $\mathcal{X}$ of $\alpha_1$ and $\alpha_2$ and two actions $\mathcal{A}_1$ and $\mathcal{A}_2$ that update the same feature $r$ or mode $\mu$ in $\mathcal{X}$ such that: (1) $\mathcal{A}_1$ and $\mathcal{A}_2$ are performed by different apps ($\alpha_1$ and $\alpha_2$), (2) $\mathcal{A}_1$ and $\mathcal{A}_2$ write different values ($v_1$ and $v_2$, or $\mu_1$ and $\mu_2$), (3) there is no such $\mathcal{A}_3$ that updates the same $r$ or $\mu$ and that the update is ordered between $\mathcal{A}_1$ and $\mathcal{A}_2$, and (4) $\mathcal{A}_2$ was not initiated by a direct user action.*

*Definition 3.2.* **Physical-Medium Conflict.** *Two apps $\alpha_1$ and $\alpha_2$ conflict iff one app performs an action that affects a physical medium* (e.g.*, motion) and the other app reads from a sensor that can sense that physical medium* (e.g.*, a motion sensor).*

## 3.2 Smart Home App Pairs

***Choice of Apps.*** We studied 198 official and 69 third-party smart home apps that we have collected from the SmartThings official Github [67] and other third-party repositories. While the statistics of app usages and installations are proprietary, all the apps that we used in this study can be obtained easily from the aforementioned repositories. Today, the SmartThings official Github [67] has an active user community—it has been forked into personal repositories more than 70,000 times. Any user can get and upload any app's source code to the SmartThings Marketplace via the SmartThings Groovy IDE [24]. Thus, users can install and use any app.

***App Pairing.*** These apps were initially developed to perform their specific functionality. There are no standardized guidelines either from SmartThings or from the community as to how to develop an app in a way so that it can safely interact with other apps.

**Table 1: Groups of apps for device-type pairing.**

| Group | Capability | Subgroup | App | |
|---|---|---|---|---|
| | | | # Apps | # Pairs |
| Switches | `switch` | General | 24 | 276 |
| | | Lights | 32 | 496 |
| | | AC/fan/heat | 3 | 3 |
| | | Vent | 3 | 3 |
| | | Camera | 2 | 1 |
| Locks | `lock` | | 21 | 210 |
| Thermostats | `thermostat` | | 19 | 171 |
| Lights | `colorControl` | Hue | 13 | 78 |
| | | Non-Hue | 11 | 55 |
| Dimmers | `switchLevel` | | 11 | 55 |
| Alarms | `alarm` | | 10 | 45 |
| Valves | `valve` | | 7 | 21 |
| Music Players | `musicPlayer` | | 5 | 10 |
| Relay | `relaySwitch` | | 5 | 10 |
| Speech Synthesizers | `speechSynthesis` | | 3 | 3 |
| Cameras | `imageCapture` | | 2 | 1 |
| | | **Total** | **171** | **1,438** |

Our process for manual examination was to independently examine the source code of each app pair by at least two of the authors. In the event that the two examiners disagreed about whether an app pair conflicted, they discussed their disagreement on the app-pair's classification and reached a consensus. There are 35,511 app pairs given the 267 apps we collected above. From this huge set of pairs, we identify 2,844 pairs of apps that potentially interact with each other. We next explain how we use the three interact-with conditions to identify these 2,844 pairs. We will then study how many of these 2,844 pairs contain conflicts in Sections 4–6.

***Device-Type Pairing.*** To identify apps that have device interactions, we first divide the 267 apps into groups based on what type of device an app aims to manage, as shown in Table 1. Clearly, if two apps do not access a common device, it is impossible for them to have *device* interaction.

Out of the 267 apps, we excluded 132 apps for three reasons. First, we excluded apps that take inputs from outside of the SmartThings platform. For instance, the IFTTT (If-This-Then-That) [21] app functions as a bridge between the SmartThings platform and IFTTT, a *third-party* platform. These apps typically wait for a third-party application built on a third-party platform (*e.g.*, IFTTT and other similar platforms) to send commands and generate events through HTTP endpoints. We do not have access to the source code of such third-party applications; thus, it is not possible to accurately reason about potential interactions. Second, we excluded apps that only send messages to a smartphone about the state of sensors because these apps do not interact with other apps. Third, we also excluded apps that use third-party specific device handlers since these apps cannot share a device with other apps. Therefore, we included 135 apps for *device* interaction. Some of them access multiple devices and, thus, are included in multiple groups of devices—hence, a total of 171 apps. At the end, we identified a total of 1,438 pairs from the 171 apps classified in various device-type-based groups.

For some groups, we identify all pairs of apps from the group as *device*-interaction pairs. For example, the *Locks* group contains 21 apps, we inspected all the $\binom{21}{2} = 210$ pairs and confirmed them all to be *device*-interaction pairs.

For some groups that provide *generic* functionality, such as Switches and Lights, we further create sub-groups and only identify apps that belong to the same sub-group as having a *device* interaction. For example, for the Switches group, out of a total of 64 apps, 24 access general switches (276 pairs), 32 access light switches (496 pairs), 3 access AC/fan/heater (3 pairs), 3 access the ventilation system (3 pairs), and 2 access cameras (1 pair). We also found 8 apps (not included in Table 1) that control specific devices (*e.g.*, curling-iron) that are not shared by other apps; hence, no pairs were constructed for these apps. The Lights group consists of apps that use the light device handler (*i.e.*, `capability.colorControl`) to turn the lights on or off, set their illuminance level [18], or change their colors. Each group was divided into a subgroup of apps that controls Philips Hue lights and another subgroup that controls non-Hue lights. In the Lights group, there are 13 apps for Hue leading to 78 pairs and 11 apps for non-Hue leading to 55 pairs.

**Table 2: Groups of apps for physical-medium pairing.**

| Output | # Apps | Sensor | # Apps | # Pairs |
|---|---|---|---|---|
| Lights | 42 | Illum. | 5 | 205 |
| Moving Dev. | 2 | Motion | 39 | 78 |
| Water Valves | 2 | Water | 11 | 21 |
| Sound Dev. | 21 | Sound | 1 | 21 |
| | | | **Total** | **325** |

***Physical-Medium Pairing.*** Two apps can interact via a *physical medium*; *e.g.*, one app generates an output that could be a *physical* input to the other app. To illustrate, consider an app that changes the state (*i.e.*, toggle on/off) of light bulbs. These changes also affect the illuminance produced by the light bulbs, which can become an input to apps that read from illuminance sensors.

Table 2 reports results for apps that interact physically. We grouped them based on the output-input relationships, such as lights (output) and illuminance sensors (input), moving devices (output) and motion sensors (input), water valves (output) and water sensors (input), or sound-generating devices (output) and sound sensors (input). For the light-illuminance-sensor relationship, for example, we constructed a total of 205 pairs for the 42 apps that control lights and the 5 apps that read from illuminance sensors.

***Global-Variable Pairing.*** Apps can also interact if they access the same global variable. Currently, there is only one global variable in the SmartThings platform that multiple apps can read from and write into: `location.mode`. We grouped together all the 47 apps that access it for a total of 1,081 pairs.

## 3.3 Threats to Validity

***External Validity.*** This study focused on Samsung's SmartThings platform and thus may miss interaction patterns specific to other platforms. However, we believe that most of the findings and insights revealed in this study are universal for smart home applications and frameworks. For instance, our results also apply to rule-based systems, *e.g.*, IFTTT—two rules: (1) "if the humidity is high, turn off the AC" and (2) "if the temperature is low, turn on the AC", have a conflict by our definition if the humidity is high and the temperature is low. Even for interactions that are specific to the SmartThings platform (*e.g.*, concurrent accesses to the `location.mode` variable), the patterns discovered under such interactions are general. For example, other platforms would also have global variables that serve similar purposes and hence our results can be generalized to these other platforms as well.

***Internal Validity.*** This study covers *all of the 198 official apps* that we could find in the SmartThings official Github repository and the example set for SmartThings tutorials as of July 2018. We added 69 third-party apps that we gathered from various other sources.

While we studied the *complete set* of the official apps, the third-party apps used in the study may not be exhaustive. Nevertheless, our experience shows that the patterns that exist in the official apps are similar to those in the third-party apps. We believe adding new third-party apps would not change the main findings and insights.

In this study, we limited the scope of app interactions to pairs, and hence, there could be new types of interactions that manifest only when three or more apps are involved. However, we have already manually inspected a large number of triplets and not found any new interaction patterns that do not exhibit in pairs.

We manually inspected app pairs to determine whether the two apps in each pair can conflict. The manual determination is subjective in some cases—it reflects the authors' beliefs of whether the interactions between a pair of apps represent an unintended outcome. For example, if one app turns a light on and a second app based on the absence of motion from a sensor turns the light off, we classify this as a conflict. However, users may compose apps with the intention of this app interaction. As another example, certain interactions are made over physical mediums; for instance, the sound generated by a speaker app could become the input of a sound sensor used by a different app. In this case, whether the sensor can pick up the sound depends on whether it is physically close to the speaker generating the sound. In the study, we assume that this interaction can actually happen although the speaker and the sensor may be far away in a real-life deployment.

Conflicts that have safety or security aspects are certainly critical and could be harmful. However, it is somewhat difficult to determine the potential safety hazards or implications of a conflict as they can depend on the specific deployment. For example, if a conflict causes a smart outlet to remain on, whether it is a safety hazard depends on what is plugged into the smart outlet, *e.g.*, toaster versus LED light. Nevertheless, even benign conflicts can render apps useless—they make a smart home system unpredictable and difficult to rely on with any confidence, ultimately causing users to get rid of the system.

Our ultimate goal is to identify all *avoidable* conflicts and their possible sources so that actions can be taken in future development and/or deployment to mitigate potential conflicts. Some conflicts can be potentially handled by the development of API with support for common app interaction patterns. On the contrary, if physical proximity is a concern, we could develop an analysis that warns the user during installation. This explains why we treated these two scenarios differently.

## 4 DEVICE INTERACTION

This section presents our findings for apps that form pairs with *device* interactions. When we first studied this category, we found that some apps monitor status changes but do not initiate any changes on devices. When such an app is paired with another device monitor app, both apps *concurrently read* the device status and neither of them makes any changes to the device status. We refer to such a pair of apps as having a *read-read* relationship. 128 (8.9%) pairs have this relationship and thus do not interact. We classified *device* interactions into *non-conflicting* and *conflicting* interactions; the statistics of the classification are reported in Table 3.

**Table 3: Statistics for *device* interaction.**

| Relationship | # Pairs | Percentage |
|---|---|---|
| Read-read | 128 | 8.9% |
| **Non-conflicting Interactions** | | |
| Direct-direct | 20 | 1.4% |
| Composable | 319 | 22.2% |
| Different-feature | 52 | 3.6% |
| Same-feature | 90 | 6.3% |
| | **481** | **33.5%** |
| **Conflicting Interactions** | | |
| Feature conflicts | 632 | 43.9% |
| Invalid-local-state | 76 | 5.3% |
| Dropped-update | 121 | 8.4% |
| | **829** | **57.6%** |
| **Total** | **1,438** | |

### 4.1 RQ1: Types of Non-Conflicting Interactions

We observed three types of non-conflicting interactions. First, although two apps can access the same device, their accesses can only be triggered *manually* by users. Consequently, whether they conflict with each other depends on how users operate them. For example, `Big-Turn-ON` is such an app: it turns on switches when the user touches the app's user interface [8]. Two users may concurrently initiate conflicting commands to a switch through two apps like `Big-Turn-ON`. We consider this type of conflicts out of the control of apps. We refer to this type of interaction as a *direct-direct* relationship. We found that this relationship holds for 20 (1.4%) app pairs in the *device* category (see Table 3).

Second, certain apps can work together to realize desired functionality, and hence are *intended* to interact with each other. We refer to this type of interaction as a *composable* relationship and corresponding apps as *composable* apps. We found that this composable relationship holds for 319 (22.2%) pairs in the *device* category.

Note that many of these composable apps were developed independently. For example, the `FireCO2Alarm` app sets off the alarm and triggers door-unlocks when smoke/fire is detected [63], while the `Initial-State-Event-Streamer` app [17] monitors and forwards events from many devices including the alarm device handler to a website [22] that allows users to remotely monitor device activities. These two apps were independently developed, but they could interact to fulfill a desired functionality at run time—notifying a user through the specific website that an alarm is set off.

Third, some apps simultaneously access different features of the same device or the same feature of the same device in a consistent way, and hence do not conflict with each other.

An example of the former (*i.e.*, accesses to different features) is the `Keep-Me-Cozy` and `Thermostats` [12, 20] pair of apps from the Thermostats group. One app calls methods on the thermostat to set heating or cooling points (*e.g.*, `setHeatingSetpoint()` and `setCoolingSetpoint()`), while the other app sets the mode of the thermostat (*e.g.*, via `setThermostatMode()`). Although these two apps control the same shared device, they operate on different features of the device. Hence, although the first app *interacts-with* the second app, there is *no conflict* between them. We refer to this interaction as a *different-feature* relationship and found this relationship holds for 52 (3.6%) pairs in the *device* category.

An example of the latter (*i.e.*, consistent accesses to the same feature) is the following pair of apps from the Locks group: the `Lock-It-at-a-Specific-Time` and `Auto-Lock-Door` apps [5, 66]. Both apps call `lock.lock()` to door-lock. We consider this interaction non-conflicting, since these apps' actions would lead the shared device to the *same state* and hence the expected outcome is not compromised. We refer to this interaction as a *same-feature* relationship and found it to hold for 90 (6.3%) pairs in the *device* category.

## 4.2 RQ2: Types of Conflicting Interactions

Of the 1,438 app pairs in the *device* category, 829 pairs exhibit conflicting behaviors. We classified these conflicting behaviors as either *feature conflicts* and *saved-state conflicts*.

***Feature Conflicts.*** There are many pairs where the two apps attempt to update the same device state with *incompatible values*. An example is the `FireCO2Alarm` and `Lock-It-When-I-Leave` pair discussed in Section 1. Recall that the `FireCO2Alarm` app attempts to door-unlock during a fire while the `Lock-It-When-I-Leave` app could potentially door-lock. We refer to these conflicts as *feature conflicts*. A majority of the app pairs: 632 (43.9%) pairs in the *device* category have feature conflicts.

***Saved-State Conflicts.*** Many apps use their local variables to keep track of device states and guide their own device updates. These apps easily become broken when paired with other apps that can update the same devices—a concurrent update from the other app would make this app's variable inconsistent with the device state.

Consider `Auto-Humidity-Vent` that turns on/off a fan based on the humidity level [2]. This app conflicts with the `Big-Turn-OFF` app that allows a user to manually turn off the fan [7] for the following reason. When `Auto-Humidity-Vent` detects that the room humidity is above a threshold, it turns on the fan and simultaneously updates its local state variable `state.fansOn` to `true`. A user may then use the `Big-Turn-OFF` app to turn off the fan, causing the room humidity to increase above the threshold. Unfortunately, since the local variable `state.fansOn` remains *true* in the `Auto-Humidity-Vent` app, unaware of the fan being turned off by `Big-Turn-OFF`, `Auto-Humidity-Vent` would stop functioning, incorrectly assuming that the fan is already on. We refer to this scenario as *invalid-local-state* conflicts, and found that 76 (5.3%) pairs in the *device* category exhibit this pattern.

A common pattern we observed is an app that stores and restores the state of a device. For example, the `Thermostat-Auto-Off` app restores the state of the thermostat to a previously stored state. Consider an execution in which after the `Thermostat-Auto-Off` app saves the current state (*e.g.*, `off`) of the thermostat into a local variable, a second app changes the actual device state to a different value (*e.g.*, `"cool"`), which does not propagate to `Thermostat-Auto-Off`'s internal state. The next time `Thermostat-Auto-Off` tries to restore the thermostat state, the restoration will be based on the stale and wrong value saved in the local variable. Thus the update performed by the second app is dropped. We refer to this scenario as *dropped-update* conflicts, and found 121 (8.4%) pairs in the *device* category exhibit this pattern.

## 4.3 RQ3: Prevalence of Conflicts

As reported in Table 3, 91.1% of the pairs in *device* category have actual interactions (*i.e.*, at least one device updates the device state), while 8.9% of the pairs have *read-read* relationships and hence do not actually interact. Of the pairs that have actual interactions, the majority (57.6%) have conflicts.

**Table 4: Statistics for *physical-medium* interaction.**

| Medium | # Pairs | Percentage |
|---|---|---|
| **Non-Conflicting Interactions** | | |
| Water | 10 | 3.1% |
| Sound | 21 | 6.4% |
| | **31** | **9.5%** |
| **Conflicting Interactions** | | |
| Water | 11 | 3.4% |
| Motion | 78 | 24.0% |
| Light state | 151 | 46.5% |
| Light color | 20 | 6.2% |
| Light brightness | 5 | 1.5% |
| Light combination | 29 | 8.9% |
| | **294** | **90.5%** |
| **Total** | **325** | |

## 4.4 RQ4: Unsafe Coding Patterns

We found there are at least two unsafe coding patterns for *device* interactions: (1) *blind-update* and (2) *saved-state*. The *blind-update* pattern occurs in apps that blindly update the same state of the same device without checking the current state of the device. The *saved-state* pattern occurs when an app that saves the state of a device feature into a local variable and later uses the saved value. This may cause updates from other apps to be discarded. In some cases, a check of the current state before doing the update could help the app verify that its local state is consistent with the device state. However, with the existing APIs, there is no way to do the check-and-update in an *atomic* way—an app could only retrieve the device state by invoking a method $m_1$ and then update the state by invoking another method $m_2$; the state could be changed by another app after $m_1$ returns but before $m_2$ is completed.

## 5 PHYSICAL-MEDIUM INTERACTION

This section presents our findings for apps that interact via the physical world. In this category, two apps are paired when the output from the first app can physically become the input of the second app and affect its operation. Table 4 reports our findings.

## 5.1 RQ1&2: Types of (Non-)Conflicting Interactions

***Motion.*** The first set of physical interactions are due to motion. An example pair is `Neato-(Connect)` and `Forgiving-Security` [1, 23]. `Neato-(Connect)` is a third-party app that controls a Neato vacuum-cleaning robot. When the app activates the robot, the robot starts cleaning the house. While it is moving around the house, its movement could trigger a motion sensor used by the `Forgiving-Security` app and thus set off a security alarm—a *false alarm*. Of the 325 app pairs in the *physical-medium* category, 78 pairs (24.0%) interact via motion and all exhibit conflicts.

***Light.*** A similar set of app pairs are based on interactions via light. The `Turn-On-at-Sunset` and `Light-Up-the-Night` apps [13, 16] are an example. Consider a deployment in which each app controls a different light bulb. At sunset, the `Turn-On-at-Sunset` app may turn on a light bulb whose light may affect the illuminance sensor of the `Light-Up-the-Night` app. The `Light-Up-the-Night` app is supposed to turn on a light bulb when its illuminance sensor detects that the surrounding is dark. If the light bulb controlled by the `Turn-On-at-Sunset` app is sufficiently close to the illuminance sensor used by the `Light-Up-the-Night` app, the sensor may pick up some light from the light bulb. This could cause the `Light-Up-the-Night` app to determine that there is no need to turn on the light bulb, and hence, the two apps conflict.

Some apps can control a light bulb by changing its on/off state, colors, or brightness levels. Any of these changes can potentially be detected by an illuminance sensor [18].

Table 4 summarizes our findings: 151 pairs (46.5%) have a conflict through the change of light's on/off state; 20 pairs (6.2%) conflict through the change of light's color; 5 pairs (1.5%) conflict through the change of light's brightness; and 29 pairs (8.9%) conflict through a combination of the three.

***Water.*** Physical interactions can also occur via water. An example pair that interacts via water consists of the `Sprayer-Controller-2` and `Close-The-Valve` apps [3, 6]. The former schedules irrigation for a certain amount of time periodically, while the latter closes a water valve when the water sensor detects moisture. When the water coming from a water sprayer controlled by the `Sprayer-Controller-2` app reaches the water sensor used by the `Close-The-Valve` app, the two apps interact. This interaction potentially results in a conflict because a bad moisture sensor placement could cause the `Close-The-Valve` app to prevent the irrigation that has been scheduled by the `Sprayer-Controller-2` app.

Our results show that 21 pairs interact through water: 11 of them have conflict and 10 do not. In each of these 10 pairs, the app that controls the water valve actually closes it when it detects moisture. Therefore, no water can be produced and detected by the water sensor of the other app.

***Sound.*** Apps can also interact via sound. For example, an interesting app pair is `Bose-SoundTouch-Control` and `InfluxDB-Logger`, which reads from a sound sensor [19]. In fact, the latter can be paired with any other sound-producing apps, such as those that control speakers, alarms, or music players.

Our findings show that there are 21 pairs (6.4%) that interact via sound but we could not find any conflicts among them. Typically, a pair consists of a sound-producing app and the `InfluxDB-Logger` app. Since the `InfluxDB-Logger` app only logs the status of the sound sensor, the two apps are actually *composable*—similar to the *composable* relationship in the *device* interaction (see Section 4.1).

***Physical Factors.*** The *physical-medium* interaction depends on certain physical factors. The position of the first app's actuator relative to the second app's sensor determines whether the output from the actuator could reach the sensor. If their proximity is sufficiently close for the actuator's output to affect the sensor, the two apps interact; otherwise, they do not. When we performed this study, we assumed that their locations are sufficiently close. Although it is a conservative approximation, this is the best we could do and our findings can help developers and users to avoid such conflicts.

## 5.2 RQ3&4: Prevalence of Conflicts/Unsafe Coding

Table 4 summarizes the statistics for the *physical-medium* interaction pairs. Our findings suggest that typically, when a pair of apps interact through a physical medium, they will most likely conflict. In most cases, the second app does not expect to receive any input from the first app. It normally expects sensor inputs from its surroundings. Out of the 325 pairs with *physical-medium* interaction, 90.5% (294 pairs) of them have a conflict. We did not observe any coding patterns that cause conflicts in this category. Hence, we concluded that the *conflict* in pairs with *physical-medium* interaction is caused mainly by the physical proximity between the actuators and sensors of the conflicting apps.

**Table 5: Statistics for *global-variable* interaction.**

| Relationship | # Pairs | Percentage |
|---|---|---|
| Read-read | 405 | 37.5% |
| **Non-Conflicting Interactions** | | |
| Direct-direct write-write | 28 | 2.6% |
| App write-read | 302 | 27.9% |
| Direct write-read | 221 | 20.4% |
| App-app write-write | 1 | 0.1% |
| | **552** | **51.0%** |
| **Conflicting Interactions** | | |
| App-app write-write | 44 | 4.1% |
| App-direct write-write | 80 | 7.4% |
| | **124** | **11.5%** |
| **Total** | **1,081** | |

## 6 GLOBAL-VARIABLE INTERACTION

This section presents our findings for app pairs that have *global-variable* interactions. As discussed in Section 3.1, since SmartThings only has one global variable `location.mode` that allows both reads and writes, we consider two apps to have *global-variable* interaction if they both access `location.mode`. Our statistics are reported in Table 5. 405 (37.5%) of the pairs (reported as pairs with *read-read* relationships in Table 5) contain apps that only read from `location.mode`. These apps do not *actually* interact.

### 6.1 RQ1: Types of Non-Conflicting Interactions

The first type contains apps that only write `location.mode` and they are controlled manually by the user. We refer to this as a *direct-direct write-write* relationship. As discussed earlier in Section 4.1, we did not consider these apps as conflicting since the user controls them. This group contains 28 pairs (2.6%), reported as pairs with *direct-direct write-write* relationships in Table 5.

A second type, consisting of 302 app pairs (27.9%), exhibits *app write-read* relationships, exemplified by the `Greetings-Earthling` and `Hello,-Home-Phrase-Director` apps [4, 11]. The `Greetings-Earthling` app changes the value of `location.mode` when the presence sensor detects that the homeowner arrives home. On the other hand, the `Hello,-Home-Phrase-Director` app sends a greeting message to the homeowner depending on the value of `location.mode`. In this case, the two apps have a *composable* relationship: one app reads the variable updated by the other.

A third type, consisting of 221 app pairs (20.4%), exhibits *direct write-read* relationships: one app requires the user to manually control the app to write into `location.mode`, while the other reads from it. This is the intended usage scenario of `location.mode`, namely to facilitate interactions between apps through mode changes. Hence, these *write-read* interactions are not conflicts.

Finally, we found one pair in which both apps write into `location.mode` and yet do not conflict. This pair consists of the `Greetings-Earthling` and `Bon-Voyage` apps [9, 11]. The `Greetings-Earthling` app writes into `location.mode` when the user arrives at home, while the `Bon-Voyage` app writes into the same location when the user leaves. Hence, they do not conflict as they have disjoint intents and never write at the same time. This is an exception to our current formal definition that can be improved.

### 6.2 RQ2: Types of Conflicting Interactions

When two apps both write into `location.mode`, in most cases, conflicts would result. There are two types of write-write conflicts: *app-app write-write* and *app-direct write-write*. For example, there exists an *app-app write-write* conflict between the `Smart-Security` and `Good-Night` apps [10, 15], which both attempt to write into `location.mode`. While the `Smart-Security` app updates `location.mode` with *Home*, the `Good-Night` app changes `location.mode` to *Night* or *Away*. In `Smart-Security`, the update to `location.mode` occurs when intrusion is detected. This is rather an important update and the user certainly does not want the result of the `Smart-Security` app to be compromised. There are 44 pairs (4.1%) of such conflicts.

An *app-direct write-write* conflict occurs when in one app the update of the global variable is triggered by a non-user input, *e.g.*, a sensor, while in the other app the user performs an operation that triggers the update. For example, the first app uses the motion sensor to detect if there is anyone home and updates `location.mode` based on the sensor input. The second app lets the user control the light—when the user turns on the light, `location.mode` is automatically updated. This category has 80 (7.4%) conflicting pairs.

### 6.3 RQ3&4: Prevalence of Conflicts and Unsafe Coding

There are a total of 124 (11.5%) conflicting app pairs. Thus, conflicts are *not* prevalent for this type of interaction.

We found that *concurrent-writes* to `location.mode` is an unsafe pattern, which is due to the SmartThings APIs that allow apps to directly change the value of `location.mode`. For instance, in the case of the `Smart-Security` app, a good practice would be to not allow other apps to write into `location.mode` when the alarm is sounding; otherwise, the alarm may be stopped abruptly before it is noticed. In the case of modes, the combination of (1) changing the API to specify a duration for the mode change and (2) allowing the user to specify priorities would resolve many of the conflicts.

## 7 DETECTING CONFLICTS

In this section we address **RQ5:** How can we automatically detect conflicts?

***Traditional Concurrency Detection.*** Traditional concurrency detects data races by checking whether the accesses to a certain location are ordered by a *happens-before* relation, which depends on the usage of locks and atomic instructions in the code. With a pair of smart home apps, however, the situation is different. For example, with the pair `FireCO2Alarm` and `Lock-It-When-I-Leave` apps in Section 1, the `FireCO2Alarm` app has an equal chance to door-unlock anytime before or after the `Lock-It-When-I-Leave` app door-locks. Thus, the notion of happens-before relation, which existing concurrency testing and verification tools would rely on, is absent in the interaction between these apps. As a result, we cannot use these existing tools to detect conflicts between the apps.

***IoTCheck.*** We developed IoTCheck, a tool that automatically identifies conflicts by *model-checking* pairs of apps. A model checker checks, exhaustively and automatically, if a system meets a specification. Model checking is particularly useful in detecting app conflicts due to its ability to exhaustively check all potential interactions between apps.

We begin by summarizing the key insights from our manual study that we used for designing IoTCheck. Our study shows that most device conflicts occur when two apps issue conflicting updates to the same device. We found that when one app writes to a device feature and another app reads from the same device feature, it typically does not represent a conflict; this scenario commonly occurs when apps compose. We also found that it is important to consider the reason why two apps perform conflicting updates. If both updates are performed in response to user requests, there is typically no conflict since the actions are triggered by the user. Finally, we found that conflicts on global variables occur only when two apps both write to the global variable; read-write interactions typically represent normal cooperation between apps, *not* conflicts. IoTCheck model-checks pairs of apps and monitors for conflicting updates to the same device or global variables from different apps. IoTCheck directly executes the original app code, eliminating the need to build models of the apps. IoTCheck extends the Java Pathfinder (JPF), an explicit state-based model checking infrastructure [76].

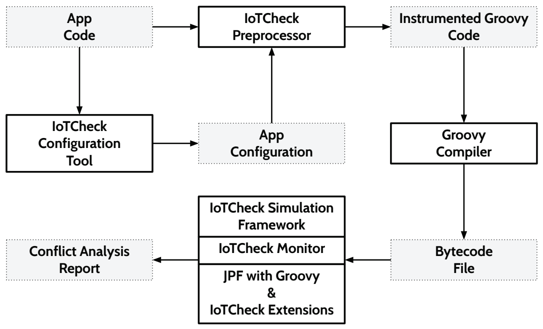

**Figure 3: IoTCheck Architecture.**

***Architecture.*** Figure 3 presents IoTCheck's architecture. The arrows represent the workflow of IoTCheck that starts from app code as an input to the IoTCheck configuration tool and IoTCheck preprocessor. Each SmartThings app has a configuration method that asks users for configuration information—this configuration is automatically generated by IoTCheck without human help. The IoTCheck configuration tool then outputs app configuration files, which, together with the original app, are processed by the IoTCheck preprocessor. The IoTCheck preprocessor generates model checker hooks to enable JPF to generate device events, combines multiple apps into the same program, and sets up the necessary configuration to run the program. It then outputs instrumented Groovy code which is compiled into bytecode by the Groovy compiler.

We developed a SmartThings simulation framework for IoTCheck. This framework contains virtualized devices (*i.e.*, device handlers) for all of the devices used by our benchmark apps. While an actual SmartThings device handler controls an actual device, a virtualized device handler changes the value of a state variable that represents the value of a device feature. Thus, a virtual device handler for a door lock changes the value of the door lock state variable instead of controlling an actual Zigbee door lock (see Figure 1). These device handlers are under the control of the JPF model checker—JPF triggers device events such as a motion detected by a motion sensor, or a temperature value change detected by a temperature sensor. For devices such as temperature sensors, there is a large range of potential temperatures that would make model checking infeasible without using symbolic techniques. IoTCheck thus supports a set of potential temperature readings (*e.g.*, a hot reading and a cold reading), which is practical given the nature of many smart home apps. IoTCheck does not currently model physical interactions between devices (other than to flag that they could potentially interact); this remains future work.

Finally, IoTCheck model-checks the generated bytecode using the JPF model checker. We developed IoTCheck monitor as a JPF listener that performs conflict analysis while JPF is executing the bytecode. When a conflict is detected, the listener halts JPF and immediately reports the conflict. Otherwise, JPF finishes its execution and the listener reports that there is no conflict.

***Challenges.*** There are 3 challenges in extending JPF for IoTCheck: **(1) JPF does not provide out-of-the-box support for checking Groovy code.** One challenge is that the Groovy runtime system keeps its own internal state that thwarts JPF's state matching algorithm; this often prevents even very simple Groovy programs from model-checking. IoTCheck extends JPF to consider only the state of the virtual smart home devices and the apps when matching states. This creates a second issue—JPF generates state matching points at many execution points. After eliminating Groovy runtime state from state matching, there can be spurious state matches terminating JPF before the state space is fully explored. To solve this problem, IoTCheck extends JPF to only match states right before generating a new event.
**(2) Groovy is a dynamic language.** Thus, method calls are resolved at runtime. The same call stack from the perspective of the program can be implemented by many different bytecode-level call stacks due to Groovy's method lookup and caching mechanisms. Since the call stack is considered by JPF's state matching algorithm, this can cause the algorithm to fail to match conceptually identical states and increase the state space to be explored. IoTCheck extends JPF's state matching algorithm to match conceptually identical call stacks with different bytecode-level stacks.
**(3) Scalability is a challenge for JPF as an explicit-state model checker.** IoTCheck initially exhaustively model-checks a app pair for up to 30 minutes. If it either a detects a conflict or completes, IoTCheck outputs the result and finishes. Otherwise, IoTCheck falls back on JPF's heuristic search and performs it for an extended 30-minute period. If no conflict is detected during this period or the tool runs out memory (usually caused by bigger apps that have tens of events), IoTCheck reports that the result is inconclusive. Future work can employ techniques such as partial order reduction to further improve IoTCheck's performance.

***Detection.*** Conflicts cannot be directly checked on the executions JPF explores because state-based model checking is only guaranteed to explore all program states and transitions and not all possible paths through the state machine. Consider apps $\alpha_1$ and $\alpha_2$ where $\alpha_1$ only turns the light on and $\alpha_2$ can turn the light on and off. A conflict only occurs when $\alpha_1$ turns the light on followed by $\alpha_2$ turning the light off. However, all states and transitions can be reached without exploring this execution path. Thus, we must analyze the state machine to determine whether it contains a conflicting path.

IoTCheck's conflict analysis is an online analysis of the state machine that JPF explores. Our analysis is similar to a standard dataflow compiler analysis with the exception that in our context

**Table 6: Comparison between manual study and IoTCheck.**

| Interaction | IoTCheck | Manual Study | |
|---|---|---|---|
| | | Conflict | No conflict |
| ***Device*** | **Conflict** | 679 | 38 |
| | **No conflict** | 33 | 101 |
| | **Not terminated** | 16 | 396 |
| | **Excluded** | 100 | 75 |
| ***Global-Variable*** | **Conflict** | 98 | 16 |
| | **No conflict** | 0 | 318 |
| | **Not terminated** | 0 | 388 |
| | **Excluded** | 26 | 235 |

nodes represent states and edges represent transitions. IoTCheck updates its analysis results as JPF explores new states and halts the exploration process when a conflict is detected. We abstract state machine as a set of nodes $n \in \mathcal{N}$ that represent the JPF states, and edges $e \in \mathcal{E}$ that represent transitions between JPF states. We denote sequences of actions using $A$. Each transition $e$ has a corresponding sequence of actions $A_e$. The relevant actions are write($\alpha$, $d$, $\tau$, $r$, $v$) and modewrite($\alpha$, $\mu$). We define in($n$) to be the set of incoming edges to $n$ and src($e$) to be the source node of the edge $e$. The analysis computes the set $S(n)$ of the most recent updates to each device feature and mode at node $n$. We define app($S, d, r$) to be the set of apps that have most recently updated $r$ on $d$ and value($S, d, r$) to be the value of that update. We define modeapp($S$) to return the set of apps that have most recently updated the mode and modevalue($S$) to return the values of the most recent update to the mode set.

Figure 4 presents equations that formalize our analysis. These equations are evaluated using a standard fixed point algorithm whenever JPF explores a new transition to either an existing state or a new state. Function $\phi$ applies the sequence of actions in transition to the set S for the previous node to compute the transition's contributions to set S for the destination node. The function update applies an action to set S.

***Results.*** We repeated the same set of evaluations, but using IoTCheck to check for conflicts instead of manual inspection. Table 6 compares IoTCheck's results with those from the manual study. We did not use IoTCheck to detect conflicts in *physical-medium* interactions since these conflicts depend on physical factors.

For the *device* interaction, we initially found 829 conflicting pairs through manual study: 632 pairs with *feature conflict*, 76 pairs with *invalid-local-state* conflicts, and 121 pairs with *dropped-update* conflicts (see Table 3). From the 829 pairs, we had to exclude 100 conflicting pairs because of the 8 apps that we could not run on IoTCheck: 5 apps use third-party features and 3 apps have serious bugs. Because of these 8 apps, we also had to exclude 75 non-conflicting pairs. Overall, IoTCheck was able to find conflicts in 679 pairs but failed to detect conflicts in 33 pairs—a thorough manual inspection confirmed that 8 pairs are indeed non-conflicting (*i.e.*, mistakes in our manual study), while other conflicts were not detected due to IoTCheck's limitations (*e.g.*, in our modeling of time). It also did not terminate for 16 pairs labeled as conflicting in the manual study, but 4 of them are indeed non-conflicting. Surprisingly, IoTCheck found 38 *new* conflicting pairs that were overlooked in our manual study and labeled as non-conflicting. Thus, in total IoTCheck found 717 conflicting pairs. For the 497 pairs labeled as non-conflicting in the manual study, IoTCheck confirms that 101 pairs are indeed non-conflicting, whereas it did not terminate for 396 of them.

$$S(n) = \bigcup_{\epsilon \in \text{in}(n)} \phi(A_e, \text{ismanual}(\epsilon), S(\text{in}(\epsilon))) \qquad \phi(\emptyset, \lambda, S) = S$$

$$\phi(A; \text{write}(\alpha, d, \tau, r, v), \lambda, S) = \begin{cases} \text{conflict}, & \textbf{if}(\exists a \in \text{app}(S, d, r).a \neq \alpha \land \text{value}(S, d, r) \neq v \land \neg\lambda) \\ \text{update}(\phi(A, \lambda, S), \mathit{write}(\alpha, d, \tau, r, v)) & \textbf{otherwise} \end{cases}$$

$$\phi(A; \text{modewrite}(\alpha, \mu), \lambda, S) = \begin{cases} \text{conflict}, & \textbf{if}(\exists a \in \text{modeapp}(S).a \neq \alpha \land \text{modevalue}(S) \neq \mu \land \neg\lambda) \\ \text{update}(\phi(A, \lambda, S), \text{modewrite}(\alpha, \mu)) & \textbf{otherwise} \end{cases}$$

$$\text{update}(S, \mathcal{A}) = \{\mathcal{A}' \in S \mid \neg\mathcal{A} \triangleq \mathcal{A}'\} \cup \{\mathcal{A}\} \qquad (\text{modewrite}(\alpha, \mu) \triangleq \text{write}(\alpha, d, \tau, r, v)) := \mathit{false}$$

$$(\text{write}(\alpha, d, \tau, r, v) \triangleq \text{write}(\alpha', d', \tau', r', v')) := (d = d') \land (r = r') \qquad (\text{modewrite}(\alpha, \mu) \triangleq \text{modewrite}(\alpha', \mu')) := \mathit{true}$$

**Figure 4: Conflict Analysis**

For the *global-variable* interaction, our manual study found 124 pairs of conflicting apps: 44 pairs with *app-app write-write* conflicts and 80 pairs with *app-direct write-write* conflicts (see Table 5). With IoTCheck, we were able to find conflicts in 98 of the 124 pairs. We had to exclude 26 of the pairs with conflicts because of 6 apps that we could not run on IoTCheck: 5 apps use third-party features and 1 app has serious bugs. Additionally, IoTCheck found 16 pairs with a conflict that was initially labeled as a non-conflicting pair. Because we excluded 6 apps, we had to exclude 235 non-conflicting pairs initially observed in the manual study. Among the 706 non-conflicting pairs labeled in the manual study, IoTCheck was able to complete its check and found no conflicts in 318 of them. IoTCheck did not terminate for 388 of them. For the *physical-medium* interaction, IoTCheck generates a warning if one app uses a device that could be the physical input of a device used by the other app.

***Statistics.*** The average runtime for IoTCheck to find conflicts is 27 seconds for the *device* interaction, and 11 seconds for the *global-variable* interaction. These suggest that conflicts are found quickly: the 30-minute time limit is enough to perform an exhaustive model checking in general. Thus, classifying non-terminating runs as non-conflict gives IoTCheck a precision of 100% and a specificity of 100%. The recall is 95.1% for the *device* interaction pairs and 100% for the *global-variable* interaction pairs—overall recall is 95.7%.

***Soundness.*** IoTCheck is sound in the sense that if it declares that two apps have a conflict, there is indeed an execution that has the conflict. Whether this conflict represents a problem in the real world is a very complicated question and can depend on (1) the intended use of the homeowner and (2) the home environment. The false positives/negatives in our manual study were typically due to subtle issues involving complex logic that had several conditions for generating commands or subtle concurrent executions—please see our tool and dataset releases for a full accounting [70–72].

## 8 RELATED WORK

The research community has recently looked into smart home apps [25, 30, 35, 36, 39, 40, 60, 69, 82]. Fernandes *et al.* present a thorough study on the SmartThings environment [39]. They pointed out underlying security issues and a simple program analysis to detect the *overprivilege* issue in the app source code. In [40], Fernandes *et al.* present a solution to prevent applications from leaking confidential information.

Researchers have presented new techniques to model-check and analyze confidential information leakage in smart home applications. The techniques presented in [59, 60] require translating the apps to perform the model checking using SPIN [45]. The limitation is that the expressiveness of app features could be lost in translation: with just 3 apps the authors found 1 feature that their system could not express concisely [59]. Other work [30, 35, 36] ignores internal application state, and thus admits executions that cannot happen. Several of our apps depend on internal state to decide whether to perform an action, and thus they would not be accurately modeled by their techniques. While conflicts between apps are discussed in [36], they considered a much smaller corpus of apps and a number of of them are self-crafted to generate the intended conflicts. Unfortunately, their system is not publicly available for comparison.

The interactions of smart home apps also appear similar to event-based races in mobile apps [31, 46, 47, 56, 65]. Related work on mobile apps deals with events only in one app by introducing various new synchronization mechanisms. However, our work focuses on the interactions between multiple apps. The event handlers in these apps are developed by different programmers with absolutely no coordination. In addition, a number of apps also allow the user to generate arbitrary events, *e.g.*, using a touch screen. Hence, even if the ordering between events in one app can be clearly defined, the ordering between events across multiple apps combined with user-generated events is complicated and arbitrary—synchronizations in individual apps would not be useful in this context.

There have also been efforts to resolve the conflicts between smart home apps from the perspective of dependencies between application components at the system level [57, 74, 75, 78, 79]. Several systems [74, 75, 78] provide frameworks for programming networks of sensors and actuators. DepSys [57] provides infrastructure with comprehensive strategies to specify, detect, and resolve conflicts through the use of user-specified metadata. Kripke [79] performs conflict detection through the use of model checking. Our work is orthogonal to this body of work that attempts to deal with conflicts between apps at the system level, by viewing apps as black boxes. Our work, on the contrary, studies how apps interact and what can be done at the source code level to mitigate conflicts.

## 9 CONCLUSION

This paper presents a comprehensive study of interactions and conflicts between smart home apps, as well as an automated tool for finding conflicts.

## ACKNOWLEDGMENTS

We thank our anonymous ESEC/FSE reviewers for their invaluable feedback. This project was partly supported by the National Science Foundation under grants CNS-1613023, CNS-1703598, CNS-1763172, CNS-2006437, CNS-2007737, CCF-1837120, CCF-2006948, OAC-1740210 and by the Office of Naval Research under grants N00014-16-1-2913 and N00014-18-1-2037.

## REFERENCES

[1] 2013. Forgiving Security. https://github.com/imbrianj/forgiving_security/blob/master/forgiving_security.groovy.

[2] 2014. Auto Humidity Vent. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/jonathan-a/auto-humidity-vent.src/auto-humidity-vent.groovy.

[3] 2014. Close The Valve. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/close-the-valve.src/close-the-valve.groovy.

[4] 2014. Hello, Home Phrase Director. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/tslagle13/hello-home-phrase-director.src/hello-home-phrase-director.groovy.

[5] 2014. Lock it at a specific time. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/user8798/lock-it-at-a-specific-time.src/lock-it-at-a-specific-time.groovy.

[6] 2014. Sprayer Controller 2. https://github.com/erocm123/SmartThingsPublic-1/blob/master/smartapps/sprayercontroller/sprayer-controller-2.src/sprayer-controller-2.groovy.

[7] 2015. Big Turn OFF. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/big-turn-off.src/big-turn-off.groovy.

[8] 2015. Big Turn ON. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/big-turn-on.src/big-turn-on.groovy.

[9] 2015. Bon Voyage. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/bon-voyage.src/bon-voyage.groovy.

[10] 2015. Good Night. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/good-night.src/good-night.groovy.

[11] 2015. Greetings Earthling. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/greetings-earthling.src/greetings-earthling.groovy.

[12] 2015. Keep Me Cozy. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/keep-me-cozy.src/keep-me-cozy.groovy.

[13] 2015. Light Up the Night. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/light-up-the-night.src/light-up-the-night.groovy.

[14] 2015. Lock It When I Leave. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/lock-it-when-i-leave.src/lock-it-when-i-leave.groovy.

[15] 2015. Smart Security. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/smartthings/smart-security.src/smart-security.groovy.

[16] 2015. Turn On at Sunset. https://github.com/SmartThingsCommunity/Code/blob/master/smartapps/sunrise-sunset/turn-on-at-sunset.groovy.

[17] 2016. Initial State Event Streamer. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/initialstate-events/initial-state-event-streamer.src/initial-state-event-streamer.groovy.

[18] 2016. Understanding Illuminance: What's in a Lux? https://www.allaboutcircuits.com/technical-articles/understanding-illuminance-whats-in-a-lux/.

[19] 2017. InfluxDB Logger. https://github.com/codersaur/SmartThings/blob/master/smartapps/influxdb-logger/influxdb-logger.groovy.

[20] 2017. Thermostats. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/thermostats.src/thermostats.groovy.

[21] 2018. IFTTT. https://www.ifttt.com/.

[22] 2018. Initial State. https://www.initialstate.com/.

[23] 2018. Neato (Connect). https://github.com/alyc100/SmartThingsPublic/blob/master/smartapps/alyc100/neato-connect.src/neato-connect.groovy.

[24] 2019. SmartThings Groovy IDE. https://graph.api.smartthings.com/.

[25] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1362–1380.

[26] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD)*. 1–8.

[27] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. 2010. Detecting Dependences and Interactions in Feature-Oriented Design. In *IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*. 161–170.

[28] Sven Apel, Alexander Von Rhein, Thomas ThüM, and Christian KäStner. 2013. Feature-interaction Detection Based on Feature-based Specifications. *Computer Networks: The International Journal of Computer and Telecommunications Networking* 57, 12 (August 2013), 2399–2409.

[29] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2016. Practical, formal synthesis and automatic enforcement of security policies for android. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 514–525.

[30] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2018. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *arXiv preprint arXiv:1809.06962* (2018).

[31] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 332–348. https://doi.org/10.1145/2814270.2814303

[32] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 71–85.

[33] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141.

[34] Nguyen Tan Cam, Pham Van Hau, and Tuan Nguyen. 2016. Android security analysis based on inter-application relationships. In *Information Science and Applications (ICISA) 2016*. Springer, 689–700.

[35] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. [n.d.]. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[36] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association.

[37] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting Data Races in Cilk Programs That Use Locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. ACM, New York, NY, USA, 298–309. https://doi.org/10.1145/277651.277696

[38] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[39] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 636–654.

[40] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 531–548. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/fernandes

[41] Cormac Flanagan and Stephen N Freund. 2004. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. ACM, New York, NY, USA, 256–267. https://doi.org/10.1145/964001.964023

[42] The Apache Software Foundation. 2003-2018. The Apache Groovy programming language. http://groovy-lang.org/.

[43] Google. 2018. Android Things website. https://developer.android.com/things/.

[44] Yi He, Qi Li, and Kun Sun. 2017. LinkFlow: Efficient Large-Scale Inter-app Privacy Leakage Detection. In *International Conference on Security and Privacy in Communication Systems*. Springer, 291–311.

[45] Gerard J Holzmann. [n.d.]. *The SPIN model checker: Primer and reference manual.* Vol. 1003.

[46] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/2594291.2594330

[47] Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-based Races in Android Apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 257–270. https://doi.org/10.1145/3173162.3173173

[48] Michael Jackson and Pamela Zave. 1998. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Transactions on Software Engineering* 24, 10 (October 1998), 831–847.

[49] Youn Kyu Lee, Jae Young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A sealant for inter-app security holes in android. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 312–323.

[50] Pattara Leelaprute, Takafumi Matsuo, Tatsuhiro Tsuchiya, and Tohru Kikuno. 2008. Detecting Feature Interactions in Home Appliance Networks. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*. 895–903.

[51] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security and Privacy Conference*. Springer, 513–527.

[52] Xinyi Li, Lei Zhang, and Xipeng Shen. 2019. IA-graph Based Inter-app Conflicts Detection in Open IoT Systems. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 135–147.

[53] Xinyi Li, Lei Zhang, Xipeng Shen, and Yong Qi. 2017. *A Systematic Examination of Inter-App Conflicts Detections in Open IoT Systems*. Technical Report TR-2017-1. North Carolina State University, Dept. of Computer Science.

[54] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic Race Detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 443–457. https://doi.org/10.1145/3009837.3009857

[55] Shan Lu, Soyeon Park, and Yuanyuan Zhou. 2012. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering* 38, 4 (2012), 844–860.

[56] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. *SIGPLAN Not.* 49, 6 (June 2014), 316–325. https://doi.org/10.1145/2666356.2594311

[57] Sirajum Munir and John A. Stankovic. 2014. DepSys: Dependency Aware Integration of Cyber-Physical Systems for Smart Homes. In *ICCPS '14: ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014)* (Berlin, Germany) *(ICCPS '14)*. IEEE Computer Society, Washington, DC, USA, 127–138. https://doi.org/10.1109/ICCPS.2014.6843717

[58] Chandrakana Nandi and Michael D. Ernst. 2016. Automatic Trigger Generation for Rule-based Smart Homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS)*. 97–102.

[59] Julie L Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. 2017. IOTA: a calculus for internet of things automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 119–133.

[60] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the Safety of Systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. ACM, New York, NY, USA, 191–203. https://doi.org/10.1145/3281411.3281440

[61] openHAB. 2018. openHAB website. https://www.openhab.org/.

[62] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. 2011. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Proceedings of the 15th International Software Product Line Conference (SPLC)*. Article 6, 6:1–6:8 pages.

[63] Yves Racine. 2014. FireCO2Alarm SmartApp. https://github.com/yracine/device-type.myecobee/blob/master/smartapps/FireCO2Alarm.src/FireCO2Alarm.groovy.

[64] Ajitha Rajan, Lydie du Bousquet, Yves Ledru, German Vega, and Jean-Luc Richier. 2010. Assertion-based Test Oracles for Home Automation Systems. In *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPRES)*. 45–52.

[65] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/2509136.2509538

[66] Chris Sader. 2013. Auto Lock Door SmartApp. https://github.com/smartthings-users/smartapp.auto-lock-door/blob/master/auto-lock-door.smartapp.groovy.

[67] SmartThings. 2018. SmartThings Public GitHub Repo. https://github.com/SmartThingsCommunity/SmartThingsPublic.

[68] Samsung SmartThings. 2018. Samsung SmartThings website. http://www.smartthings.com.

[69] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. 2017. Smartauth: User-centered Authorization for the Internet of Things. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, Berkeley, CA, USA, 361–378. http://dl.acm.org/citation.cfm?id=3241189.3241219

[70] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, and Guoqing Harry Xu. 2020. IoTCheck. http://plrg.ics.uci.edu/iotcheck/. https://doi.org/10.5281/zenodo.3866497

[71] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, and Guoqing Harry Xu. 2020. IoTCheck and manual study supporting materials. http://plrg.ics.uci.edu/iotcheck/. https://doi.org/10.5281/zenodo.3866499

[72] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, and Guoqing Harry Xu. 2020. IoTCheck Vagrant package. http://plrg.ics.uci.edu/iotcheck/. https://doi.org/10.5281/zenodo.3866491

[73] Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-An, Gregg Rothermel, and Jackson Dinh. 2017. An efficient, robust, and scalable approach for analyzing interacting android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 324–334.

[74] Pascal A Vicaire, Enamul Hoque, Zhiheng Xie, and John A Stankovic. 2012. Bundle: A group-based programming abstraction for cyber-physical systems. *IEEE Transactions on Industrial Informatics* 8, 2 (2012), 379–392.

[75] Pascal A Vicaire, Zhiheng Xie, Enamul Hoque, and John A Stankovic. 2010. Physicalnet: A generic framework for managing and programming across pervasive computing networks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 269–278.

[76] William Visser, Klaus Havelund, Guillaume Brat, SeungJuun Park, and Flavio Lerda. 2003. Model checking programs. 10 (April 2003), 203–232. Issue 2.

[77] Michael Wilson, Mario Kolberg, and Evan H. Magill. 2008. Considering side effects in service interactions in home automation-an online approach. *Feature Interactions in Software and Communication Systems IX* (2008), 172–187.

[78] Anthony D Wood, John A Stankovic, Gilles Virone, Leo Selavo, Zhimin He, Qiuhua Cao, Thao Doan, Yafeng Wu, Lei Fang, and Radu Stoleru. 2008. Context-aware wireless sensor networks for assisted living and residential monitoring. *IEEE network* 22, 4 (2008).

[79] Miki Yagita, Fuyuki Ishikawa, and Shinichi Honiden. 2015. An Application Conflict Detection and Resolution System for Smart Homes. In *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems* (Florence, Italy) *(SEsCPS '15)*. IEEE Press, Piscataway, NJ, USA, 33–39. http://dl.acm.org/citation.cfm?id=2821404.2821413

[80] Svetlana Yarosh and Pamela Zave. 2017. Locked or Not?: Mental Models of IoT Feature Interaction. In *Proceedings of the 2017 Conference on Human Factors in Computing Systems (CHI)*. 2993–2997.

[81] Adarsh Yoga and Santosh Nagarakatte. 2016. Atomicity Violation Checker for Task Parallel Programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. ACM, New York, NY, USA, 239–249. https://doi.org/10.1145/2854038.2854063

[82] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. HoMonit: Monitoring Smart Home Apps from Encrypted Traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1074–1088. https://doi.org/10.1145/3243734.3243820