IotSan: Fortifying the Safety of IoT Systems

Dang Tu Nguyen*, Chengyu Song*, Zhiyun Qian*, Srikanth V. Krishnamurthy*,

Edward J. M. Colbert†, and Patrick McDaniel‡

*UC Riverside, †U.S. Army Research Laboratory, ‡The Pennsylvania State University

{tnguy208, csong, zhiyunq, krish}@cs.ucr.edu, edward.j.colbert2.civ@mail.mil, mcdaniel@cse.psu.edu

ABSTRACT

Today's IoT systems include event-driven smart applications (apps) that interact with sensors and actuators. A problem specific to IoT systems is that buggy apps, unforeseen bad app interactions, or device/communication failures, can cause unsafe and dangerous physical states. Detecting flaws that lead to such states, requires a holistic view of installed apps, component devices, their configurations, and more importantly, how they interact. In this paper, we design IotSan, a novel practical system that uses model checking as a building block to reveal "interaction-level" flaws by identifying events that can lead the system to unsafe states. In building IotSan, we design novel techniques tailored to IoT systems, to alleviate the state explosion associated with model checking. IotSan also automatically translates IoT apps into a format amenable to model checking. Finally, to understand the root cause of a detected vulnerability, we design an attribution mechanism to identify problematic and potentially malicious apps. We evaluate IotSan on the Samsung SmartThings platform. From 76 manually configured systems, IotSan detects 147 vulnerabilities. We also evaluate IotSan with malicious SmartThings apps from a previous effort. IotSan detects the potential safety violations and also effectively attributes these apps as malicious.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*;

ACM Reference Format:

Dang Tu Nguyen*, Chengyu Song*, Zhiyun Qian*, Srikanth V. Krishnamurthy*, Edward J. M. Colbert†, and Patrick McDaniel‡ *UC Riverside, †U.S. Army Research Laboratory, ‡The Pennsylvania State University {tnguy208, csong, zhiyunq, krish}@cs.ucr.edu, edward.j.colbert2.civ@mail.mil, mcdaniel@cse.psu.edu . 2018. IotSan: Fortifying the Safety of IoT Systems. In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281440

1 INTRODUCTION

A variety of IoT (Internet-of-Things) systems are already widely available on the market. These systems are typically controlled by *event-driven* smart apps that take as input either sensed data, user inputs, or other external triggers (from the Internet) and command one or more actuators towards providing different forms of automation. Examples of sensors include smoke detectors, motion sensors, and contact sensors. Examples of actuators include smart locks, smart power outlets, and door controls. Popular control platforms on which third-party developers can build smart apps that interact wirelessly with these sensors and actuators include Samsung's SmartThings [73], Apple's HomeKit [4], and Amazon's Alexa [3], among others.

While conceivably, IoT is here to stay, current research studies on security/safety of IoT systems are limited in two fronts. First, they focus on *individual components* of IoT systems: there are papers on the security of communication protocols [24, 31, 40, 57, 70], firmware of devices [1, 15, 21, 72, 81, 90], platforms [28, 51], and smart apps [27, 28, 51, 83]. Very few efforts have taken a holistic perspective of *an IoT system*. Second, most current research efforts only focus on securing the cyberspace, and do not address the safety and security of the physical space, which is one of the key obstacles for real-world IoT deployment [60].

Our thesis is that a holistic view of an IoT system is important *i.e.*, the distributed sensors and actuators, and the apps that interact with them need to be considered jointly. While the compromise of an individual component may lead to the compromise of the whole system, certain complex security and safety issues are only revealed when the interactions between components (*e.g.*, a plurality of poorly designed apps) and/or possible device/communication failures are considered. These latent problems are very real since apps are often developed by third-party vendors without coordination, and are likely to be installed by one or more users (*e.g.*, family members) at different times. Moreover, both legitimate device failures [29, 33, 86, 88] (*e.g.*, from battery depletion) and induced communication failures (*e.g.*, via jamming [69]) can lead to missed interactions between autonomous components, which in turn can cause the entire system to transition into a bad state. These issues are especially dangerous, because bad or missed interactions can be deliberately induced by attackers via spoofing sensors [76, 79], luring users to install malicious apps [51], or jamming sensor reports.

Goals: In this paper, our goal is to build a holistic system which, given an IoT system and a set of default plus user-defined safety properties with regards to both the cyber and physical spaces, (a) finds if components of an IoT system or interactions between components can lead to bad states that violate these properties; and, (b) attributes the detected violations to either benign misconfigurations or potential malicious apps. With regards to (a) we account for cases wherein app interactions or failed device(s)/communications

CoNEXT '18, December 4–7, 2018, Heraklion, Greece
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281440

can cause a bad state. With regards to (b) we look for repeated instantiations of unsafe states since malicious apps are likely to consistently try to coerce the IoT system into exploitable bad states (*e.g.*, those described in [51]).

To achieve our goal, we need to solve a set of technical challenges. Among these, the key challenge lies in the scope of the analysis: as the number of IoT devices and apps is already large and is only likely to grow in the future [35, 49], physical replication and testing of IoT systems is hard (due to scale). Thus, it is desirable to build a realistic model of the system, which captures the interactions between sensors, apps, and actuators.

Our solution: We achieve our goal by addressing the above and other practical challenges, in a novel framework IoTSan (for IoT Sanitizer). In brief, IoTSan uses model checking as a basic building block. Towards alleviating the state space explosion problem associated with model checking [18], we design two optimizations within IoTSan to (i) only consider apps that interact with each other, and (ii) eliminate unnecessary interleaving that is unlikely to yield useful assessment of unsafe behaviors. We also design an attribution module which flags potentially malicious apps, and attributes other unsafe states to bad design or misconfiguration.

We develop a prototype of IoTSan based on the Spin model checker [43] and apply it to the Samsung SmartThings platform. As one contribution, we design an automated model generator that translates apps written in Groovy (the programming language of SmartThings apps) into Promela, the modeling language of Spin. To evaluate IoTSan, we postulate 45 common sense safety properties and consider 150 smart apps with 76 configurations. With this setup, IoTSan discovered 147 violations of 20 safety properties due to app interactions (135 violations) and device/communication failures (12 violations). In an extreme case, 4 smart apps needed to interact to cause a violation, which is extremely difficult to spot manually. We evaluate our attribution module with 9 malicious apps from [51] that are relevant to our problem scope (*e.g.*, causing bad physical states). IoTSan attributes all 9 apps to be potentially malicious.

A summary of our contributions is as follows:

- We map the problem of detecting potential safety issues of an IoT system into a model checking problem. We develop novel pre-processing methods to alleviate the state explosion problem in model checking.
- We design IoTSan to detect safety violations in IoT systems and develop a prototype that applies to the Samsung SmartThings platform. We provide the source code of IoTSan for download at https://github.com/dangtunguyen/IoTSan [1]. We develop tools to automatically translate the app source code into Promela. We evaluate IoTSan with 150 smart apps from the SmartThings' market place and discover 147 possible safety violations.
- We propose a method to attribute safety violations to either bad apps or misconfigurations. The method attributes 9 known malicious apps with 100% accuracy.

2 BACKGROUND AND SYNOPSIS

Today's IoT systems [3, 4, 48, 56, 63, 73, 85] typically consist of three major components viz., (i) a hub and the IoT devices it controls, (ii) a platform (can be the hub, a cloud backend, or a combination) where smart apps execute, and (iii) a companion mobile app and/or a web-based app to configure and control the system. Without loss of generality, we design IoTSan assuming this underlying architecture. Therefore, although the implementation of IoTSan is tailored to the SmartThings platform given its recent popularity, [13, 14, 27, 28, 51, 83], conceptually IoTSan is also applicable to other IoT platforms. We use the term "IoT system" to refer to those used in smart homes as in recent papers such as [13, 14, 27, 28, 51, 83] for ease of exposition; however, our approach can apply to other application scenarios (*e.g.*, IoT based enterprise deployments or manufacturing systems [23, 45, 58, 64]).

[1]A more detailed technical report is also available at this site.

2.1 Samsung SmartThings

Overview: Like the other systems mentioned above, SmartThings has an associated hub and a companion mobile app, that communicate with a cloud backend via the Internet, using the SSL protocol [6]. Developers can create smart apps using the Groovy programming language. The platform and apps interact with devices through *device handlers*; written in Groovy, these are virtual representations of physical devices that expose the devices' capabilities. To publish a device handler, a developer needs to get a certificate from Samsung. Typically, smart apps and device handlers are executed in the SmartThings cloud backend inside sandboxes.

Programming model: A smart app subscribes to events generated by device handlers (*e.g.*, motion detected) and/or controls some actuators using method calls (*e.g.*, turn on a bulb). Smart apps can also send SMS and make network calls using the SmartThings' APIs. A smart app can discover and connect to devices, in two ways. Typically, at installation time, the companion app shows a list of supported devices to a user; after configuration, the list of the user's chosen devices are returned to the app. The second (lesser-known) way is that SmartThings provides APIs that allow apps to query all the devices connected to the hub. Besides subscribing to device events, smart apps can also register callbacks for events from external services (*e.g.*, IFTTT [46]) and timers.

Communications: The hub communicates with IoT devices using a protocol such as ZWave or ZigBee. Experiments using the EZSync CC2531 Evaluation Module USB Dongle [47] of Texas Instruments, reveal that the ZigBee implementation in SmartThings supports four (single hop) MAC layer retransmissions. In addition, SmartThings has an application support sublayer that performs 15 end-to-end retransmissions (for a total of 60 retransmissions of a packet). These are in line with ZigBee specifications as also verified in [2, 8, 53, 59]. Thus, typically, it is rare that the system will transition to unsafe states because of benign packet losses.

2.2 Misconfiguration Problems

Besides malicious apps, misconfiguration is a common cause for safety violations. When installing a smart app, a user has to configure the app with sensor(s) and actuator(s). Poor configurations can transition the IoT system to unsafe physical states. There are many common causes for such misconfigurations, *e.g.*, (i) the app's description is unclear, (ii) there are too many configuration options, and (iii) normal users often do not have good domain knowledge to clearly understand the behaviors of smart devices and smart

```
1 preferences {
2   section("Choose a temperature sensor... "){
3     input "sensor", "capability.temperatureMeasurement", title:
        "Sensor"
4   }
5   section("Select the heater or air conditioner outlet(s)... "){
6     input "outlets", "capability.switch", title: "Outlets",
        multiple: true
7   }
8   section("Set the desired temperature..."){
9     input "setpoint", "decimal", title: "Set Temp"
10  }
11  section("When there's been movement from (optional)"){
12    input "motion", "capability.motionSensor", title: "Motion",
        required: false
13  }
14  section("Within this number of minutes..."){
15    input "minutes", "number", title: "Minutes", required: false
16  }
17  section("But never go below (or above if A/C) this value with
        or without motion..."){
18    input "emergencySetpoint", "decimal", title: "Emer Temp",
        required: false
19  }
20  section("Select 'heat' for a heater and 'cool' for an air
        conditioner..."){
21    input "mode", "enum", title: "Heating or cooling?", options:
         ["heat","cool"]
22  }
23 }
```

Figure 1: Example of input info needed from users to configure the app *Virtual Thermostat.*

apps. To exemplify these issues, we conduct a user study (more details in §10) where we asked 7 student volunteers to configure various apps as they deemed fit. Among these apps, one app is called *Virtual Thermostat* and describes itself as "Control a space heater or window air conditioner (AC) in conjunction with any temperature sensor, like a SmartSense Multi." Figure 1 shows the inputs needed from a user, which include a temperature measurement sensor (lines 2-4), the power outlets into which the heater or the AC are plugged (lines 5-7), a desired temperature (lines 8-10), etc. Although the developers use the word *or* and the app only expects either a heater or an AC, 5 out of 7 student volunteers thought the app controls *both* a heater and an AC to maintain the desired temperature and mis-configured the app to control both the AC outlet and the heater outlet. To exacerbate the confusion, the app expects the configuration of outlets (`capability.switch`) instead of the actual devices that are plugged into the outlets (*i.e.*, AC or heater) (note that the SmartThings UI displays all available outlets to the user). As a result of volunteer misconfigurations, when the temperature is higher than a predefined threshold, the *Virtual Thermostat* would turn on both the configured outlets (*i.e.*, both the heater and the AC). This violates the following two commonsense properties: (i) a heater is turned on when temperature is above a predefined threshold and (ii) an AC and a heater are both turned on.

2.3 Model Checking as a Building Block

The problem of reasoning if and why the IoT system could transition into a bad physical state is challenging because the number of apps and devices is likely to grow in the future and thus, analyzing all

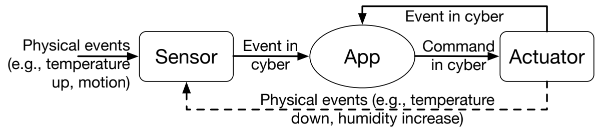


Figure 2: Chain of events in an IoT system.

possible interactions between them will be hard. Static analysis tools tend to sacrifice completeness for soundness, and thus result in lots of false positives. In contrast, typical dynamic analyses tools verify the properties of a program during execution, but can lead to false negatives.

Model checking is a technique that checks whether a system meets a given specification [50], by systematically exploring the program's state. In an ideal case, the model checker exhaustively examines all possible system states to verify if there is any violation of specifications relating to safety and/or liveness properties. However, the complexity of modern system software makes this extremely challenging computationally. So in practice, when the goal is to find bugs, a model checker is usually used as a *falsifier i.e.*, it explores a portion of the reachable state space and tries to find a computation that violates the specified property. This is sometimes also called bounded model checking [10, 20, 26, 52, 61].

We adopt model checking as a basic building block since: (i) it provides the flexibility towards verifying all the desired properties with linear temporal logic[2], (ii) it provides concrete counterexamples [5, 80] which are very useful in analyzing why and how the bad states occur, (iii) its holistic nature of checking can capture interactions among multiple apps, and (iv) it is more efficient than exhaustive testing [9]. However, a successful model checker must address the state explosion problem, *i.e.*, the state space could become unwieldy and requires exponential time to explore.

Given its popularity and flexibility in modelling both concurrent and synchronous systems [17, 25, 54], we use Spin [43] for checking if a given set of safety properties can be possibly violated. Since an IoT system may be composed of a large number of apps and smart devices, we use Spin's verification mode with BITSTATE hashing—an approximate technique that stores the hash code of states in a bitfield instead of storing the whole states. Although the BITSTATE hashing technique does not provide a complete verification, empirical results and theoretical analysis have proved its effectiveness in terms of state coverage [12, 16, 41, 42, 44].

3 SCOPE AND THREAT MODEL

In this work, our goal is to detect safety issues (*i.e.*, vulnerabilities) of IoT systems that are exploitable by attackers to transition the system into bad physical states or leak sensitive information. Safety requirements (*i.e.*, definition of bad states and information leakage) can come from both the users and security experts. Examples of bad physical states are (i) the front door is unlocked when no one is at home, and (ii) a heater is turned off when the temperature is below a predefined threshold. With regards to information leakage we require that: (i) private information is sent out via only

[2] Linear temporal logic (LTL) is a modal temporal logic with modalities referring to time. LTL is used to verify properties of reactive systems [5].

message interfaces (*e.g.*, *sendSmsMessage* and *sendPushMessage* in SmartThings) but not via network interfaces (*e.g.*, *httpPost* in SmartThings), and (ii) the recipients of methods for sending messages match the configured phone numbers or contacts. We point out that legitimate apps might use network interfaces to send some control information (*e.g.*, relating to crashes) back to the server. In such cases, we assume that users dictate whether to allow/disallow such operations (based on their privacy preferences).

We consider all devices (hub, sensors, and actuators), the cloud, and the companion app as our trusted computing base (TCB), and do not consider software attacks against them. However, IoTSan does mitigate physical attacks that can inject event(s) into the system (*e.g.*, by physically increasing the temperature or spoofing the sensors) or maliciously induced device or communication failures (*e.g.*, by jamming [69]). IoTSan seeks to identify and prevent such events from leading the system into safety violations. However, targeted solutions to those attacks (*e.g.*, preventing spoofing of sensors or jamming mitigation) are out-of-scope.

We also consider potential bad states that can arise due to natural device failures. Note that many users have reported the failures of their ZigBee and Z-Wave IoT devices (*e.g.*, motion sensors, water leak sensors, presence sensors, and garage door openers) in the SmartThings Community [29, 33, 86, 88]. Failures could also result from device batteries running out. We seek to identify if such device failures can cause an IoT system to transition into a bad physical state.

Malicious apps can exploit weaknesses in the configuration and attack other apps by introducing problematic events. We only seek to attribute an app as possibly malicious and leave the confirmation to human experts or other systems.

4 SYSTEM OVERVIEW

Figure 2 illustrates a high level view of the chain of events in an IoT system. In brief, sensors sense the physical world and convert them into events in the cyber world; these events, in turn, are passed onto apps that subscribe to such events. Upon processing the cyber events these apps may output commands to actuators, which then trigger new physical or cyber events. Apps may also directly generate new cyber events. Therefore, a single event could lead to a large sequence of subsequent cyber/physical events.

Figure 3 depicts the architecture of our system IoTSan. It consists of five modules viz., *App Dependency Analyzer*, *Translator*, *Configuration Extractor*, *Model Generator*, and *Output Analyzer*. In designing IoTSan, we tackle two main challenges: (i) alleviating the state space explosion with model checking [18] for our context, and (ii) the translation of smart apps' source code to Promela (to facilitate model checking). We address the first problem partially in the *App Dependency Analyzer* and partially in the *Model Generator*. The second problem is handled partially in the *Translator* and partially in the *Model Generator*.

App Dependency Analyzer (§ 5): This module constructs dependency graphs to capture interactions between event handlers of different apps and identifies handlers that must be jointly analyzed by the model checker. This precludes the unnecessary analysis of unrelated event handlers.

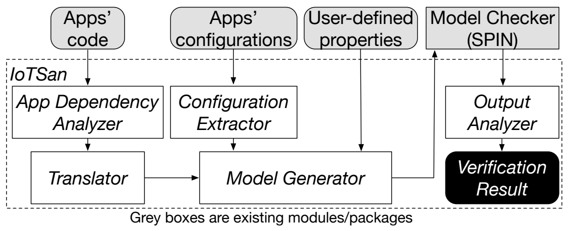


Figure 3: IoTSan architecture overview.

Table 1: Comparison of IoTSan and related work.

Feature	SIFT [55]	DeLorean [22]	Soteria [14]	IoTSan
Detects physical safety violations	✓	✓	✓	✓
Detects information leakage				✓
Detects violations due to communication/device failures				✓
Detects violations due to misconfiguration problems				✓
Handles complex code beyond IFTTT rules		✓	✓	✓
Performs violation attribution				✓
Accounts for app interactions	✓		✓	✓

Translator (§ 6): We build a translator within IoTSan, that automatically converts Groovy programs into Promela. In doing so, we address the following challenges:

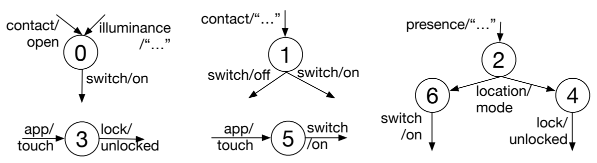
- *Implicit Types.* In Groovy programs, data types of variables and return types of functions are not explicitly declared. To solve this problem, we design an algorithm to infer data types of variables and return types of functions.
- *Built-in Utilities.* Groovy has many built-in utilities, *e.g.*, `find`, `findAll`, `each`, `collect`, `first`, `+` on list types, and `map`. We manually analyzed the behavior of each utility and translated them into corresponding code in Promela.

Configuration Extractor (§ 7): IoT platforms often provide a companion mobile app and/or web-based app to manage/configure the installed smart apps and devices of an IoT system. This module automatically extracts the system's configurations from the manager app.

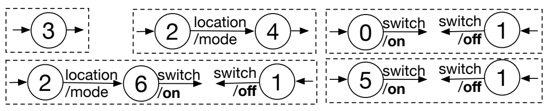
Model Generator (§ 8): This module takes the Promela code of event handlers, the configuration of the IoT system, and safety properties (both pre-defined and user-defined) as inputs, and creates the Promela model of the system. We use sequential design to model the IoT system instead of concurrent design. This significantly reduces the problem size by eliminating unnecessary interleaving that is unlikely to yield useful assessment of unsafe behaviors. The generated model is checked by Spin for possible property violations.

Output Analyzer (§ 9): This module analyzes the verification logs and attributes safety violations to potentially malicious apps, bad designs or misconfiguration. Based on the result, it provides the user, a suggestion to either remove a bad app(s) or change an app(s)'s configuration.

Our work in perspective: IoTSan can be envisioned as a service that jointly considers the apps, devices and their configurations of an IoT system, and checks whether a set of a priori defined properties hold. In addition to detecting safety violations of the physical



(a) Dependency graph.



(b) Related sets (each box represents a related set).

Figure 4: Example of a dependency graph and its corresponding related sets.

space, it also detects information leakage. Finally, it also determines if communication/device failures can cause unsafe states and detects violations due to misconfiguration problems. In Table 1 we show the features that IotSan offers compared to the most related recent systems. A discussion of related work is deferred to § 12.

5 APP DEPENDENCY ANALYZER

The model checker should not have to check the interactions between event handlers that do not interact. To find event handlers that can interact and thus jointly influence actuator actions, this module constructs a *dependency graph* (DG).

Extracting input/output events: Each smart app registers one or more *event handlers* that get notified of events to which it has subscribed. An event handler takes one or more input events, and can induce zero or more output events. Input events are (i) explicitly declared in the `subscribe` commands or, (ii) identified via APIs that read states of smart devices, or (iii) indicated by interrupts at specific times defined by `schedule` method calls. Output events are invoked via APIs that change states of smart devices. We enumerate the input and output events of an app using static analysis (details are straightforward and are omitted to save space).

Dependency Graph Construction: Once the input and output events are identified, we construct a directed DG as follows. Each event handler is denoted by a vertex in the DG. An edge from a vertex u to a vertex v ($u \rightarrow v$) is added if the output events of u overlap with the input events of v. u is then called the *parent* vertex of the *child* vertex v. The vertices in a strongly connected component are merged into a composite vertex (a union of input and output events). A *leaf* vertex does not have any child.

Example: To illustrate, consider the following example. Table 2 summarizes the event handlers and the associated input/output events with a set of sample smart apps. The description of an event is in the format *attribute*/*event type* (*e.g.*, contact/open means "a contact sensor is open"); empty quotes ("...") denote "any" event of that type. Given these apps, we show the DG that is built in Figure 4a. For each vertex, the incoming arrows denote input events and the outgoing arrows denote output events. For example, vertex 2 has two children viz., vertex 4 and vertex 6; all vertices except vertex 2 are leaf vertices.

Related sets: The initial *related set* of a leaf vertex $v \in$ DG includes all of its ancestors and v itself. There is no need to find such related sets for vertices that are not leaves, since those sets are subsets of other leaves' related sets. Table 3a shows the initial related sets in the DG from Figure 4a.

The initial related sets constructed as above are incomplete. This is because, two vertices u and v may have common output events but the types of these events could be different or what we call *conflicting*. For example, nodes 0 and 1 have conflicting output events viz., switch/off and switch/on. In such cases, the related sets to which u and v belong, must be merged to account for such conflicts. Table 3b shows the related sets of vertices with potential output conflicts in our example. Note here that to check for such output conflicts, we need to examine $O(E^2)$ links in the worst case (given E output edges from the event handlers); our experiments show that such checks are very fast.

We point out that if a related set i is a subset of a bigger related set j, the model checker automatically verifies i when j is verified; thus, there is no need to re-verify i. In Table 3c and Figure 4b, we show the final related sets associated with the DG in Figure 4a after removing all redundant subsets. These related sets are jointly analyzed by the model checker.

6 TRANSLATOR

Given its popularity and ease of use [32, 34, 75, 82], we build IotSan using the Bandera Tool Set [37, 38], which is a collection of program analysis, transformation, and visualization components designed to apply model-checking to verify Java source code. Bandera generates a program model and specification in the language of one of several existing model-checking tools (including Spin, dSpin, SMV, JPF). When a model-checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java lock objects [37, 38].

Since Bandera does not handle Groovy code, in order to analyze smart apps for SmartThings, we need to convert their code into Java which is challenging for the following reasons. First, since SmartThings added several language features to Groovy to simplify smart app development, the standard Groovy compiler cannot directly process an app's code and SmartThings's compiler is not open sourced. Second, Groovy uses dynamic typing [36] (*i.e.*, data types are checked at run-time) but Java is static typed (*i.e.*, data types are explicitly declared and checked at compile-time). Thus, we need to perform type inference during the translation of Groovy into Java. Lastly, Groovy supports many built-in utilities such as list and map, not supported by Bandera (*i.e.*, Bandera supports only Java's *array* type).

The key component we develop is the G2J Translator (see Figure 5), which translates the smart app Groovy source into Java's Abstract Syntax Trees (ASTs). In addition, the *SmartThings Handler* is designed to handle the new language syntaxes introduced by SmartThings, and the *GParser* parses the regular Groovy source code into Groovy ASTs. Basically, each smart app in Groovy is translated into a Java class, whose method comprises of a method's header and a block of statements. The translation procedure of a block is straightforward: iterate through the statement list of the

Table 2: An example to showcase the construction of a dependency graph.

App's Name	Event Handler	Vertex's ID	Input Events	Output Events
Brighten Dark Places	contactOpenHandler	0	contact/open, illuminance/"..."	switch/on
Let There Be Dark!	contactHandler	1	contact/"..."	switch/on, switch/off
Auto Mode Change	presenceHandler	2	presence/"..."	location/mode
Unlock Door	appTouch	3	app/touch	lock/unlocked
	changedLocationMode	4	location/mode	lock/unlocked
Big Turn On	appTouch	5	app/touch	switch/on
	changedLocationMode	6	location/mode	switch/on

Table 3: Related sets of the dependency graph in Figure 4a: (a) Initial related sets, (b) Potential conflicting sets, and (c) Final related sets.

(a)

Set	Vertexes
1	0
2	1
3	3
4	5
5	2, 4
6	2, 6

(b)

Set	Vertexes
1	0, 1
2	1, 5
3	1, 2, 6

(c)

Set	Vertexes
1	3
2	2, 4
3	0, 1
4	1, 5
5	1, 2, 6

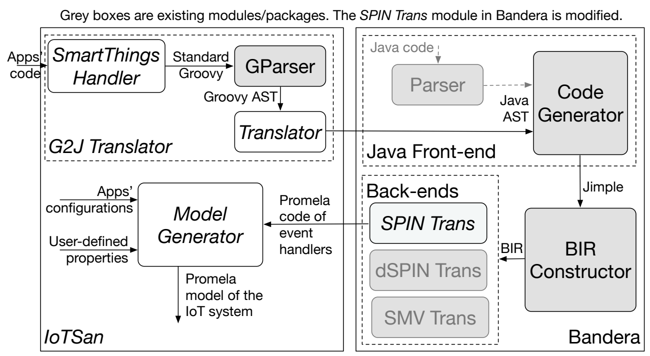


Figure 5: IoTSan is built around Bandera.

input Groovy block, translate each Groovy statement into Java, add the result to a list of Java statements, and build a Java block from the result list. To implement these, we extended the Groovy compiler (*org.codehaus.groovy*) which is then integrated into the Bandera's front-end.

Handling SmartThings' language features: There are several new language syntaxes introduced in SmartThings. Our *SmartThings Handler* parses these new syntaxes and converts them into vanilla Groovy code using specifications based on the domain knowledge of SmartThings. For instance, (as can be seen in in Figure 1) each *input* function defines a global variable (or a class field) of the app. Therefore, we traverse the Groovy's AST of the app and visit all *input* functions to extract all global variables of the app. In addition, apps can use some predefined objects or variables (*e.g.*, *location*) and APIs (*e.g.*, *setLocationMode*), which are not defined in vanilla Groovy. Therefore, we manually add definitions of these global objects.

Type inference: Although the Groovy Compiler *org.codehaus. groovy* already has a sub-package *CompileStatic* for performing static type inference, it only works when the argument type and

```
private onSwitches() {
  switches + onSwitches
}
```

(a) Groovy's code

```
private STSwitch[] onSwitches(){
  STSwitch[] STSwitchArr0;
  int arrIndex0 = 0;
  int index3 = 0;
  while(index3 < TheBigSwitch_switches.length){
    STSwitch it = TheBigSwitch_switches[index3];
    STSwitchArr0[arrIndex0] = it;
    arrIndex0++;
    index3++;
  }
  int index4 = 0;
  while(index4 < TheBigSwitch_onSwitches.length){
    STSwitch it = TheBigSwitch_onSwitches[index4];
    STSwitchArr0[arrIndex0] = it;
    arrIndex0++;
    index4++;
  }
  return STSwitchArr0;
}
```

(b) Corresponding Java's code

Figure 6: Example of translating a Groovy method into the corresponding Java's method.

the return type of a method are given. In other words, a variable declared inside a method can take different runtime types depending on the argument type. Thus, we still need to infer the argument and return type statically. To do so, we consult the calling context of each method invocation by recursively tracking the arguments and return values to their corresponding anchor points—declaration of variables with explicit types (Groovy supports static typing as well), assignment to constant values (*e.g.*, we can infer that the type of variable *a* is numeric from *def a = 0*), assignment to return values of known APIs, and known objects and their properties. The inference procedure works roughly as follows. When traversing the AST of a method, we store the names and data types of variables at anchor points; the types of other variables are inferred by propagating the types from anchor points. This is done iteratively until we find no more new variables whose type can be inferred.

Handling Groovy's built-in utilities: Another challenge arises when we translate Groovy into Java for use with Bandera. We find that Bandera understands only a very basic set of Java. For instance, it supports only the *array* type natively. In contrast, Groovy's collection types (*e.g.*, *Collection*, *List*, *ArrayList*, *Set*, *Map*, and *HashSet*) all need to be translated into Java's *array* type. We support the popular collection types that are commonly used in smart apps. An example is shown in Figure 6 that translates one Groovy list into a corresponding Java implementation using array. Since the type of *switches* and *onSwitches* is *List of STSwtich*, we infer the return type of *onSwitches()* method as *List of STSwtich*, which is translated into Java's array type (*i.e.*, *STSwitch[]*). The + operation on *List* type (line 2 in Figure 6a) is automatically translated into corresponding Java's code (lines 2-17 in Figure 6b). Finally, since this method is a non-void method, we add an explicit *return* statement (line 18 in Figure 6b).

7 CONFIGURATION EXTRACTOR

IoT platforms typically provide a mobile companion app and/or a web-based app to manage and configure smart apps and devices. For Samsung SmartThings, we develop a crawler in Java, using the *Jsoup* package to automatically extract the system's configuration from the management web app [78]. Given a SmartThings account (user's name and password), the crawler logs in to the management web app and extracts (i) installed devices, (ii) installed smart apps, and (iii) configurations of apps. Moreover, whenever a user installs a new generic smart device (*e.g.*, a smart power outlet), we have an interface to get the device association info (*e.g.*, this new outlet is used to control an AC) from the user. The extracted configuration is then saved to a file and used later by the *Model Generator*. The process is straightforward and we omit the details in the interest of space.

8 MODEL GENERATOR

Modeling an IoT system: To correctly verify safety properties, we need to model two key components (not part of the app code): (i) the IoT platform and its interactions with smart apps and (ii) IoT devices and their interactions with smart apps. IoT platforms [3, 4, 46, 63, 73] typically provide apps with some methods to register callback functions (*i.e.*, event handlers). Based on apps' configurations provided by the *Configuration Extractor*, we model these special registration functions so as to invoke callbacks at appropriate times.

We model IoT devices (sensors and actuators) as per their specifications. Note that both sensors and actuators can generate events of interest to apps. For instance, a motion sensor can generate motion active/inactive events whereas a door lock (actuator) can generate status update events (locked/unlocked). Each device is modeled as having an event queue and a set of notifiers to inform the smart apps that have subscribed to specific types of events. Currently, we support 30 different IoT devices. Note here that we model events generated by the environment (*e.g.*, *sunrise* and *sunset*) as sensor generated inputs and location mode changes (*e.g.*, *Home*, *Away*, and *Night*) as actuations; thus inputs such as users leaving home (sensed input) can trigger the mode to change from *Home* to *Away* (actuation).

We model system time as a monotonically increasing variable. We extract the triggering times and callback functions from the scheduling method calls. The callback functions are then triggered at appropriate times based on the value of the modeled system time.

Algorithm 1 shows the pseudo code of the main process that models behaviors of an IoT system. The model checker enumerates all possible permutations of the input physical events up to a maximum number of events per user's configuration to exhaustively verify the system. At each iteration, a sensor and a corresponding physical event in the permutation space are selected (line 2). Then, the selected sensor updates its state and event queue, and notifies its subscribers of the state change event (line 3). When an event is pending, it is dispatched to the subscribed apps and the corresponding event handlers of apps are invoked to handle the event (lines 4-6). Each event handler may send some commands to some actuators, which may generate some new cyber events and trigger event handlers of the subscribers.

To model natural or induced (*e.g.*, using jamming [69]) device/-communication failures, when generating a sensor event we enumerate two scenarios: (i) the sensor is available/online and (ii) the sensor is unavailable/offline. Similarly, whenever receiving a command from a smart app, an actuator may be either online or offline. If a device is offline, it will not change its state and hence *not* broadcast a state change event to its subscribers. If a device is online,

Algorithm 1 Modeling an IoT system

```
1:  for $i$ = 1 to maximum number of events do {Main event loop of an IoT system}
2:      Select a sensor and a corresponding event in the permutation space {Generate a physical event}
3:      sensor_state_update(evt)
4:      while any event pending do
5:          dispatch_event(evt) {Dispatch the pending event to the subscribed apps and invoke the corresponding app_event_handler(evt) to process the event}
6:      end while
7:  end for
    {sensor_state_update(evt)}
8:  if $evt \neq$ current state of the sensor then
9:      Add the $evt$ to the event queue
10:     Update the state of the sensor
11:     Notify the subscribers of the state change event
12: end if
    {app_event_handler(evt)}
13: if some conditions hold then
14:     Send some command to some actuator {Invoke actuator_state_update(evt), which may subsequently generate some new event}
15: end if
    {actuator_state_update(evt)}
16: Verify conflicting and repeated commands violations
17: if $evt \neq$ current state of the actuator then
18:     Add the $evt$ to the event queue
19:     Update the state of the actuator
20:     Notify the subscribers of the state change event
21: end if
```

Table 4: Sample safe physical states.

Category	Number of properties	Sample property
Thermostat, AC, and Heater	5	Temperature should be within a predefined range when people are at home
Lock and door control	8	The main door should be locked when no one is at home
Location mode	3	Location mode should be changed to Away when no one is at home
Security and alarming	14	An alarm should strobe/siren when detecting smoke
Water and sprinkler	3	Soil moisture should be within a predefined range
Others	5	Some devices should not be turned on when no one is at home

the communication (*i.e.*, sending a state change event or receiving a command) between the device and the hub/cloud may either succeed or fail (we enumerate both cases).

Concurrency Model: Since an app's event handler is only triggered by the subscribed event(s) and event handlers of different apps do not share any global variable [3, 4, 46, 63, 73], the execution of an app's event handler can be considered as atomic. This means that the concurrency level of a model only depends on the interleaving of apps' event handlers. To model a concurrent IoT system therefore, we only need to verify the behaviors of the system with interleavings of all of the external events (*e.g.*, smoke detected) sensed by sensors and internal events (*e.g.*, unlocked) caused by apps' behaviors. Even though the events are concurrent, the interleaving is in fact reflected by the order of the (incoming) events processed by event handlers, *i.e.*, we can obtain the strict concurrency by considering all order permutations of external and internal events. However, this approach takes a very long verification time as the number of events grow, and causes the state space to explode. Instead, we can obtain a weaker concurrency by considering the permutations of only external events in a sequential design shown in Algorithm 1. This implicitly assumes that the internal events associated with an external event are handled atomically in order. It is unclear if enforcing strict concurrency would lead to the discovery of more unsafe states. We experiment with the two design options with several small systems and find that the sequential approach offering weak consistency, discovered all violations that the strict concurrent model found. Based on this, we use the sequential approach given that it significantly mitigates the time complexity of execution.

The IoT system model in Promela: With the concurrent approach, each device and smart app is modeled by a process (*i.e.*, *proctype*). There is also a process for generating the sensed and environmental events. The processes communicate with each other using message passing (*i.e.*, *chan*). We use a single process for the whole system with our sequential design, using *inline* methods to model the behavior of devices and smart apps. The devices, smart apps, and event generators, communicate via shared global variables.

Safety Properties: We seek to verify 45 properties of the following types:

- *Free of conflicting commands* [68]: When a single external event happens, an actuator should not receive two conflicting commands (*e.g.*, both on and off) – (1 property).
- *Free of repeated commands*: When a single event happens, an actuator should not receive multiple repeated commands of the same type or with the same payload – (1 property). The latter could indicate a potential DoS or replay attack.
- *Safe physical states*: Table 4 shows some sample safe physical states that the user desires the system to satisfy. These kinds of properties can be verified using linear temporal logic (LTL) [5] – (38 properties). We envision that a more complete list will likely be provided by safety regulations associated with the IoT industry in the future.
- *Free of other known suspicious app behaviors—security-sensitive command and information leakage*: Examples of security-sensitive commands are *unsubscribe* (disabling an app's functionality) and creating fake events (*e.g.*, an app may generate a "smoke detected" event when there is no smoke in the physical environment); we raise violations when such commands are executed. Information leakage can occur with *sending SMS* and *using network interfaces*. When *sending SMS* is triggered, for instance, we check whether the recipient matches with the configured phone number to prevent leakage – (4 properties).
- *Robustness to device/communication failure*: An app should quickly check that a command sent to an actuator was acted upon to be robust to device and communication failures. Upon detecting a failure, the app should notify users via SMS/Push messages. This property can be verified using LTL as well – (1 property).

Note that we provide users with an interface to select the list of safety properties they want to verify. Based on the device association information (recall § 7) provided by the *Configuration Extractor*, the LTL format of the selected properties are automatically generated.

Example: Consider the smart home of a single owner Alice (say), which comprises of a smart lock that controls the main door viz., `Door Lock`, and a presence sensor viz., `Alice's Presence` (which checks if Alice is at home). Assume that Alice installs two smart apps: *Auto Mode Change*, which manages the location mode based on the events from `Alice's Presence` and, *Unlock Door*, which unlocks the `Door Lock` based on explicit user input or a "location mode" change event. When this system is analyzed by the model checker, a violation is detected as described below.

Figure 7 shows the (filtered) violation log (a counter-example) output by Spin. The format of each line in the violation log is as follows: file name (*SmartThings0.prom*), line number, state number, and the executed code. In particular, the counter example has the following steps. **(1)** The event *not present* is generated by `Alice's presence` if Alice leaves home (line 1) and its subscribers are notified of this state change (line 2). **(2)** The app *Auto Mode Change* reads and processes this state change event (lines 3-5) and notifies the location manager to change the location mode to *Away* (line 6). **(3)** The location manager changes its mode and notifies its subscribers of this change (lines 7-8). **(4)** The app *Unlock Door* reads and processes this mode change event (lines 9-10) and sends an *unlock* command to the device `Door Lock` (line 11), which unlocks the door (lines 12-13). Thus, the system enters an unsafe physical state (*i.e.*, the main door is unlocked when no one is at home) (lines 14-15).

Upon closer inspection, the description of *Unlock Door* suggests that it unlocks the door *only upon user input*. However, in practice, it also unlocks the door whenever the location mode changes (*i.e.*, there is an inconsistency between the app's description and its implementation).

9 OUTPUT ANALYZER

The *Output Analyzer attributes* a violation to either a misconfiguration or a malicious app using a heuristic-based algorithm. The algorithm consists of two phases. In the first phase, when a user installs a new smart app, the output analyzer enumerates all possible configurations for this app. It verifies if the user-defined properties hold with each configuration independently. If the proportion of

```
 1  SmartThings0.prom:2690 (state 295) [generatedEvent.evtType = notpresent]
 2  SmartThings0.prom:2609 (state 332) [g_STPresSensorArr.element[STPresSensorIndex].subNotifiers[index2] = g_STPresSensorArr.element[
               STPresSensorIndex].subNotifiers[index2] + 1]
 3  SmartThings0.prom:2725 (state 757) [((g_STPresSensorArr.element[AutoModeChange_people.element[0].gArrIndex].subNotifiers[
               AutoModeChange_people.element[0].eventCountIndex] > 0))]
 4  SmartThings0.prom:2728 (state 759) [g_STPresSensorArr.element[AutoModeChange_people.element[0].gArrIndex].subNotifiers[AutoModeChange_people
               .element[0].eventCountIndex] = g_STPresSensorArr.element[AutoModeChange_people.element[0].gArrIndex].subNotifiers[AutoModeChange_people
               .element[0].eventCountIndex] - 1]
 5  SmartThings0.prom:1913 (state 937) [(!((location.mode == AutoModeChange_newMode)))]
 6  SmartThings0.prom:2308 (state 1797) [ST_Command.evtType = Away]
 7  SmartThings0.prom:2438 (state 1765) [location.mode = HandleLocationEvt_mode]
 8  SmartThings0.prom:2451 (state 1788) [location.subNotifiers[index0] = location.subNotifiers[index0] + 1]
 9  SmartThings0.prom:2704 (state 346) [((location.subNotifiers[UnlockDoor_location] > 0))]
10  SmartThings0.prom:2707 (state 348) [location.subNotifiers[UnlockDoor_location] = location.subNotifiers[UnlockDoor_location] - 1]
11  SmartThings0.prom:1832 (state 596) [ST_Command.evtType = unlock]
12  SmartThings0.prom:2357 (state 665) [HandleSTLockEvt_state = 48]
13  SmartThings0.prom:2553 (state 703) [g_STLockArr.element[m_JJJCTEMP_0.gArrIndex].currentLock = HandleSTLockEvt_state]
14  spin: _spin_nvr.tmp:3, Error: assertion violated
15  spin: text of failed assertion: assert(!(!((((g_STPresSensorArr.element[alicePresence_STPresSensor].currentPresence != 18)||(g_STLockArr.
               element[doorLock_STLock].currentLock!=48))))))
```

Figure 7: Example violation log (filtered).

violations (violation ratio) is greater than a predefined threshold (*e.g.*, 90%), the new smart app is attributed as a malicious app.

If this is not the case, in the second phase, the new app is verified in conjunction with other apps that were previously installed by the user. Again, all configurations are considered. If the violation ratio is greater than a predefined threshold, the new app is attributed as a bad app and a report is provided to the user. Otherwise, the violation is attributed to misconfiguration and suggestions of safe configurations with regards to the user defined properties are provided. If there is no violation, a successful verification is reported.

10 EVALUATIONS

Our experiments (model checking) are performed on a MacBook Pro with macOS Sierra, 2.9 GHz Intel Core i5, 16 GB 1867 MHz DDR3, and 256 GB SSD. We check if there are violations of the properties discussed in §8. We also look at other performance metrics such as the running times, and the scale ratio (which quantifies the reduction in the number of event handlers to be jointly verified) to evaluate IoTSan.

10.1 Test Cases and Configurations

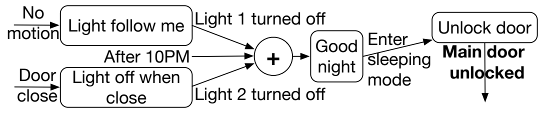
We perform four different sets of experiments described below. The first three examine the fidelity with which bad apps and configurations are identified. The last set evaluates the performance of different design choices we make.

Market apps with expert configurations: We check the safety properties with 150 apps (assuming they are benign) from the SmartThings’ market place [19, 77, 78]. We (the authors) came up with independent configurations for the apps (based on common sense with regards to how the apps may be used). To illustrate, consider the app *Virtual Thermostat*, the required input to which is shown in Figure 1. Assume that the following devices are deployed: (1) one temperature sensor (myTempMeas), (2) one outlet to control the heater (myHeaterOutlet), (3) one outlet to control the air conditioner (myACOutlet), (4) one outlet to control the light in the living room (livRoomBulbOutlet), (5) one outlet to control the light in the bedroom (bedRoomBulbOutlet), (6) one outlet to control the light in the bathroom (batRoomBulbOutlet), (7) one motion sensor in the living room (livRoomMotion), and (8) one motion sensor in the bathroom (batRoomMotion). Our configuration is as follows: myTempMeas for the temperature sensor (line 3 in Figure 1), myACOutlet for “outlets" (line 7 in Figure 1), 75 as the “setpoint" temperature if people are present (line 9 in Figure 1), livRoomMotion for “motion" (line 12 in Figure 1), 10 “minutes" for turning off the AC/heater when no motion is sensed (line 15 in Figure 1), 85 as the “emergencySetPoint" temperature at which the AC is turned on (to set) regardless of whether people are present (line 18 in Figure 1), and “cool" for “mode" (line 21 in Figure 1).

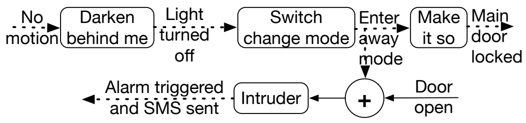
We randomly divide the 150 apps into six groups (25 apps per group) with one configuration each, and feed them into IoTSan. Upon encountering a violation, we remove the minimum number of the associated apps (*e.g.*, if there are two apps causing conflicting commands, we randomly remove one of them); we then iterate the process. The experiment stops when no violation is detected. These experiments are performed with and without device/communication failures.

Market apps with non-expert configurations: To eliminate biases, we also conduct a user study where we request 7 independent student volunteers to configure 10 groups of apps with the assumption that they would deploy them at home. Each group comprises of about 5 related apps (as determined by our app dependency analyzer). A group receives one configuration from each volunteer and this leads to a total of 70 configurations. Our Office of Research Integrity determined that there was no need to go through an IRB approval process (since no private information is collected).

Malicious apps: We consider 25 malicious apps created in [51]. In this set, we find that only 9 apps are relevant to our evaluations (*e.g.*, affect the physical state and can be compiled correctly by the SmartThings’ own web-based IDE). There are four apps that IoTSan cannot currently handle viz., *Midnight Camera*, *Auto Camera*, *Auto Camera 2*, and *Alarm Manager*, since they dynamically discover and control the devices in the system; we will extend IoTSan to handle such apps in future work. We evaluate whether IoTSan correctly attributes these malicious apps when they are installed together with other apps. The configurations of the 9 malicious apps are identical to those in [51]. We also choose 11 potentially bad apps (found via the previous experiments) from the market place for a total of 20 bad apps. In conjunction, we select 10 good



(a) Example violation due to bad app interactions.



(b) Example violation due to a device failure. Dotted arrows are expected events that do not occur due to the failure of the motion sensor.

Figure 8: Violation examples: boxes depict apps and high level abstractions are shown for inputs/outputs.

apps from the market place to create a reasonable input set. Here, we specifically evaluate the fidelity of our attribution module.

Performance: We compare the performance of concurrent *versus* sequential design. We use two bad groups of apps viz., (Auto Mode Change, Unlock Door) and (Brighten Dark Places, Let There Be Dark), and one good group of apps viz., (Good Night, It's Too Cold) that control 3 switch devices, 3 motion sensors, and 1 temperature measurement sensor.

10.2 Identifying Unsafe Configurations

Market apps with expert configurations: Table 5 summarizes the results from our first set of experiments in the absence of device and communication failures. The apps in parenthesis jointly cause a violation. We find 38 violations of 11 properties, some of which can be very dangerous from a user's perspective. For example, there is violation where "The main door is unlocked when people are sleeping at night", which involves 4 apps. The interactions between the apps that lead to this violation is shown in Figure 8a: when lights are turned off at night a mode change is initiated by the `Good Night` app, which in turn causes the unsafe action of unlocking the main door by the `Unlock door` app.

Device/communication failures cause violations of 9 additional properties with some dangerous cases. One such case is showcased in Figure 8b. When people leave home, the `Make it so` app should automatically lock the entrance door; however, due to the failure of the motion sensor, the `Make it so` app is not triggered and thus, the door is left unlocked. Moreover, this failure also causes *NO* notification to be sent to law enforcement upon physical intrusion. An alarming discovery is that none of the analyzed apps check if the commands sent to the actuators were actually carried out (which might not be the case if the device has failed).

Market apps with non-expert configurations: The verification results from the second set of experiments are in Table 6. From 10 groups of apps with 70 configurations, we find 97 violations of 10 properties. For example, the property "An AC and a heater are both turned on" is violated by 21 configurations across 5 groups. Note that in some configurations multiple properties are violated and thus, the number of violations is more than the number of configurations.

10.3 Violation Attribution

IoTSan attributes *all* of the ContexIoT's malicious apps [51] correctly when each is independently considered with violation ratios of 100 % (recall §9). Two apps violated the information leakage property as the command *httpPost* was executed. Two apps violated the "using security-sensitive command property", *i.e.*, they generated fake carbon monoxide detection events and an *unsubscribe* is executed. The remaining 5 apps violated safety properties in the physical space, *e.g.*, *a main door is unlocked when no one is at home* and, *when smoke is detected, a water valve switch is turned off.* From among the 11 market apps, 6 were detected with a 100% violation ratio, both when verified independently and in conjunction with other apps; they were thus attributed as bad apps. The remaining were attributed to cause violations (with 70% or lower violation ratio) due to bad configurations (there existed safe configurations with no violations).

10.4 Scalability

Table 7a shows the scalability benefits of our app dependency analyzer in the above experiments with 150 market apps. In this table, "*Original Size*" is the total number of event handlers of a group and "*New Size*" is the number of event handlers of the largest related set after running the *App Dependency Analyzer* module. On average, *App Dependency Analyzer* reduced the problem size by a factor 3.4x.

10.5 Concurrent vs. Sequential

Model checkers using both concurrent and sequential design detected all violations within 1 second. Table 7b shows the runtimes of the two models with a good group of apps (2 apps and 7 devices), which does not violate any property. We see that sequential design significantly reduces the runtime of the verification. Note that *forever* means the experiment ran for a week and then was forced to stop. Moreover, we also verified the runtime of our sequential approach with a much bigger system, which comprises of 5 related apps and 10 devices and does not have any violation. As shown in Table 8, the verification time for 10 events is about 5 hours, which is quite reasonable for a laptop with limited computing resources.

11 DISCUSSION

Application to other IoT Platforms: For ease of exposition, our narrative integrated some aspects of implementation specific to SmartThings, when describing the design of IoTSan. Conceptually, the design of IoTSan applies to other IoT platforms. To illustrate, given its recent popularity we choose IFTTT (IF This Then That [46]) [55, 62, 84] to show that this is the case. IFTTT is a task automation platform for IoT deployments. An IFTTT rule (also called applet) comprises of two main parts: "Trigger Service" (This) and "Action Service" (That). To apply IoTSan to IFTTT, most of the modules (*i.e.*, *App Dependency Analyzer*, *Model Generator*, and *Output Analyzer*) can be reused almost as is; the relatively big change will be in the *Translator.*

IFTTT to Java Translator: We use the crawler of [62] to fetch the published applets from IFTTT website into a *json* file. We then developed an *IFTTT Handler* in Java based on the *org.json.simple* package to extract the subscribed device and event from the trigger service, and the controlled device and expected command from

Table 5: Verification results with market apps.

Violation type	Number of violations	Example violated property	Apps related to example
Conflicting commands	8	A light receives "on" and "off" simultaneously	(Brighten Dark Places, Let There Be Dark)
Repeated commands	10	A light receives repeated "on" commands	(Automated light, Brighten My Path)
Unsafe physical states	20	A heater is turned off at night when temperature is below a predefined threshold	(Energy Saver)
		The main door is unlocked when people are sleeping at night	(Light Follows Me, Light Off When Close, GoodNight, Unlock Door)

Table 6: Verification results with market apps, with volunteer configuration.

Violation type	Number of violated properties	Number of violations
Conflicting commands	1	19
Repeated commands	1	12
Unsafe physical states	8	66

Table 7: (a) Scalability with dependency graphs. (b) Runtimes with concurrent and sequential design.

(a)

Group	Original Size	New Size	Scale Ratio
1	37	11	3.4
2	27	5	5.4
3	34	23	1.5
4	30	12	2.5
5	42	19	2.2
6	34	6	5.7
Mean scale ratio			3.4

(b)

Number of events	Concurrent	Sequential
1	1s	1s
2	56.5s	1s
3	139m	1s
4	forever	1s
5		1s
6		4.2s
7		16.3s

Table 8: Verification time vs. number of events.

Number of events	6	7	8	9	10	11
Verification time	6.61s	50.9s	396s	49.83m	5.89h	23.39h

the action service of each IFTTT rule. The translation is relatively simple. Each rule is considered as an app, which has only a single event handler, in IoTSan and is translated into a Java class. Each event handler (*i.e.*, a Java method) has only a single instruction (*i.e.*, the expected command); the subscribed device and controlled device become class fields. Even though the technical details of *IFTTT Handler* are somewhat different from *SmartThings Handler*, the translation procedures are very similar (*e.g.*, all Java objects and grammars are exactly the same).

Minor changes in Model Generator: Each service is mapped onto (modeled as) a sensor device(s) or an actuator device(s). We have modeled 8 popular IoT-related services based on the events/actions they provides on the IFTTT website. For example, Amazon Alexa [3] and Google Assistant are modeled as sensor devices; Nest Thermostat is modeled as an actuator device. The difference is that Samsung SmartThings inherently provides handlers for several kinds of devices (*e.g.*, outlet, lock, motion sensor, and contact sensor). The change needed is to add more device types to the collection of modeled devices.

We have validated our basic IFTTT prototype implementation with 10 IoT rules/applets (from [46]) assuming that all of these rules are installed in a smart home. We perform limited experiments and as shown in Table 9 (hyperlinks to a rule –e.g., rule #1 – can be seen by clicking on the rule), we find 7 violations of 4 unsafe physical states.

Table 9: Verification results with IFTTT rules.

Violated properties	*Related rules*
Siren/strobe is not activated when intruder (*i.e.*, motion) is detected	(rule #1, rule #4), (rule #3, rule #4)
Siren/strobe is activated when no intruder is detected	(rule #2)
The main/front door is unlocked when no one is at home	(rule #5), (rule #6)
A phone call is not triggered when intruder is detected	(rule #7, rule #10), (rule #8, rule #10)

Limitations: While our prototype of IoTSan has been shown to be very effective in identifying bad apps and unsafe configurations, it has the following limitations. *First,* the Spin model checker has a predefined threshold for the size of Promela code (and cannot handle a file size greater than this). Depending on apps' source code sizes and dependencies among the apps, IoTSan can handle a system with about 30 apps. We assume that users are unlikely to have many more than this today and will investigate further scalability in the future. *Second*, we require smart apps to explicitly subscribe to specific devices they want to control and cannot handle smart apps that dynamically discover devices and interact with them. Such apps are very dangerous since they can control any device without permissions from users. Identifying such apps and ensuring that they do not compromise the physical state is beyond the scope of this effort. *Third*, in Algorithm 1, we let the model checker enumerate all possible permutations of the event types; thus, it may consider scenarios that are unlikely to happen in the real world (*e.g.*, the temperature is set to a minimum value in the first iteration and set to a maximum value in the second one). However, we include these scenarios to catch bad or malicious apps. If such scenarios can be eliminated, the state explosion issue can be further mitigated. *Fourth*, we do not explicitly model the behavior of the physical environment after an actuator executes a command (*e.g.*, the system temperature should increase after a heater is turned on). However, such physical changes are implicitly covered by the way the model checker exhaustively verifies a system. *Fifth*, the G2J Translator currently does not support heterogeneous collections (*e.g.*, a list, array, or map that stores entries of different types) and dynamic features (*e.g.*, overloading operator and generic data types). Note that most of the SmartThings apps do not use these features.

12 RELATED WORK

IoT Security: Current research on IoT security can be roughly divided into three categories that focus on devices [30, 39, 71], protocols [31, 40, 57, 70], and platforms. There have been efforts addressing information leakage and privacy [7, 11, 13, 74, 89, 91], and vulnerabilities of firmware images [21]. Fernandes *et al.*, have recently reported security-critical design flaws in the IoT permission model that could expose smart home users to significant harm such as break-ins [27]. To address these, several efforts [28, 51, 83, 87] have proposed modifications to a smart app's source code and the platform, to enforce good behaviors of smart apps at run time. In contrast, our work statically identifies possible violations of given physical/cyber safety properties of IoT systems without requiring any app modifications.

Model Checking: Model checking has been used to verify system-level threats [65–67] and basic correctness properties [14, 22, 55, 68] of IoT systems. In contrast with these efforts, IotSan targets developing a practical platform for ensuring the physical safety of today's IoT systems. It not only addresses the practical challenges (*e.g.*, scale issues and making Groovy amenable to model checking) in identifying configurations that violate user properties relating to the physical state, but also addresses robustness (failures) and security issues (malicious app attribution). Table 1 shows what IotSan offers compared to the most related recent systems.

13 CONCLUSIONS

Badly designed apps, undesirable interactions between installed apps and/or device/communication failures can cause an IoT system to transition into bad states. In this paper, we design and prototype a framework IotSan that uses model checking as a basic building block to identify causes for bad physical/cyber states and provides counter-examples to exemplify these causes. IotSan addresses practical challenges such as alleviating state space explosion with model checking, and automatic translation of app code into a form amenable for model checking. Our evaluations show that IotSan identifies many (sometimes complex) unsafe configurations, and flags considered bad apps with 100% accuracy.

ACKNOWLEDGMENT

We thank our shepherd Cole Schlesinger and the anonymous reviewers for their constructive comments which helped us significantly improve the paper.

The effort described in this article was partially sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Cooperative Agreement W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon.

REFERENCES

[1] M. Ahmad. 2014. Reliability Models for the Internet of Things: A Paradigm Shift. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 52–59.

[2] R. Alena, R. Gilstrap, J. Baldwin, T. Stone, and P. Wilson. 2011. Fault tolerance in ZigBee wireless sensor networks. In *2011 Aerospace Conference*. 1–15.

[3] Amazon. 2018. Alexa. https://developer.amazon.com/alexa. (June 2018).

[4] Apple. 2018. HomeKit. https://developer.apple.com/homekit/. (June 2018).

[5] C. Baier and J. P. Katoen. 2008. *Principles of Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England.

[6] Brian Belleville, Patrick Biernat, Adam Cotenoff, Kevin Hock, Tanner Prynn, Sivaranjani Sankaralingam, Terry Sun, and Daniel Mayer. 2018. Internet of Things Security. https://www.nccgroup.trust/us/our-research/internet-of-things-security/. (2018).

[7] Elisa Bertino, Kim-Kwang Raymond Choo, Dimitrios Georgakopolous, and Surya Nepal. 2016. Internet of Things (IoT): Smart and Secure Service Delivery. *ACM Trans. Internet Technol.* 16, 4 (Dec. 2016).

[8] August Betzler, Carles Gomez, Ilker Demirkol, and Josep Paradells. 2014. A Holistic Approach to ZigBee Performance Enhancement for Home Automation Networks. *Sensors* 14, 8 (2014), 14932–14970.

[9] Dirk Beyer and Thomas Lemberger. 2017. Software Verification: Testing vs. Model Checking. In *Hardware and Software: Verification and Testing*. Springer International Publishing, 99–114.

[10] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. *Symbolic Model Checking without BDDs*. Springer, Heidelberg, 193–207.

[11] Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N. Asokan. 2015. Smart and secure cross-device apps for the internet of advanced things. In *Proc. Financial Cryptography and Data Security*. Puerto Rico, US.

[12] T. Cattel. 1994. Modelization and verification of a multiprocessor realtime OS kernel. In *Proc. 7th FORTE Conference*. Bern, Switzerland, 35–51.

[13] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *USENIX Security 18*. Baltimore, MD.

[14] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *USENIX ATC 18*. Boston, MA.

[15] H. Chandra, E. Anggadjaja, P. S. Wijaya, and E. Gunawan. 2016. Internet of Things: Over-the-Air (OTA) firmware update in Lightweight mesh network protocol for smart urban development. In *APCC 16*. 115–118.

[16] J. Chaves. 1991. Formal methods at AT&T, an industrial usage report. In *Proc. 4th FORTE Conference*. Sydney, Australia, 83–90.

[17] Yunja Choi. 2007. From NuSMV to SPIN: Experiences with model checking flight guidance systems. *Springer Formal Methods in System Design* 30, 3 (01 Jun 2007), 199–216.

[18] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. *Tools for Practical Software Verification*. Springer, Heidelberg. 1–30 pages.

[19] SmartThings Community. 2018. Community Smart Apps. https://community.smartthings.com/c/smartapps. (Sept. 2018).

[20] Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. 2012. *Context-Bounded Model Checking with ESBMC 1.17*. Springer, Heidelberg, 534–537.

[21] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *Proc. USENIX Security 14*. San Diego, CA, USA, 95–110.

[22] Jason Croft, Ratul Mahajan, Matthew Caesar, and Madan Musuvathi. 2015. Systematically Exploring the Behavior of Control Programs. In *USENIX ATC 15*. Santa Clara, CA, 165–176.

[23] CropMetrics. 2018. Irrigation management. http://cropmetrics.com/. (2018).

[24] Dolly Das and Bobby Sharma. 2016. General Survey on Security Issues on Internet of Things. *International Journal of Computer Applications* 139, 2 (2016).

[25] Yifei Dong, Xiaoqun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David S. Warren. 1999. *Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools*. Springer, Heidelberg, 74–88.

[26] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. 2010. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks Using Model Checking. In *USENIX Security 10*. USA.

[27] E. Fernandes, J. Jung, and A. Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *Proc. IEEE Symposium on Security and Privacy*. San Jose, CA, USA, 636–654.

[28] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security 16*. USA, 531–548.

[29] Joe Filippello. 2018. SmartSense Presence Sensor Failure. https://community.smartthings.com/t/smartsense-presence-sensor-failure/16644/9. (June 2018).

[30] D. Fisher. 2018. Pair of Bugs Open Honeywell Home Controllers Up to Easy Hacks. https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/113965/. (June 2018).

[31] B. Fouladi and S. Ghanoun. 2013. *Honey, I'm home!! - hacking z-wave home automation systems*. Black Hat, Las Vegas, NV, USA.

[32] Patrice Godefroid and Koushik Sen. 2018. *Combining Model Checking and Testing*. Springer International Publishing, 613–649.

[33] Dorset Gray. 2018. Devices Offline and Unavailable. https://community.smartthings.com/t/devices-offline-and-unavailable/100248. (June 2018).

[34] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. 2014. Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence* (2014), 315–349.

[35] Amy Groden-Morrison. 2018. How the Internet of Things will Drive Mobile App Development. https://www.alphasoftware.com/blog/internet-of-things-will-drive-mobile-app-development/. (June 2018).

[36] Groovy. 2018. Type checking extensions. http://docs.groovy-lang.org/next/html/documentation/type-checking-extensions.html. (June 2018).

[37] John Hatcliff and Matthew Dwyer. 2001. *Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software.* Springer, Heidelberg, 39–58.

[38] John Hatcliff and Matthew Dwyer. 2018. About Bandera. http://bandera.projects.cs.ksu.edu/. (June 2018).

[39] A. Hesseldahl. 2018. A Hacker's-Eye View of the Internet of Things. https://www.recode.net/2015/4/7/11561182/a-hackers-eye-view-of-the-internet-of-things. (June 2018).

[40] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner. 2016. Smart locks: Lessons for securing commodity internet of things devices. In *ACM ASIACCS 16*. China, 461–472.

[41] G. J. Holzmann. 1994. Proving the value of formal methods. In *Proc. 7th FORTE Conference*. Bern, Switzerland, 385–396.

[42] G. J. Holzmann. 1994. The theory and practice of a formal method: NewCoRe. In *13th IFIP World Computer Congress*. Germany.

[43] G. J. Holzmann. 1997. The Model Checker Spin. In *IEEE Transaction on Software Engineering*. Vol. 23. 279–295.

[44] G. J. Holzmann. 1998. An Analysis of Bitstate Hashing. In *Formal Methods in System Design*, Vol. 13. 289–307.

[45] IBM. 2018. IBM IoT for manufacturing. https://www.ibm.com/internet-of-things/industries/iot-manufacturing. (2018).

[46] IFTTT. 2018. IFTTT Homepage. https://ifttt.com/. (June 2018).

[47] Texas Instruments. 2018. EZSync CC2531 Evaluation Module USB Dongle. http://www.ti.com/tool/CC2531EMK. (June 2018).

[48] Intel. 2018. Smart Buildings. https://www.intel.com/content/www/us/en/internet-of-things/smart-building-solutions.html. (June 2018).

[49] BI Intelligence. 2018. Here's how the Internet of Things will Explode by 2020. http://www.businessinsider.com/iot-ecosystem-internet-of-things-forecasts-and-business-opportunities/2016-2. (June 2018).

[50] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 21.

[51] Y. J. Jia, Q. A. Chen, S. Wangy, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS'17*. USA.

[52] Daniel Kroening and Michael Tautschnig. 2014. *CBMC – C Bounded Model Checker.* Springer, Heidelberg, 389–391.

[53] J. S. Lee, Yuan-Ming Wang, and C. C. Shen. 2012. Performance evaluation of ZigBee-based sensor networks using empirical measurements. In *IEEE CYBER 12*. 58–63.

[54] Flavio Lerda, Nishant Sinha, and Michael Theobald. 2003. Symbolic Model Checking of Software. *Elsevier Electronic Notes in Theoretical Computer Science* 89 (Sept. 2003), 480–498.

[55] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: Building an Internet of Safe Things. In *ACM IPSN '15*. USA, 298–309.

[56] Logitech. 2018. Harmony Hub. https://www.logitech.com/en-us/product/harmony-hub. (June 2018).

[57] N. Lomas. 2018. Critical Flaw IDed In ZigBee Smart Home Devices. https://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/. (June 2018).

[58] Medria Solution. 2018. Livestock monitoring. http\protect\kern+.2222em\relax//www.medria.fr/en/solutions/. (2018).

[59] M. U. Memon, L. X. Zhang, and B. Shaikh. 2012. Packet loss ratio evaluation of the impact of interference on ZigBee network caused by Wi-Fi (IEEE 802.11b/g) in e-health environment. In *2012 IEEE 14th International Conference on e-Health Networking, Applications and Services (Healthcom)*. 462–465.

[60] Andrew Meola. 2018. How the Internet of Things will affect security & privacy. http://www.businessinsider.com/internet-of-things-security-privacy-2016-8. (June 2018).

[61] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. *LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR.* Springer Berlin Heidelberg, Berlin, Heidelberg, 146–161.

[62] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An Empirical Characterization of IFTTT: Ecosystem, Usage, and Performance. In *ACM IMC '17*. USA, 398–404.

[63] Microsoft. 2018. Azure IoT. https://azure.microsoft.com/en-us/services/iot-hub/. (June 2018).

[64] Microsoft. 2018. Microsoft IoT for manufacturing. https://www.microsoft.com/en-us/internet-of-things/manufacturing. (2018).

[65] M. Mohsin, Z. Anwar, G. Husari, E. Al-Shaer, and M. A. Rahman. 2016. IoTSAT: A formal framework for security analysis of the Internet of Things (IoT). In *IEEE CNS 16*. USA, 180–188.

[66] M Mohsin, Z. Anwar, Farhat Zaman, and Ehab Al-Shaer. 2017. IoTChecker: A data-driven framework for security analytics of Internet of Things configurations. *Elsevier Computer and Security* 70 (Sept. 2017), 199–223.

[67] M Mohsin, MU Sardar, O. Hasan, and Z. Anwar. 2017. IoTRiskAnalyzer: A Probabilistic Model Checking Based Framework for Formal Risk Analytics of the Internet of Things. *IEEE Acess* 5 (April 2017), 5494–5505.

[68] 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 (Onward! 2017)*. 119–133.

[69] K. Pelechrinis, M. Iliofotou, and S. V. Krishnamurthy. 2011. Denial of Service Attacks in Wireless Networks: The Case of Jammers. *IEEE Communications Surveys Tutorials* 13, 2 (Second 2011), 245–257.

[70] Eyal Ronen, Colin O'Flynny, Adi Shamir, and Achi-Or Weingarten. 2017. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *Proc. IEEE Symposium on Security and Privacy*. San Jose, CA, USA, 195–212.

[71] E. Ronen and A. Shamir. 2016. Extended functionality attacks on IoT devices: The case of smart lights. In *Proc. 2016 IEEE European Symposium on Security and Privacy*. Germany, 3–12.

[72] J. E. Giral Sala, R. Morales Caporal, E. Bonilla Huerta, J. J. Rodriguez Rivas, and J. d. J. Rangel Magdaleno. 2016. A Smart Switch to Connect and Disconnect Electrical Devices at Home by Using Internet. *IEEE Latin America Transactions* 14, 4 (April 2016), 1575–1581.

[73] Samsung. 2018. SmartThings. https://www.smartthings.com/. (June 2018).

[74] Letian Sha, Fu Xiao, Wei Chen, and Jing Sun. 2017. IIoT-SIDefender: Detecting and defense against the sensitive information leakage in industry IoT. *World Wide Web* (Apr 2017), 1–30.

[75] Natarajan Shankar. 2018. *Combining Model Checking and Deduction.* Springer International Publishing, 651–684.

[76] Hocheol Shin, Yunmok Son, Young-Seok Park, Yujin Kwon, and Yongdae Kim. 2016. Sampling Race: Bypassing Timing-Based Analog Active Sensor Spoofing Detection on Analog-Digital Systems. In *USENIX Workshop on Offensive Technologies*.

[77] SmartThings. 2018. SmartThings Community on GitHub. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps. (Sept. 2018).

[78] SmartThings. 2018. SmartThings management page. https://graph-na02-useast1.api.smartthings.com/. (June 2018).

[79] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, Yongdae Kim, et al. 2015. Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors.. In *USENIX Security 15*. 881–896.

[80] Spin. 2018. What is Spin? http://spinroot.com/spin/whatispin.html. (June 2018).

[81] A. Tekeoglu and A. S. Tosun. 2016. A Testbed for Security and Privacy Analysis of IoT Devices. In *IEEE MASS 16*. 343–348.

[82] Bent Thomsen, Kasper Søe Luckow, Lone Leth, and Thomas Bøgholm. 2015. *From Safety Critical Java Programs to Timed Process Models.* Springer International Publishing, 319–338.

[83] 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 *USENIX Security 17*. Vancouver, BC, 361–378.

[84] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *ACM CHI Conference on Human Factors in Computing Systems*. USA, 3227–3231.

[85] Vera. 2018. Smart Home Controller. http://getvera.com/controllers/vera3/. (June 2018).

[86] Amauri Viguera. 2018. More unavailable devices. https://community.smartthings.com/t/more-unavailable-devices/98584. (June 2018).

[87] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and Logging in the Internet of Things. In *NDSS'18*. USA.

[88] Evan Wilkins. 2018. Devices showing up as 'This device is unavailable at the moment'. https://community.smartthings.com/t/devices-showing-up-as-this-device-is-unavailable-at-the-moment/94724. (June 2018).

[89] Judson Wilson, Dan Boneh, Riad S. Wahby, Philip Levis, Henry Corrigan-Gibbs, and Keith Winstein. 2017. Trust but Verify: Auditing the Secure Internet of Things. In *ACM MobiSys '17*. USA, 464–474.

[90] F. Xiao, L. T. Sha, Z. P. Yuan, and R. C. Wang. 2017. VulHunter: A Discovery for unknown Bugs based on Analysis for known patches in Industry Internet of Things. *IEEE Transactions on Emerging Topics in Computing* PP, 99 (2017), 1–1.

[91] Yuchen Yang, Longfei Wu, Guisheng Yin, Lijie Li, and Hongbin Zhao. 2017. A Survey on Security and Privacy Issues in Internet-of-Things. *IEEE Internet of Things Journal* PP (April 2017), 1–10.