SOTERIA: Automated IoT Safety and Security Analysis

Z. Berkay Celik, Patrick McDaniel, and Gang Tan, ***The Pennsylvania State University***

https://www.usenix.org/conference/atc18/presentation/celik

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

SOTERIA: Automated IoT Safety and Security Analysis

Z. Berkay Celik, Patrick McDaniel, and Gang Tan

Department of Computer Science and Engineering
The Pennsylvania State University
{zbc102,mcdaniel,gtan}@cse.psu.edu

Abstract

Broadly defined as the Internet of Things (IoT), the growth of commodity devices that integrate physical processes with digital systems have changed the way we live, play and work. Yet existing IoT platforms cannot evaluate whether an IoT app or environment is safe, secure, and operates correctly. In this paper, we present SOTERIA, a static analysis system for validating whether an IoT app or IoT environment (collection of apps working in concert) adheres to identified safety, security, and functional properties. SOTERIA operates in three phases; (a) translation of platform-specific IoT source code into an intermediate representation (IR), (b) extracting a state model from the IR, (c) applying model checking to verify desired properties. We evaluate SOTERIA on 65 SmartThings market apps through 35 properties and find nine (14%) individual apps violate ten (29%) properties. Further, our study of combined app environments uncovered eleven property violations not exhibited in the isolated apps. Lastly, we demonstrate SOTERIA on MALIOT, a novel open-source test suite containing 17 apps with 20 unique violations.

1 Introduction

The introduction of IoT devices into public and private spaces has changed the way we live. For example, home automation apps supporting smart devices of thermostats, locks, switches, surveillance systems, and Internet-connected appliances change the way we interact with our living spaces. While these systems have been widely embraced, IoT has also raised concerns about the security and safety of digitally augmented lives [18,21,24,34,36]. IoT apps have access to functions that may put the user or environment at risk, e.g., unlock doors when not at home or create unsafe or damaging conditions by turning off the heat in winter. There has been an increasing amount of recent research exploring IoT security and more broadly environmental safety.

One of the oft-discussed criticisms of IoT is that the software and hardware frameworks do not possess the capability to determine if an IoT device or environment is implemented in a way that is safe, secure, and operates correctly. The SmartThings [37], OpenHab [32], Apple's Homekit [1] provide guidelines and policies for regulating security [2, 31, 43], and related markets provide a degree of internal (hand) vetting of the apps prior to distribution [3, 40]. Recent technical community efforts have explored vulnerability analysis within targeted IoT domains [21, 30], while others focused on sensitive data leaks and correctness of IoT apps using a range of analyses [8, 17, 25, 45]. However, tools and algorithms for evaluating general safety and security properties within IoT apps and environments are at this time largely absent.

In this paper, we present SOTERIA[1], a static analysis system for validating whether an IoT app or IoT environment (collection of apps working in concert) adheres to identified safety, security, and functional properties. We exploit existing IoT platforms' sensor-computation-actuator program structures to translate source code of an IoT app into an intermediate representation (IR). Here, the SOTERIA IR models the app's lifecycle–including app entry points, event handler methods, and call graphs. From this, SOTERIA uses the IR to perform efficient static analysis extracting a state model of the app; the state model includes its states and transitions. A set of IoT properties is systematically developed, and model checking is used to check that the app (or collection of apps) conforms to those properties. In this work, we make the following contributions:

- We introduce SOTERIA, a system designed for model checking of IoT apps. SOTERIA automatically extracts a state model from a SmartThings IoT app and applies model checking to find property violations.
- We used SOTERIA on 65 different IoT apps (35 apps from the official SmartThings repository and 30 community-contributed third-party apps from the official SmartThings forum) and reveal how safety and security properties are violated.
- We develop an IoT-specific test corpus MALIOT, an open-source repository of 17 flawed apps that containing an array of safety and security violations.

[1] Soteria is the goddess in Greek mythology preserving from harm.

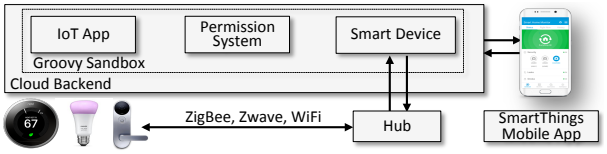

Figure 1: The architecture of SmartThings IoT platform.

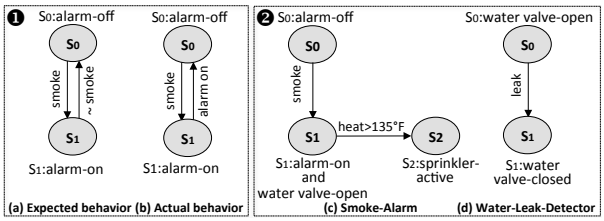

Figure 2: ❶ shows the state models of the expected and actual behavior of the `Smoke-Alarm` app. The app fails because of a bug which halts the alarm when smoke is present. ❷ shows the state models of the `Smoke-Alarm` and `Water-Leak-Detector` apps violating a property when they installed together. The environment fails when the apps interact–the `Water-Leak-Detector` app shuts off water valve when a fire is detected.

2 Background

IoT platforms provide a software stack used to develop applications that monitor and control IoT devices.[2] For example, Fig. 1 shows the three components of the Samsung's SmartThings Platform: a hub, apps, and the cloud backend [40]. The hub controls the communication between connected devices, cloud back-end, and mobile apps. Apps are developed in the Groovy language (a dynamic, object-oriented language) and executed in a Kohsuke sandboxed environment. The cloud backend creates software proxies called SmartDevices that act as a conduit for physical devices, as well as run the apps.

The permission system in SmartThings allows a developer to specify devices and user inputs required for an app at install time. Devices in SmartThings have capabilities (i.e., permissions) that are composed of *actions* and *events*. Actions represent how to control or actuate device states and events are triggered when device states change. SmartThings apps control one or more devices. Apps subscribe to device events or other pre-defined events such as the icon-clicking event, and an event handler is invoked to handle it, which may lead to further events and actions.

Users can install SmartThings apps either from the market or proprietary system via SmartThings Mobile. In the former, publishing an app in the official market requires the developer to submit the source code of the app for review. Official apps appear in the market after the completion of a lengthy review process [40]. In the latter, organizations can develop an app and make it accessible using the Web IDE. These self-published apps do not receive any official review process and are often shared in the SmartThings official community forum [41].

3 Motivation and Assumptions

Example IoT Applications. We introduce three running examples used throughout for exposition and illustration:
The `Smoke-Alarm` app contains a smoke-detection alarm, a water valve (basement), and a light switch (living room). The app sounds the smoke alarm and turns on the light when smoke is detected; when smoke is detected and a heat level is reached, the app opens the water valve to activate fire sprinklers; finally, it turns off the alarm and closes water valve when smoke is clear. Also, it turns on the light switch when the smoke-detector battery is low.
The `Water-Leak-Detector` app detects a water leak with a moisture sensor and shuts off the main water supply valve in order to prevent any further water damage.
The `Thermostat-Energy-Control` app locks the front door and sets the heating thermostat temperature to a pre-defined value when the user-presence mode is changed (e.g., from the user-away mode to the user-home mode or vice versa). When the energy usage is above a pre-defined consumption threshold, it turns off the thermostat switch.

SOTERIA illustrated. Here we informally illustrate SOTERIA analysis through a single and multi-app example.

Consider the `Smoke-Alarm` app. We first model the app's source code as a transition system. Fig. 2(1a) presents the expected behavior of the smoke alarm; the alarm sounds when smoke is detected and not otherwise. The state model starts from an initial state S_0 and transits to state S_1 when smoke is detected. The state transitions are controlled by the output of the smoke sensor: "smoke-detected" (smoke) and "not detected" (~smoke). Fig. 2(1b) is the actual behavior extracted from the open-source implementation of a smoke alarm (that has a bug). We use SOTERIA to validate the above safety property–i.e., "does the alarm always sound when there is smoke?" To perform this analysis SOTERIA encodes the safety property in temporal logic and verifies it on the model with a symbolic model checker. Naturally, the analysis showed a violation; the actual behavior of the app stops the sound moments after the alarm sounds (the state transition from S_1 to S_0). In this case, users may not hear the short or intermittent alarm with potentially disastrous consequences.

Now consider the situation when both `Smoke-Alarm` and `Water-Leak-Detector` apps are co-located in an environment. Fig. 2(2c) and 2(2d) presents expected behavior of the `Smoke-Alarm` and `Water-Leak-Detector` apps, respectively. Here, we use SOTERIA to validate the property "does the sprinkler system activate when there is a fire?". The model checker revealed that there was a safety violation: the `Water-Leak-Detector` app shuts off the water valve and stops fire sprinklers when it detects water release from sprinklers. In this case, the joint behavior of the otherwise-safe apps leaves users are at risk from fire.

[2] While the SOTERIA approach is largely agnostic to the specific IoT software platform, we focus on Samsung's SmartThings Platform [37].

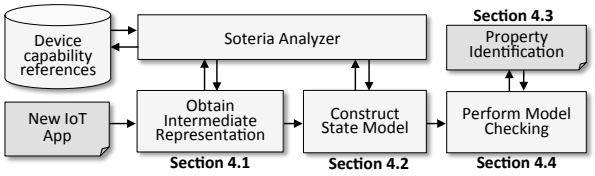

Figure 3: Overview of SOTERIA architecture.

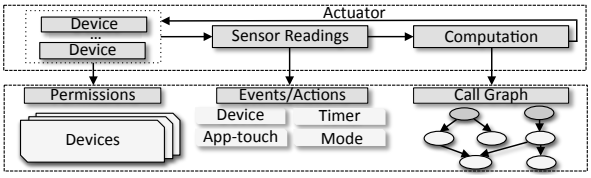

Figure 4: Components of the intermediate representation (IR).

Assumptions and Threat Model. We assume violations can be caused by design flaws or malicious intent. In the latter, the adversary may insert malicious code resulting in insecure or unsafe states, e.g., as seen in attacks on smart light bulbs [36] and home security systems [35]. We do not evaluate adversaries' ability to thwart security measures (e.g., crypto, forged inputs) or explore user privacy, but defer those investigations to future work.

4 SOTERIA

Fig. 3 provides an overview of the four stages of the SOTERIA system analysis. SOTERIA first extracts an intermediate representation (IR) from the source code of an IoT app (Sec. 4.1). The IR is used to model the lifecycle of an app including entry points, event handler methods, and call graphs. Second, SOTERIA uses the IR to extract a state model of the app; the state model includes its states and transitions (Sec. 4.2). Lastly, a set of IoT properties is developed (Sec. 4.3), and model checking is used to check that the app conforms to those properties when running independently or interacting with other apps (Sec. 4.4).

4.1 From Source Code to IR

The first step toward modeling an IoT app is to extract an IR from the app's source code. SOTERIA builds the IR from a framework-agnostic component model, which is comprised of the building blocks of IoT apps, shown in Fig. 4. A broad investigation of existing IoT environments showed that the programming environments could be generalized into three component types: (1) *Permissions* grant capabilities to devices used in an app; (2) *Events/Actions* reflect the association between events and actions: when an event is triggered, an associated action is performed; and (3) *Call graphs* represent the relationship between entry points and functions in an app. The IR has several benefits. First, it allows us to precisely model the app lifecycle as described above. Second, it is used to abstract away parts of the code that are not relevant to property analysis, e.g., `definition` blocks that specify app meta-data and `logger` logging code. Third, it allows efficiently extract the states and state transitions from the implementation (see below). Presented in Fig. 5, we use the `Smoke-Alarm` app to illustrate the use of the IR.

Permissions. When a SmartThings app gets installed or updated, the permissions for devices and user inputs are displayed to the user (and explicitly accepted). The permissions are read-only, and app logic is implemented using the permissions. SOTERIA visits permissions of an app to extract its devices and user inputs. Turning to the IR in Fig. 5, the permission block (lines 1–7) defines: (1) the devices; a smoke detector, a switch, an alarm, a valve, and a battery in the smoke detector; and (2) user input: "thrshld" is used to determine whether the battery level of the smoke detector is low. For each permission, the IR declares a triple following keyword "input". For a device, the triple associates an identifier for the device, called the *device handle*, to its platform-specific device name in order to determine the interface that the device may access. For instance, an app may associate identifier `the_switch` with a switch device, which is in either the "off" or the "on" state. For a user input, the triple in the IR contains the variable name storing the user input, its type, and a tag showing the kind of input such as the user-defined input. In this way, we obtain a complete list of devices and user inputs that an app might access.

Events/Actions. Similar to mobile applications, an IoT app does not have a main method due to its event-driven nature. Apps implicitly define entry points by subscribing events. The event/actions block in an IR is built by analyzing how an app subscribes to events. Each line in the block includes three pieces of information: a device handle, a device event to be subscribed, and an event handler method to be invoked when that event occurs (lines 9–10). Event handler methods are commonly used to take device actions. Therefore, an app may define multiple entry points by subscribing multiple events of a device or devices. Turning to our example, the event of "smoke-detected" state change is associated with an event handler method named `h1()` and the event of "battery" level state change with `h2()`. We also found that events are not limited to device events; we call these *abstract events*: (1) *Timer events*; event-handlers are scheduled to take actions within a particular time or at pre-defined times (e.g., an event-handler is invoked to take actions after a given number of minutes has elapsed or at specific times such as sunset); (2) *App touch events*; for example, some action can be performed when the user clicks on a button in an app; (3) what actions get generated may also depend on *mode events*, which are behavior filters automating device actions. For instance, an app running in "home" mode turns off the alarm and turns on the alarm when it is in the "away" mode. SOTERIA examines all event subscriptions and finds their corresponding event-handler methods; it creates a dummy main method for each entry point.

```
1:  // Permissions block
2:  input (smoke_detector, smokeDetector, type:device)
3:  input (the_switch, switch, type:device)
4:  input (the_alarm, alarm, type:device)
5:  input (the_valve, valve, type:device)
6:  input (the_battery, battery, type:device)
7:  input (thrshld, number, type:user_defined)
8:  // Events/Actions block
9:  subscribe(smoke_detector, "smoke", h1)
10: subscribe(the_battery, "battery", h2)
11: // Entry point
12: h1(){
13:   if(evt.value == "detected") {
14:       the_alarm.siren()
15:       the_valve.open()
16:   }
17:   if(evt.value=="clear"){
18:       the_alarm.off()
19:       the_valve.close()
20:   }
21: }
22: // Entry point
23: h2(){
24:   batteryLevel = p()
25:   if(batteryLevel < thrshld){
26:       the_switch.on()
27:   }
28: }
29: p(){
30:   return the_battery.currentValue("battery")
31: }
```

Figure 5: The IR of `Smoke-Alarm` app constructed with SOTERIA.

Asynchronously Executing Events. While each event corresponds to a unique event-handler, the sequence of event handler invocations cannot be decided in advance when multiple events happen at the same time. For instance, in our example, there could be a third subscription in the event/actions block that subscribes to the switch-off event to invoke another event handler method. We consider eventually consistent events, which means any time an event handler is invoked, it will finish execution before another event is handled, and the events are handled in the order they are received by an edge device (e.g., a hub). We base our implementation on path-sensitive analysis that analyzes an app's event handlers, which can run in arbitrary sequential order. This analysis is enabled by constructing a separate call graph for each entry point.

Call Graphs. We create a call graph for each entry point that defines an event handler method. Turning to the IR in Fig. 5, we define call graphs for two entry points `h1()` and `h2()` (line 12 and 23). `h1()` invokes `p()` to get the current battery level of the smoke detector. Addressed below, note that these initial graphs are sometimes incomplete because of dynamic method invocations (reflection).

4.2 State Model Extraction

SOTERIA next extracts a state model from the IR model.

Definition of State Models. An IoT app manages one or more devices. Each device has a set of attributes, which are the states of the device. For instance, in the `Water-Leak-Detector` app, the water sensor has a boolean-typed attribute, whose value signals the "water-detected" or "water-undetected" status. Hence, we naturally model the states in the model from the values of device attributes. IoT apps are event-driven: events such as state changes or user input trigger event handlers, which can in turn change device attributes by invoking device actions. Therefore, by analyzing an IoT app's code, we can add state transitions and label them with events that trigger the transitions (changes to attribute values).

More formally, we define the state model of an IoT app as a triple (Q, Σ, δ), where Q is a set of states, Σ is a set of transition labels, and δ is a state-transition function that represents labeled transitions between states. We restrict our attention to deterministic state models, as we believe this is a condition for safe operation of IoT devices. In fact, after a state model extracted, SOTERIA reports nondeterministic state models as a safety violation.

Challenges in Extracting State Models. Although it may appear at first glance to be straightforward, extracting state models is fraught with challenges. First, extraction faces state-explosion problem. For instance, a thermostat device may have an integer-discrete or continuous temperature attribute would lead to many different states–adding a state for every possible value in such cases would result in state explosion. To address this, SOTERIA implements a form of property abstraction that collapses states by aggregating attribute values (see Sec. 4.2.1).

A second challenge concerns with model precision. A state model is an abstraction of an app's logic and necessarily has to over-approximate. A sound over-approximation can cause false positives during model checking. One such approximation that caused false positives for an earlier version of SOTERIA was that the labels on transitions were only events and thus too coarse-grained. It turns out that many IoT apps change device states *conditionally*; for example, an app may turn off a switch when the energy consumption is above some threshold and turn on the switch when the energy consumption is below another threshold. For precision, the current version of SOTERIA performs a path-sensitive analysis to extract predicates that guard state changes and adds the predicates as part of state-transition labels. We detail how state transitions are constructed in Sec. 4.2.2.

Finally, the SmartThings platform has a number of idiosyncrasies that SOTERIA's model extraction must address. For instance, SmartThings apps are written in Groovy, a dynamically typed language that supports call by reflection; as another example, SmartThings apps can use special objects for persistent data storage. We will discuss how these issues are addressed in Sec. 4.2.3.

4.2.1 Extracting States

As discussed, states in an app's state model should represent device attribute values. Turning to the `Water-Leak-Detector` app, this app has two devices: a water sensor and a valve, both of which are represented as Boolean attributes. Therefore, the app's state model has four states: water-detected and valve-closed; water-detected and valve-open; water-undetected and valve-closed; water-undetected and valve-open. The number of possible states of an app is determined by the Cartesian product of the attributes of its device. For instance, an app implementing two devices that have `A` and `B` attributes

Algorithm 1: Computing dependence from device's code

```
Input  : ICFG: Inter-procedural control flow graph
Input  : A numerical-valued attribute
Output : Dependence relation dep
1  worklist ← ∅; done ← ∅; dep ← ∅
2  for an id in a device action call that sets the attribute at node n do
3  |  worklist ← worklist ∪ {(n: id)}
4  end
5  while worklist is not empty do
6  |  (n: id) ← worklist.pop()
7  |  done ← done ∪ {(n: id)}
   |  /* a def of (n: id) at node n' means a path from n'
   |     to n exists and on the path there is no other
   |     assignment to id */
8  |  for a def of (n: id) at node n' of form id = e and e has only a
   |    single identifier id' do
9  |  |  worklist ← worklist ∪ ({(n': id')} \ done)
10 |  |  dep ← dep ∪ {(n: id, n': id') }
11 |  end
12 end
```

should have states of all pairs $(\mathtt{a}, \mathtt{b})$, where $\mathtt{a} \in \mathtt{A}$ and $\mathtt{b} \in \mathtt{B}$.

Identification of Device Attributes. An IoT platform supports many physical devices. Sound model extraction requires identifying the complete set of device attributes. Prior work has used binary instrumentation to observe the runtime behavior of apps to infer the set of device operations used with a particular state [16]. However, this is not an option on some IoT platforms such as SmartThings where app execution is inside proprietary back-ends. Another option would be to use the built-in capability files, which come with devices. The capability file for a device identifies device permissions but not attribute values–and thus do not provide enough information for analysis.

Thus, to identify device attributes, SOTERIA uses platform-specific device handlers. A device handler is the representation of a physical device in an IoT platform and is responsible for communication between the device and the IoT platform (it is similar to a traditional device driver in an OS). For instance, the switch device handlers in SmartThings [44] and OpenHAB [32] IoT platforms support the "switch on" and "switch off" attributes, and allow apps to incorporate different kinds of switches in the same way. We developed a crawler script, which visits the `status` (for attributes) and `reply` (for actions) code blocks of SmartThings device handlers found in its official GitHub repository [44] and determines a complete set of attributes and actions for devices. We then created our own platform-specific *device capability reference file*, which includes for each device its complete set of attributes and actions. SOTERIA then uses this file to identify all attributes for those devices used in an app.

Numerical-Valued Device Attributes. Noted above, IoT devices may have attributes with integer or continuous values leading to many states. Returning to the previous `Thermostat-Energy-Control` app, a thermostat with 45 values (50-95°F) and a power meter with 100 energy levels would lead to (clearly intractable) 4.5K states if a state is added for each combination of attribute values.

SOTERIA performs property abstraction [5] to reduce the state space. It first performs dependence analysis on an app's source code to identify possible sources for numerical-valued attributes, and then prunes sources using path- and context-sensitivity; the remaining sources are used to construct states in the state model. The SOTERIA dependence analysis is presented in Algorithm 1 as a worklist-based algorithm. The goal of the algorithm is to identify a set of possible sources that a numerical-valued attribute can take during the execution of an app. The worklist is initialized with identifiers that are used in the arguments of device action calls that change the attribute. The worklist also labels an identifier with node information to uniquely identify the use of an identifier, because the same identifier can be used in multiple locations. The algorithm then takes an entry (n, id) from the worklist and finds a definition for *id* according to the ICFG; if the right-hand side of the definition has a single identifier, the identifier is added to the worklist;[3] furthermore, the dependence between *id* and the right-hand side identifier is recorded in dep. For ease of presentation, the algorithm treats parameter passing as inter-procedural definitions.

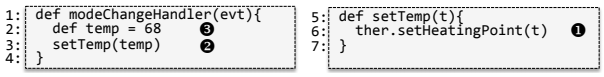

Figure 6: Property abstraction under backward flow analysis.

The dependence analysis is a form of backward taint analysis and produces a set of sources that can affect a change to a numerical-valued attribute. For those sources, SOTERIA makes them separate states in the state model and adds another state representing the rest of values.

To illustrate, we use a code block of the `Thermostat-Energy-Control` app as an example, shown in Fig. 6. There is a device action call that sets the thermostat to `t` at ❶; so the worklist is initialized to be (6:`t`); for presentation, we use line numbers instead of node numbers to label identifiers. Then, because of the function call at ❷, (3:`temp`) is added to the worklist and the dependence (6:`t`, 3:`temp`) is recorded in dep. With this dependence analysis, SOTERIA computes that the value for `t` has to be 68°F since `temp` is initialized to be a constant value at ❸. Therefore, the state model has two states for the thermostat: a state when the temperature is equal to 68°F, and a state when the temperature is not 68°F; thus, the state space for temperature values is reduced from 45 to 2.

The backward dependence analysis also produces the dep relation, through which SOTERIA constructs paths from identifier initialization points to where device changes happen. For the example in Fig. 6, it produces the path

[3] We found that SmartThings IoT apps most often propagates a developer-defined constant or a user input to places that change device attributes. Occasionally, simple arithmetic is performed; for example, the user input is stored in y, followed by $x = y + 10$, followed by a device attribute change using x. In theory, an IoT app could perform operations like $x = y + z$, where both y and z are user input or defined to be constants; however, we have not encountered this in our evaluation.

❸→❷→❶. Some produced paths by dependence analysis, however, can be infeasible paths. As an optimization, SOTERIA prunes infeasible paths using path- and context-sensitivity. For a path calculated in dependence analysis, it collects the predicates at conditional branches and checks whether the conjunction of those predicates (i.e., the path condition) is always false; if so, the path is infeasible and discarded. This is similar to how symbolic execution prunes paths using path conditions. For instance, if a path goes through two conditional branches and the first branch evaluates $x > 1$ to true and the second evaluates $x < 0$ to true, then it is an infeasible path. SOTERIA does not use a general SMT solver to check path conditions. We found that the predicates used in IoT apps are extremely simple in the form of comparisons between variables and constants (such as $x = c$ and $x > c$); thus, SOTERIA implemented its simple custom checker for path conditions. Furthermore, SOTERIA throws away paths that do not match function calls and returns (using depth-one call-site sensitivity [39]). At the end of the pruning process, we get a set of feasible paths that propagate sources defined by the developer or by user input to device action calls that change the numerical-valued attribute; and then those sources are used to define the states in the model.

4.2.2 Extracting State Transitions

If an event handler changes a device's attributes by actuating the device, it leads to a state transition. By statically analyzing event handlers, SOTERIA computes state transitions and labels them with events. When a water-detected event is generated in the `Water-Leak-Detector` app a handler method closes the valve; by analyzing the handler method, SOTERIA adds a transition with the water-detected event label from state "water-undetected and valve-open" to "water-detected and valve-closed" state.

Labeling Transitions with Predicates. Many device state changes happen in conditional branches; as a result, those state changes occur only when the predicates in the conditional branches hold. To illustrate, consider the source code in Fig. 7 abstracted from the `Thermostat-Energy-Control` app. The app has a conditional branch turning off the switch when energy usage is above a consumption threshold (`above=50`); it turns on the switch when it is below the threshold (`below=5`).

SOTERIA implements a path-sensitive analysis to capture state transitions and predicates that guard transitions. Particularly, it uses *symbolic execution* to perform path exploration on source code and accumulates path conditions during exploration. In detail, it starts the analysis at the entry of an event handler with respect to some initial state, say S_0. Then it performs forward symbolic execution along all paths, and also smartly merges paths following the ESP algorithm [13] (as a way of avoiding path explosion). For a conditional branch with condition b, it evaluates both paths and labels the true path with b and the false path with $\neg b$. If the end states for the true and false branches are the same, then the two paths are merged [13]. On the other hand, if the end states are different for the two paths, they are kept separate for further symbolic execution. SOTERIA throws away infeasible paths in a way similar to that used during property abstraction. At the end of symbolic execution, SOTERIA obtains the set of paths, their end states, and path conditions. For each path, a state transition from the initial state to the end state is added to the state model, and the transition is labeled by the event triggering the event handler and path condition.

```
1:  // Permission block
2:  Input(switch, switch)
3:  Input(power_meter, powerMeter)
4:  // Event/Action block
5:  subscribe(power_meter, power, handler)
6:  // Entry point
7:  handler(){
8:     above = 50
9:     below = 5
10:    power_val = get_power()
11:    if (power_val > above){
12:       switch.off()
13:    }
14:    if (power_val < below){
15:       switch.on()
16:    }
17: }
18: get_power(){
19:    latest_power = power_meter.currentValue("power")
20:    return latest_power
21: }
```

Figure 7: The impact of predicates on state transitions in the `Thermostat-Energy-Control` app.

We use the `Thermostat-Energy-Control` app with the initial state of "switch-on" as an illustration of this exploration. SOTERIA explores all paths, and there are two feasible paths at the end, with `currentValue("power")>50` as the path condition of the path that turns off the switch, and `currentValue("power")<5` as the path condition of the path that turns on the switch.

In addition, SOTERIA also tracks the sources of components in predicates that guard state transitions. For predicate `currentValue("power")>50` in the previous example, `currentValue("power")` is obtained from a device state and therefore is labeled as "device-state", while 50 is hardcoded by the developer and therefore is labeled as "developer-defined". In some cases, users can also define part of predicates at install time of an app. For instance, if the threshold value were entered by a user, then SOTERIA would label it as "user-defined". Labeling sources in predicates is useful for precisely stating properties used in model checking. For example, one property says that the alarm must siren when the main door is left open longer than a threshold entered by the user. In this case, there is no property violation if the threshold is not hard-coded into the app by the developer. We detail this in Sec. 4.3.

4.2.3 SmartThings Idiosyncrasies

Platform-specific Interfaces. The SmartThings platform implements a variety of programmer interfaces for an app to obtain device attribute values (for the same value). For instance, the temperature value of a thermostat can be read through the `currentState` or the `currentTemperature` interface (see Listing 1 (lines 1–8). Additionally, we found

Listing 1: Sample code blocks for SmartThings Idiosyncrasies

```
1  /* A code block of an app using platform-specific interfaces */
2  subscribe(theMotion, "motion", motionHandler)
3  subscribe(theThermostat, "thermostat", thermostatHandler)
4  // different interfaces to get device attribute values
5  def thermostatHandler() {
6      def tempAttr = theThermostat.currentState("temperature")
7      def tempAttr2 = theThermostat.currentThermostat
8  }
9  // transitions without explicit event subscriptions
10 def motionHandler(evt) {
11     if (evt.value == "active") { ... }
12     else if (evt.value == "inactive") {...}
13 }
14 /* A code block of an app using call by reflection */
15 //initial state = S0
16 def getMethod(){
17   httpGet("http://url"){ resp ->
18       if(resp.status == 200){
19           name = resp.data.toString()
20       }
21   }
22   "$name"() // dynamic method invocation
23 }
24 // check state transition from S0 to next state in both methods
25 def foo() {...}
26 def bar() {...}
```

that some apps subscribe to all device events instead of specific device events; for example, the `subscribe` interface in Listing 1 (lines 9–13) is used to subscribe to all events of a motion sensor. The event handler then gets an event value as an argument that describes what event it is. We extract precise state models by parsing the event values passed in these interfaces and adding state transitions through those interfaces.

Call by Reflection. The Groovy language supports programming by reflection (using the `GString` feature) [44], which allows a method to be invoked by providing its name as a string. For instance, a Groovy method `foo()` can be invoked by declaring a string `name="foo"` requested from an external server via the `httpGet()` interface and thereafter called by reflection through `$name` (see Listing 1 (lines 14–26)). To handle calls by reflection, SOTERIA's call graph construction adds all methods in an app as possible call targets, as a safe over-approximation. For the example in Listing 1, SOTERIA adds both `foo()` and `bar()` to the targets of the call; then it searches for state changes in each method and extracts state transitions.

4.3 Identifying IoT Properties

As many have found in the security and safety communities, identifying the correct set of properties to validate for a given artifact is often a daunting task. In this work and as described below, we use established techniques adapted from other domains to systematically identify a set of properties that exercise SOTERIA and are representative of the real world needs of users and environments. That being said, we acknowledge in practice that properties are often more contextual and the methods to find them are often more art than science. Hence, we argue that many environments will need to tailor their property discovery process to their specific security and safety needs.

We refer to a property as a system artifact that can be formally expressed via specification and validated on the application model. We extend the use/misuse case requirements engineering [29, 33, 38, 47] to identify IoT

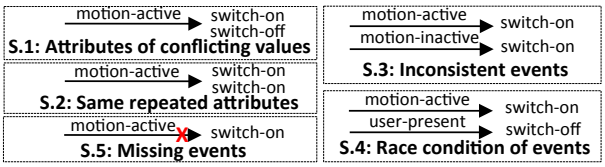

Figure 8: Illustration of general properties (S.1-S.5).

properties. This approach derives requirements (properties) by evaluating the connections between 1) *assets* are artifacts that someone places value upon, e.g., a garage door, 2) *functional requirements* define how a system is supposed to operate in normal environment, e.g., when a garage door button is opened, the door opens, and 3) *functional constraints* restrict the use or operation of assets, e.g., the door must open only when an authorized garage-door opener device requests it. We used use/misuse case requirements engineering as a property discovery process on the IoT apps used in our evaluation (See Section 6) and identified 5 general properties (S.1-S.5, see Fig. 8) and 30 application-specific properties (P.1-P.30, see Table 1).

General Properties. General properties are constraints on state models that are independent of an app's semantics–intuitively, these are states and transitions that should never occur regardless of the app domain. We develop the properties based on the constraints on states and state transitions. To illustrate, property S.1 states that a handler must not change an attribute to conflicting values on the same control-flow path, e.g., the motion-active handler must not turn on and turn off a switch in the same branch of the handler. More subtly, property S.4 states that two or more non-complementary handlers must not change an attribute to conflicting values, e.g., a user-present handler turns on the switch while a timer turns off the switch–leading to a potential race condition.

App-specific Properties. App-specific properties are developed according to use cases of one or more devices–here we take a device-centric approach. For instance, P.1 says that the door must always be locked when the user is not at home (thus involving the smart door and presence detector). Similarly, P.30, states that the water valve must be shut off when there is a water leak (thus involving the water valve and moisture sensor). We evaluated all apps using this approach, but defer discussion to the extended paper. We check the app against a property if all of the devices in the property are included in the app.

4.4 Validating Properties

Validation begins by defining a temporal formula for each property to be verified. Thereafter, SOTERIA uses a general purpose model checker to validate the property with respect to the generated model of the target app (see next section for details). What the user does with a discovered violation is outside the scope of SOTERIA. However, in most cases, we expect that the results will be recorded

ID	Property Description
P.1	The door must always be locked when the user is not home.
P.10	The alarm must always go off when there is smoke.
P.12	The light must be off when the user is not home.
P.13	The devices (e.g., coffee machine, crock-pot) must always be on at the time set by the user.
P.14	The refrigerator and security system must always be on.
P.17	The AC and heater must not be on at the same time.
P.22	The battery of devices must not be below a specified threshold.
P.28	The sound system must not play music during the sleeping mode.
P.29	The flood sensor must always notify the user when there is water.
P.30	The water valve must be closed if a leak is detected.

Table 1: Examples of application-specific properties. A complete list of properties is available in the extended paper [9].

and the code hand-investigated to determine the cause(s) of the violation. If the violation is not acceptable for the domain or environment, the app can be rejected (from the market) or modified (by the developer) as needs dictate.

Validation of properties in multi-app environments is more challenging. Apps often interact through a common device or some common abstract event (such as the home or away modes). For illustration, consider two apps (`App1` and `App2`) co-resident with the `Smoke-Alarm` and `Thermostat-Energy-Control` apps in a multi-device environment. `App1` changes the mode from away to home when the light switch is turned on, and `App2` turns off a light switch when the smoke is detected, as follows:

`Smoke-Alarm`: switch-off $\xrightarrow{\text{smoke-detected}}$ switch-on
`App1`: away-mode $\xrightarrow{\text{switch-on}}$ home-mode
`Thermostat-Energy-Control`: door-unlocked $\xrightarrow{\text{home-mode}}$ door-locked
`App2`: switch-on $\xrightarrow{\text{smoke-detected}}$ switch-off

The `Smoke-Alarm` app interacts with `App1` through the switch, and interacts with `App2` through the smoke detector and switch. The `Thermostat-Energy-Control` app interacts with `App2` through the mode-change event.

To check general and app-specific properties in the setting of multiple apps, SOTERIA builds a state model that is the union of the apps' state models. The resulting state model $\mathtt{G}'$ represents the complete behavior when running the multiple apps together. The union algorithm is presented in Algorithm 2. SOTERIA first creates an empty-transition state model $\mathtt{G}'$ whose states are the Cartesian product of the states in the input apps (line 1); note that since the input apps' states encode device attributes, the Cartesian product should remove attributes of duplicate devices (i.e., those devices that appear in multiple apps). For instance, if we consider `Smoke-Alarm` and `App1`, $\mathtt{G}'$ should have four states, and each state encodes a pair of switch and mode attributes. The algorithm then iterates through all apps' transitions and adds appropriate transitions to the union model $\mathtt{G}'$. SOTERIA's union algorithm is a modification of the multiple-graph union algorithm of igraph library [22], based on a set of constraints on transitions and states. It has a complexity of $\mathtt{O}(|\mathtt{V}| + |\mathtt{E}|)$, $|\mathtt{V}|$ and $|\mathtt{E}|$ is the number of vertices and edges in $\mathtt{G}'$.

With the union state model created, SOTERIA then performs model checking on the union model concerning properties we discussed earlier. As an example, SOTERIA reports that, when `Smoke-Alarm` and `App2` are used

Algorithm 2: Creating the union of apps' state models

Input : $\mathtt{G} = \{\mathtt{G_i}\}_{\mathtt{i=1}}^{\mathtt{n}}$: State models of n apps
Output : $\mathtt{G}'$ is the union of $\{\mathtt{G_i}\}_{\mathtt{i=1}}^{\mathtt{n}}$
`/* Initialize G' */`
1 states($\mathtt{G}'$) $\leftarrow \{v \mid v$ is a tuple of attribute values in $\mathtt{G}\}$
`/* Construct union of apps' state models */`
2 **for** *i ∈ (1: n)* **do**
3 **forall** *states* $v \in \mathtt{G}_i$ **do**
4 **forall** *transitions* $e = v \xrightarrow{l} u \in \mathtt{G}_i$ **do**
5 $\mathtt{V}'$ is a subset of states in $\mathtt{G}'$ that contain v
6 $\mathtt{U}'$ is a subset of states in $\mathtt{G}'$ that contain u
7 **forall** $v' \in \mathtt{V}'$ and $u' \in \mathtt{U}'$ **do**
8 add $e' = v' \xrightarrow{l} u'$ to $\mathtt{G}'$ and label the edge with i
9 **end**
10 **end**
11 **end**
12 **end**

together, there is a property violation of `S.1`: the smoke-detected event would make the `Smoke-Alarm` app turn on the switch, while it would also make `App2` to turn off the switch. As another example, when `Smoke-Alarm`, `App1` and `Thermostat-Energy-Control` are used together, there is a misuse case that violates property `P.3`: the door would be locked when there is smoke at home. The property violation is demonstrated as follows:

switch-off $\xrightarrow{\text{smoke-detected}}$ **switch-on** $\xrightarrow{\text{switch-on}}$ **home-mode** $\xrightarrow{\text{home-mode}}$ **door-locked**

`P.3` is violated because switch-on attribute in the `Smoke-Alarm` app is used by `App1`, which changes the mode from away to home. The mode change then triggers locking the door in `Thermostat-Energy-Control`.

5 Implementation

IR and State Model Construction. Constructing an IR from the source code requires, among other things, the building of the app's ICFG. Here the SOTERIA IR-building algorithm directly works on the Abstract Syntax Tree (AST) representation of Groovy source code. The Groovy compiler supports customizing the compilation via compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [27]). SOTERIA visits AST nodes at the compiler's semantic analysis phase where the Groovy compiler performs consistency and validity checks on the AST. Our implementation uses an `ASTTransformation` to hook into the compiler, `GroovyClassVisitor` to extract the entry points and the structure of the analyzed app, and `GroovyCodeVisitor` to extract method calls and expressions inside AST nodes. Here we AST visitors to analyze expressions and statements to construct the IR and model.

SOTERIA uses AST visitors for state model construction as well. We extend the `ASTBrowser` class implemented in the Groovy Swing console, which allows users to enter and run Groovy scripts [19]. The implementation hooks into the IR of an app in the console and dumps information to the `TreeNodeMaker` class; the information includes an AST node's children, parent, and all properties built during compilation. This includes the resolved classes,

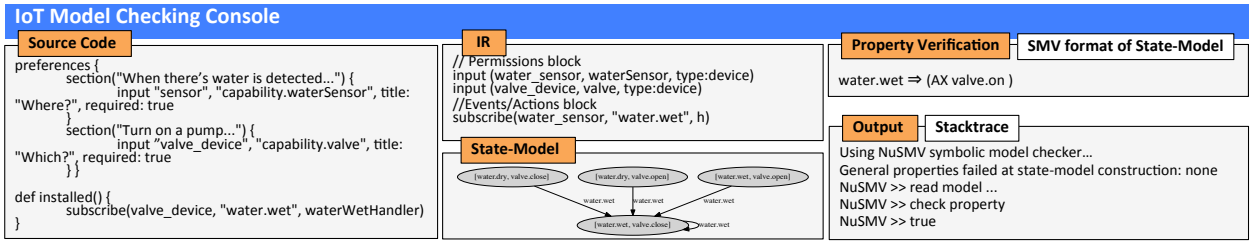

Figure 9: Our SOTERIA framework designed for IoT apps. The left region is the analysis frame; the middle region contains the IR and visual representation of the state model of an example IoT app, and the right region shows the output for a property violation.

static imports, the scope of variables, method calls, interfaces accessed in an app. We then use Groovy visitors to traverse the IR's ICFG and extract the state model.

Model Checking with NuSMV. We translate the state model of an IoT app into a Kripke structure [12]. A Kripke structure is an equivalent temporal structure of a state model and increases readability. We create a visual representation of a state model using open-source graph visualization software GraphViz [14]. We use the open-source symbolic model checker NuSMV [10] for its reliability and maturity. We express properties with temporal logic formulas [11]. NuSMV either confirms a property holds or presents a counter-example showing why the property is false. To address state explosion in apps that control a large number of devices or that have complex control logic, we use NuSMV options that combine binary decision diagrams (BDDs)-based model checking with SAT-based model checking [6]. This was successfully applied to verify models having more than 10^{20} states and hundreds of state variables [7].

Output of SOTERIA. Fig. 9 presents SOTERIA's analysis result on a sample app. It builds the app IR, extracts the state model, and displays a visual representation of the state model. For each property, SOTERIA either shows the property holds or presents a counter-example.

6 Evaluation

As a means of evaluating the SOTERIA framework, we performed an analysis on two large-scale data-sets–one market based and one synthetic. In these studies, we sought to validate the correctness, completeness, and performance of property analysis on the target datasets. We performed our experiments on a laptop computer with a 2.6GHz 2-core Intel i5 processor and 8GB RAM, using Oracle's Java Runtime version 1.8 (64 bit) in its default settings. We use NuSMV 2.6.0 for model checking and Graphviz 2.36 for visualization of a state model.

Datasets. For the market dataset, we obtained 35 official (vetted) apps (`01-035`) from the SmartThings GitHub repository [43] and 30 community-contributed third-party (non-vetted) apps (`TP1-TP30`) from the official SmartThings community forum [41] in late 2017 (see Table 2). The 65 apps were selected to include various devices and

	Nr.	Unique Devices	Avg/Max States‡	Avg/Max LOC	Func.†
Official	35	14	36/180	220/2633	All
Third-party	30	18	32/96	246/1360	All

‡ This is after applying SOTERIA's state-reduction algorithms.
† The apps cover all spectrum of functionality, including security and safety, green living, convenience, home automation, and personal care. We determined an app's functionality by checking definition blocks in its source code.

Table 2: Description of analyzed official and third-party apps.

functionality that encompass diverse real-life use-cases.

For the synthetic dataset, we introduce MALIOT [23], an open source repository containing flawed IoT apps. Inspired by other security-relevant app test suites [4, 15, 28], MALIOT includes 17 hand-crafted flawed SmartThings apps (`App1-App17`) containing property violations in an individual app and multi-app environments. 14 apps have a single property violation, and three have multiple property violations, with a total of 20 property violations. The apps include various devices covering diverse real-life use-cases. The accurate identification of property violations requires program analysis including multiple entry points, numerical-valued device attributes, and transitions guarded by predicates. Each app in MALIOT also comes with ground truth of what properties are violated; this is provided in a comment block in the app's source code.

6.1 Market App Evaluation

We first report results of the verification of general (`S.1-S.5`) and app-specific (`P.1-P.30`) properties. The properties are checked for each app and collections of apps working in concert. SOTERIA flagged that nine individual apps and three multi-app groups violate at least one property. We manually checked the property violations and verified that all reported ones are true positives. The manual checking process was straightforward to perform since SmartThings apps are relatively small.

Individual App Analysis. Table 3 the results of our analysis on single apps. SOTERIA flagged one third-party app violating multiple properties, eight third-party apps violating a single property. None of the official apps were flagged as violating properties; we believe this is because of the strict manual vetting enforced on official apps, which takes a couple of months [40]. For third-party apps, we manually verified that all reported property violations are indeed problems with the implementation. For exam-

ID	Violation Description	Violated Pr.
TP1	The music player is turned on when user is not at home.	P.13
TP2	The switch turns on and blinks lights when no user is present.	P.12
TP3	The location is changed to the different modes when the switch is turned off and when the motion is inactive.	S.4
TP4	The flood sensor sounds alarm when there is no water.	P.29
TP5	The music player turns on when the user is sleeping.	P.28
TP6	The lights turn on and turn off when nobody is at home.	P.13 and S.1
TP7	The lights turn on and turn off when the icon of the app is tapped.	S.1
TP8	The door is unlocked on sunrise and locked on sunset.	P.1
TP9	The door is locked multiple times after it is closed.	S.2

Table 3: SOTERIA's results on individual apps.

ple, a property violation happens in an app (TP6) that turns off and on a light switch when there is nobody at home; another app (TP9) unlocks the door at sunset and locks the door at sunrise–and unintended action.

To assess whether the property violations are real bugs in analyzed apps, we opened a thread in official SmartThings community forum and asked users whether the functionality of the apps confirms their expectations [42]. We got eight answers from the users that are smart home enthusiasts. These apps may have subtle and surprising uses under the right conditions: a user for TP4, said that he used his flood sensor to let him know when there is no water so that he can add water to the trees during Christmas; another user stated that TP6 might simulate occupancy of his home at night by randomly turning on/off lights when nobody is home. To guard against malicious code, those users stated that they attempted to read and understand the source code of the apps before they installed them. However, since regular users cannot be expected to read and check the source code of apps manually, SOTERIA addresses this problem by analyzing apps and presenting their potential property violations to users, which allows them to determine whether a violation is actually harmful.

Multi-App Analysis. We found that multiple apps working in concert can lead to unsafe and undesired device states. SOTERIA flagged three group of apps violating multiple properties. We examined 28 groups and found three groups that have 17 apps violate 11 properties. Table 4 shows the app groups, events, and device attributes that constitute violations, and violated properties. In the following discussion, we will use app group IDs (G.1-G.3) in Table 4. Each group includes a set of apps that a user may install together and authorize to use the same devices.

In G.1, 03 and 04 violate S.1 by setting the switch attribute to conflicting values when the contact sensor is open; there is a similar violation between 04, 08 and TP12 when the contact sensor is closed. 08 and TP12 violates S.2 by turning on the switch multiple times with the "contact sensor close" event. In addition, 03 and 04 violate S.3 by turning on the switch with complement events of "contact sensor close" and "contact sensor open". In G.2, 09, 016, and TP3 violates S.2 by turning on the switch multiple times with the "motion active" event. Additionally, the interaction between 014, 09, 016 and TP3 violates S.4 by invoking "switch on" and "switch off" actions with different device events ("contact sensor open" and "motion active"). There is a similar violation between 014 and TP2

Gr. ID	App ID	Events/Actions	Violated Pr.
G.1	03	contact sensor open → switch on	S.1, S.2, S.3
	04	contact sensor open → switch off	
		contact sensor close → switch on	
	08, TP12	contact sensor close → switch off	
G.2	014	contact sensor open → switch off	S.2, S.4
	09, 016, TP3	motion active → switch on	
	TP2	app touch → switch on	
G.3	07, TP3	switch off → change location mode	P.12, P.13, P.14, P.17, S.1, S.2
		motion inactive → change location mode	
	030, TP21	location mode change → switch off	
	031, TP22	location mode change → switch on	
	012, TP19	location mode change → set thermostat heating	
		location mode change → set thermostat cooling	

Table 4: SOTERIA's results in multi-app environments.

("contact sensor open" and "app touch"). These events may occur at the same time, which leads to a race condition. In G.3, similar to the other groups, S.1 and S.2 are violated. In addition, multiple app-specific properties are violated. 07 and TP3 change the location mode when the switch is turned off and also when motion is inactive. 030 and TP21 turn off the switch of a set of devices including a security system, smoke detector, and heater when the location is changed; 031 and TP22 turns on devices such as TV, coffee machine, A/C, and heater when the location is changed; both cases violate multiple properties (P.12, P.13, P.14 and P.17) and cause security and safety risks for users. Lastly, 012 and TP19 sets the thermostat to user settings when the switched is turned off and when the motion is inactive. These result in an unauthorized control of thermostat heating and cooling temperature values.

6.2 MALIOT Evaluation

Our analysis of SOTERIA on MALIOT showed that it correctly identified the 17 of the 20 unique property violations in the 17 apps. SOTERIA produces a false warning for an app that uses call by reflection (App5). This app invokes a method via a string. It over-approximates the call graph by allowing the method invocation to target all methods in the app. Since one of the methods turns off the alarm when there is smoke, SOTERIA reports a violation. However, it turns out that the reflective call in this app would not call the property-violating method. Note this pattern did not appear in the 65 real IoT apps we discussed earlier. Additionally, SOTERIA did not report a violation for an app that leaks sensitive data (App10) and for an app that implements dynamic device permissions (App11) as they are outside the scope of SOTERIA analysis.

6.3 MicroBenchmarks

State Reduction Efficacy. Earlier we presented algorithms for performing property abstraction on numerical-valued device attributes. To evaluate its impact, we measured the number of states before and after the application of these algorithms, and the results are presented on the top of Fig. 10. We note that SOTERIA performs state reduction only for apps with devices that have numerical-valued

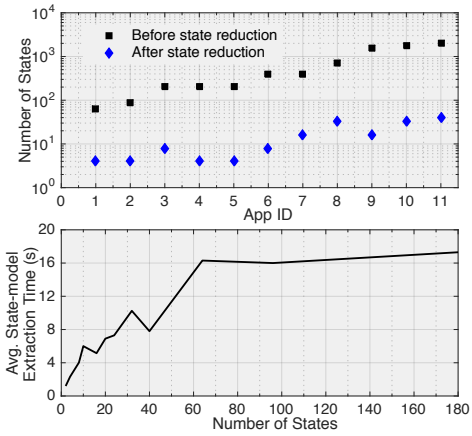

Figure 10: SOTERIA's state reduction efficacy (Top). SOTERIA's state model extraction overhead (Bottom).

attributes; examples include thermostats, batteries, and power meters. Among the devices we examine, there are ten such devices in analyzed apps, and 14 apps grant access to these devices, and the states of three apps have the same number before reduction and reduced to the same number. The figure shows that SOTERIA's state reduction often results in order of magnitude less number of states.

State Model Extraction Overhead. We ran SOTERIA with apps that have varying numbers of states and recorded the state-model generation time; the result is shown on the bottom of Fig. 10. The time includes the time for IR extraction, generating a graphical representation of the model, obtaining the SMV code of a state model, and logging (required for general properties). The average run-time for an app with 180 states was 17.3±2 secs. We note that the total time depends on the time taken by the algorithms we have developed for state reduction. For instance, an app having 32 states took more time than an app having 40 states due to many branches used in the 32-state app. Note that overheads can be mitigated by eliminating non-essential processing and other optimization.

We also measured the time for constructing a state model in multi-app environments. The state model of multiple apps requires extraction of each app's state model. SOTERIA's graph-union algorithm then finds 30 interacting apps (which have on average 64 states and six state attributes) and 4±2.1 seconds for the union algorithm.

Property Verification Overhead. We evaluated the verification time of a property on state models. The verification of a property took on the order of milliseconds to perform since the SmartThings apps have comparatively smaller state models than the large-scale ones found in other domains such as operating system kernels.

7 Limitations and Discussion

A limitation of SOTERIA is the treatment of call by reflection. As discussed in Sec. 4.2.3, SOTERIA constructs an imprecise call graph that allows a reflective call to target any method. This increases the size of state models and may lead to false positives during property checking. We plan to explore string analysis to statically identify possible values of strings and refine the target sets of method calls by reflection. Another limitation of SOTERIA is dynamic device permissions and app configurations. These may yield property violations because of the erroneous device and input configurations by users at install time. For instance, if a user enters an incorrect time value, the door may be left unlocked in the middle of the night.

SOTERIA's implementation and evaluation are based on the SmartThings programming platform designed for home automation. There are other IoT domains suitable for applying model checking for finding property violations, such as FarmBeats for agriculture [46], HealthSaaS for healthcare [20], and KaaIoT for the automobile industry [26]. We plan to extend our SOTERIA to these platforms by applying the IR-based analysis, as well as engage in large-scale analyses of IoT markets and industries.

8 Conclusions

We presented SOTERIA[4], a novel system that extracts state models from IoT code suitable for finding the security, safety, and functional errors. We evaluated SOTERIA in two studies; a study of apps on the SmartThings market, and a study on our novel MALIOT app corpus. These studies demonstrated that our approach can efficiently identify property violations and that many apps violate properties when used in isolation and when used together in multi-app environments. In future work, we will extend the kinds of analysis and provide a suite of tools for developers and researchers to evaluate implementations and study the complex interactions between users and IoT environments devices that they use to enhance their lives.

9 Acknowledgments

The authors thank Ashutosh Pattnaik and Prasanna Rengasamy for helpful discussions about this work. We also thank Megan McDaniel for taking care of our diet before the paper deadline. Research was supported in part by the Army Research Laboratory, under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and the National Science Foundation Grant No. CNS-1564105. 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 here on.

[4]The development of SOTERIA suite and its subsequent evaluation of IoT apps was a highly complex endeavor. An extended version of this paper is available with substantially more description, detail, and commentary, as well as 1) Groovy source code and IR of three example apps, 2) detailed description of general and application-specific properties, 3) advanced SmartThings idiosyncrasies for state-model extraction, and 4) description of the MALIOT apps and their property violations [9].

References

[1] APPLE HOME KIT. https://www.apple.com/ios/home/. [Online; accessed 29-April-2018].

[2] APPLE HOME KIT SECURITY AND PRIVACY ON IOS. https://www.apple.com/business/docs/iOS_Security_Guide.pdf. [Online; accessed 29-April-2018].

[3] APPLE HOMEKIT APP SUBMISSION GUIDELINE. https://developer.apple.com/app-store/review/guidelines/#homekit. [Online; accessed 9-April-2018].

[4] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN* (2014).

[5] BEYER, D., GULWANI, S., AND SCHMIDT, D. Combining model checking and data-flow analysis. *Model Checking* (2017).

[6] BIERE, A., ET AL. Symbolic model checking without BDDs. In *Algorithms for the Construction and Analysis of Systems* (1999).

[7] BURCH, J., CLARKE, E. M., AND LONG, D. Symbolic model checking with partitioned transition relations. *Research Report, CMU Computer Science* (1991).

[8] CELIK, Z. B., BABUN, L., SIKDER, A. K., AKSU, H., TAN, G., MCDANIEL, P., AND ULUAGAC, A. S. Sensitive information tracking in commodity IoT. In *USENIX Security* (2018).

[9] CELIK, Z. B., MCDANIEL, P., AND TAN, G. Soteria: Automated IoT safety and security analysis (Extended Paper). *arXiv:1805.08876* (2018).

[10] CIMATTI, A., ET AL. NuSMV 2: An open source tool for symbolic model checking. In *International Conference on Computer Aided Verification* (2002).

[11] CLARKE, E. M., AND EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs* (1981).

[12] CLARKE, E. M., GRUMBERG, O., AND PELED, D. *Model checking*. MIT press, 1999.

[13] DAS, M., LERNER, S., AND SEIGLE, M. ESP: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices* (2002).

[14] ELLSON, J., ET AL. Graphviz open source graph drawing tools. In *International Symposium on Graph Drawing* (2001).

[15] ENCK, W., ET AL. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transaction on Computer Systems* (2014).

[16] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *ACM CCS* (2011).

[17] FERNANDES, E., ET AL. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security* (2016).

[18] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security analysis of emerging smart home applications. In *Security and Privacy (S&P)* (2016).

[19] GROOVY CONSOLE - THE GROOVY SWING CONSOLE. http://groovy-lang.org/groovyconsole.html. [Online; accessed 20-April-2018].

[20] HEALTHSAAS: THE INTERNET OF THINGS (IOT) PLATFORM FOR HEALTHCARE. https://www.healthsaas.net/. [Online; accessed 20-April-2018].

[21] HO, G., LEUNG, D., MISHRA, P., HOSSEINI, A., SONG, D., AND WAGNER, D. Smart locks: Lessons for securing commodity internet of things devices. In *AsiaCCS* (2016).

[22] IGRAPH-THE NETWORK ANALYSIS PACKAGE. http://igraph.org/r/doc/. [Online; accessed 29-April-2018].

[23] IOTBENCH: A MICRO-BENCHMARK SUITE TO ASSESS THE EFFECTIVENESS OF TOOLS DESIGNED FOR IOT APPS. https://github.com/IoTBench. [Online; accessed 29-April-2018].

[24] JABLOKOW, A. How the IoT helps keep oil and gas pipelines safe. *Product Lifecycle Report* (November 2015).

[25] JIA, Y. J., ET AL. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *NDSS* (2017).

[26] KAAIOT: IOT AUTOMOTIVE. https://www.kaaproject.org/automotive/. [Online; accessed 20-January-2018].

[27] LATTNER, C. *LLVM compiler infrastructure project*. The architecture of open source applications, 2012.

[28] MCLAUGHLIN, S., AND MCDANIEL, P. SABOT: specification-based payload generation for programmable logic controllers. In *ACM CCS* (2012).

[29] MEAD, N. R. How to compare the security quality requirements engineering (square) method with other methods. Tech. rep., CMU Software Engineering Institute, 2007.

[30] OLUWAFEMI, T., ET AL. Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security. In *LASER* (2013).

[31] OPENHAB APP SUBMISSION GUIDELINE. https://marketplace.eclipse.org. [Online; accessed 17-April-2018].

[32] OPENHAB: HOME AUTOMATION. https://www.openhab.org/. [Online; accessed 15-April-2018].

[33] ORACLE SOFTWARE SECURITY ASSURANCE. http://www.oracle.com/security/software-security-assurance.html. [Online; accessed 15-April-2018].

[34] ORCUTT, M. Security experts warn congress that the internet of things could kill people. *MIT Technology Review* (2016).

[35] RONEN, E., AND SHAMIR, A. Extended functionality attacks on IoT devices: The case of smart lights. In *Euro S&P* (2016).

[36] RONEN, E., SHAMIR, A., WEINGARTEN, A.-O., AND O'FLYNN, C. IoT goes nuclear: Creating a zigbee chain reaction. In *Security and Privacy (S&P)* (2017).

[37] SAMSUNG SMARTTHINGS. https://www.smartthings.com/. [Online; accessed 9-April-2018].

[38] SCHUMACHER, M., ET AL. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2013.

[39] SHARIR, M., AND PNUELI, A. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications* (1981).

[40] SMARTTHINGS CODE REVIEW GUIDELINES. http://docs.smartthings.com/en/latest/code-review-guidelines.html. [Online; accessed 29-April-2018].

[41] SMARTTHINGS COMMUNITY-CREATED THIRD-PARTY APPS. https://community.smartthings.com. [Online; accessed 29-April-2018].

[42] SMARTTHINGS COMMUNITY FORUM USER STUDY POST. https://goo.gl/yC1wFf, 2018.

[43] SMARTTHINGS DEVELOPERS. https://github.com/SmartThingsCommunity. [Online; accessed 29-April-2018].

[44] SMARTTHINGS DOCUMENTATION. http://docs.smartthings.com. [Online; accessed 29-January-2018].

[45] TIAN, Y., ET AL. SmartAuth: user-centered authorization for the Internet of Things. In *USENIX Security* (2017).

[46] VASISHT, D., ET AL. Farmbeats: An IoT platform for data-driven agriculture. In *NSDI* (2017).

[47] YOSHIOKA, N., WASHIZAKI, H., AND MARUYAMA, K. A survey on security patterns. *Progress in Informatics* (2008).