IoTBox: Sandbox Mining to Prevent Interaction Threats in IoT Systems

Hong Jin Kang
School of Information Systems
Singapore Management University
hjkang.2018@phdcs.smu.edu.sg

Sheng Qin Sim
School of Information Systems
Singapore Management University
sqsim.2018@sis.smu.edu.sg

David Lo
School of Information Systems
Singapore Management University
david.lo@smu.edu.sg

***Abstract*—Internet of Things (IoT) apps provide great convenience but exposes us to new safety threats. Unlike traditional software systems, threats may emerge from the joint behavior of multiple apps. While prior studies use handcrafted safety and security policies to detect these threats, these policies may not anticipate all usages of the devices and apps in a smart home, causing false alarms. In this study, we propose to use the technique of mining sandboxes for securing an IoT environment. After a set of behaviors are analyzed from a bundle of apps and devices, a sandbox is deployed, which enforces that previously unseen behaviors are disallowed. Hence, the execution of malicious behavior, introduced from software updates or obscured through methods to hinder program analysis, is blocked.**

While sandbox mining techniques have been proposed for Android apps, we show and discuss why they are insufficient for detecting malicious behavior in a more complex IoT system. We prototype IoTBox to address these limitations. IoTBox explores behavior through a formal model of a smart home. In our empirical evaluation to detect malicious code changes, we find that IoTBox achieves substantially higher precision and recall compared to existing techniques for mining sandboxes.

I. Introduction

Internet of Things (IoT) systems, such as smart homes, are becoming popular and widespread. There are security risks at each level of granularity in an IoT environment. At the application-level, researchers have studied the security implications of IoT platforms that allow users to install apps that allow devices to interact with one another. A smart home comprises a collection of devices and apps. These apps control and connect different devices together, bringing many benefits and convenience to users, but this comes at the price of security risks. The increased attack surface has led to new types of attacks, such as those that introduce physical risks. For example, an app can be used to configure a smart home such that the windows will be opened if a temperature sensor in a room measures a reading above a user-specified temperature. A malicious app can spoof fake temperature readings to trigger the opening of the window, potentially allowing a break-in [1]. Other apps can open a door when no one is at home, disable a smoke detector, and induce seizures through the rapid strobing of lights [2]. Hence, it is important to understand the security risks of apps used in a smart home and defend against malicious behaviors.

One source of complexity is that apps can interact with one another through the devices they control and are triggered by. Malicious behavior can arise through the interaction of multiple apps, which may interact to produce unintended results in the physical environment. This motivates the need to study a smart home as a system of interacting apps and devices, rather than asserting that the behaviors of the individual apps in the smart home are safe.

Existing work has focused on using model checking or monitoring smart homes to ensure that joint behaviors of the apps do not violate safety properties [3]–[7]. Some of these techniques extract models of the apps through static analysis [3], [4], [6], checking them against predefined safety properties that were written by hand. For example, techniques may enforce a property that the door is never unlocked while the occupants of a smart home are away, and another property may be that the lights in the house are never automatically switched off while the occupants are home.

A problem is that these properties may not anticipate all legitimate uses of the apps. Indeed, the normal executions of some apps deliberately violate these safety properties and users may install these apps knowingly. A security policy that prohibits any switches from turning on when the users are not home will prevent the use of the legitimate app, *VacationLightingDirector* [8], which simulates occupancy in a house by occasionally switching on the lights when the user is on vacation. These techniques cannot distinguish between user-intended violations of the properties from real safety problems. Moreover, as new devices and new apps are developed and introduced into the market, there will be new forms of incorrect behavior involving new apps and devices. Existing techniques cannot defend against new modes of attacks involving new channels or devices, for which safety properties have not been written yet. This motivates the need to automatically identify relevant security policies.

In this work, we propose that we can begin addressing the above-mentioned limitations through the technique of *sandbox mining* [9], inspired by the work by Jamrozik et al. mining sandboxes for Android apps [9]. We propose IoTBox, which automatically mines a sandbox for a smart home. Rather than writing out safety properties by hand, IoTBox encodes the current behaviors of a smart home and protects the user against unexpected behaviors. The possible behaviors of a smart home are encapsulated in the sandbox, which detects changes in behavior. Changes in behaviors may be caused by the introduction of malicious behavior in an app, unexpected bugs due to a new interaction between apps, or the removal of behavior that the user depended on. After the sandbox is mined, it excludes behavior that was not previously captured during analysis. If an app is replaced or updated with malicious code, the sandbox prohibits any behavior that violates its rules and reports it. The user of the apps can investigate and decide if the new behavior should be permitted. If a change is benign and acceptable to the user, the sandbox can be relearned on the updated smart home again and will defend the smart home system against new threats.

However, unlike mining sandboxes for Android [9]–[11], the possible behaviors in a smart home cannot be as easily explored using test case generation. Unlike traditional software systems, actions invoked within the smart home may take several seconds before they execute. The state space, composed of every app located in the smart home, is enormous compared to individual Android apps. Yet, it is essential to comprehensively explore the possible behaviors of the smart home to learn accurate rules.

Another challenge is that malicious behavior may disable an action instead of invoking new actions. For example, an adversary may disable existing behavior that locks a door when the user is asleep, leaving it unlocked for an adversary. Consequently, a sandbox for IoT should detect changes in behavior that causes missing actions.

In this study, we overcome the above challenges by leveraging the precision of the formal models proposed in existing studies [6]. Specifically, IoTBox uses a formal model of a smart home [6] to identify a complete execution context for any automated action with the help of a model checker. During monitoring of the environment, IoTBox then uses these execution contexts to identify actions that it expects the smart home to automatically run. If there is a mismatch between the expected actions and actions in reality, then IoTBox detects that there is a behavioral change and alarms the user.

In this work, we present three contributions:

- We present the need for inferring security policies personalized to individual smart homes. We show that existing techniques using handcrafted policies may lead to false positives on usage contexts that are likely to be legitimate.
- To address this problem, we propose that techniques to mine sandboxes can be used for monitoring IoT environments. We develop IoTBox, which mines a sandbox by exploring behavior with the help of a model checker.
- We empirically evaluate IoTBox, comparing it to techniques for mining sandboxes of Android apps. We discuss IoTBox's mitigation to the problem of false negatives.

II. Background

In this section, we present relevant background. IoTBox builds on top of both prior studies formalizing an IoT system [6], and techniques to mine sandboxes [9], [10], [12].

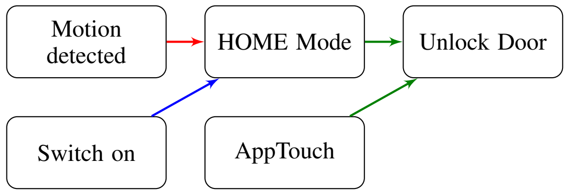

Fig. 1. This smart home transits to *HOME* mode on detecting motion (App1) or if a light is switched on (App2). The door is unlocked when the mode has changed or through a user interaction with the SmartThings app (App 3).

A. Smart Home Platforms

We focus on the apps in the Samsung SmartThings [13] and IFTTT [14] platforms. In a typical smart home platform, physical devices have a corresponding virtual representation on the platform. The state of each device is a mapping of *attribute*s to *values*. The state of a device can be modified through *actions*, such as switching on the lights (toggling *switch.off* to *switch.on*). The set of attributes and actions of a device is determined upon registration of the physical device to the smart home platform, where it is granted a *capability* (e.g. a lock, a switch, or a thermostat).

Once registered, apps can access these devices by specifying the required capability. Within the apps written in Groovy, each device can then be accessed as an object. Device attributes are read through the attributes on the object, and the actions on the device (e.g. *door.unlock()*, *alarm.off()*) are invoked through method calls. Installed apps can interact with one another through various means. For example, an app can change the mode of the smart home to *Home*, indicating that the user is home, and a second app unlocks the door when it detects that there is a mode change.

Arbitrary apps can be installed by the user on the smart home platform. Each app has a set of capabilities it requires, in which the user binds existing devices to. Apps are written following an event-handling paradigm. Usually, apps wait for events from sensor devices and trigger new actions through actuator devices. Apps can interact through devices (e.g. one app toggles a switch at a particular time of the day, and another app is triggered by the switch to turn on all the lamps in the house) or through physical channels (e.g. one app triggers a lamp in a room, which causes another app to pick up sensor readings from an illuminance sensor).

We show an example of app interactions spanning multiple apps in Figure 1. A first app switches the smart home to *HOME* mode after a motion sensor detects motion, a second app switches the smart home to *HOME* mode if a light is switched on, and a third app unlocks the door if it detects a transition to *HOME* mode or if the user unlocks the door through the SmartThings apps. This transitively creates a link between a motion sensor event and unlocking the door.

B. Formal model of a smart home

In this study, we use the formal model of a smart home proposed by Alhanahnah et al. [6]. They provide a tool, IoTCOM [6], to analyse apps written for the SmartThings and IFTTT platforms. IoTCOM provides a parser that translates these apps to Alloy [15] models. A smart home consists of a set of devices and apps. Each device has one or more capabilities, attributes, and at any time, a value associated with each attribute. Smart apps can connect these devices, such as invoking an actuator (e.g. unlocking a door) given a reading from a sensor (e.g. a motion detector sensing motion). A smart app is a collection of rules. Rules are tuples of a **Trigger** $\times$ a set of **Conditions** $\times$ a set of **Commands**.

Triggers represent the conditions in which the app is activated. These are often events from the smart home sensors, such as a door opening. A trigger comprises a device capability, an attribute associated with the capability, and the value of the attribute. Each rule has at most one trigger.

Conditions represent the logical predicates on the state of other devices/smart home. These predicates guard the invocation of a rule's commands. For example, after a rule is triggered by an event ("smoke detected"), a rule may have other conditions ("the door is locked"), before it executes a command ("unlock the door"). A trigger comprises a capability, an attribute associated with the capability, and the value of the attribute. Each rule may have any number of conditions.

Commands represent the actions taken by a rule, including device actuations that change the physical state of the smart home. Each command comprises a device capability, an attribute, and a value. A rule may have one or more commands.

The SmartThings platform also allows for state variables that persist over different executions of an app. Each state variable is encoded as a device capability in IoTCOM, allowing for analysis of behaviors that depend on these variables.

Another consideration is communication between devices and apps through physical channels. IoTCOM [6] includes the physical channels in its model of the smart home and models them as a mapping of capabilities to a physical channel. Each channel can link together an actuator device (e.g. a valve) and a sensor device (e.g. a water sensor).

IoTCOM [6] precisely represents the smart home in Alloy and uses the Alloy Analyser to assert that safety properties hold on the smart home. First, IoTCOM converts the apps in the smart home to Alloy models. Their joint behaviors are represented as a *behavioral rule graph*, which captures the behaviors of apps, linking together the triggers, conditions, and commands of relevant rules. IoTCOM then asserts that the apps do not violate any safety property. In our work, IoTBox can be built on top of any formal model, however, as prior studies faced scalability issues [3], [4], [16], we use IoTCOM's precise model of a smart home.

C. Mining sandboxes

Traditionally, malicious programs have been executed in sandboxes, which blocks access to resources that have security concerns. Researchers [9], [11] have suggested that entirely blocking/allowing access to a certain resource may be too coarsed-grained, and have proposed to identify more granular conditions of accessing each resource, restricting access if the conditions are unmet. To do so, techniques have been proposed to automate the mining of rules for a sandbox. Jamrozik et al. [9] suggest mining associations between GUI elements and sensitive API access through the generation of Android GUI tests. This allows the identification of rules permitting access to sensitive resources, such as the camera, only if the user is performing specific actions on the app.

There are two phases to mining sandboxes, the exploration phase and the sandboxing phase. In the first phase, the behaviors of an app are explored and encoded into the sandbox. In the sandboxing phase, new behaviors that were not seen during the first phase are prohibited. If the app requires a new behavior, the sandbox should prohibit it or defer the request for approval by a human user.

For example, if a new version of an app is released, a user can install and run it in the sandbox. As the sandbox detects previously unseen behavior (e.g. reading a file), it alerts the user. Then, the user assesses the situation, and if the user determines if the new behavior is desirable, its execution is permitted. Otherwise, the execution of the potentially dangerous behavior is stopped.

For these techniques to work effectively, it is necessary to sufficiently explore the app. If a normal behavior is not accessed during the exploration, it will produce false alarms during the sandboxing phase. Existing techniques [9]–[12] rely on test generation to explore the app. These studies suggest that test generation can be effective for exploring behaviors in individual Android applications and Linux containers.

Key to mining sandboxes with test case generation is the *test complement exclusion* [17] method. There is no guarantee that behaviors that have not been observed will not occur in the future. Test complement exclusion turns this limitation into a guarantee by using the sandbox to allow only behaviors seen as the sandbox is mined. Therefore, if no malicious behavior was observed, then no malicious behavior can execute.

Existing techniques differ in the context considered to determine if a given resource should be accessible. If the rule allowing a resource access is too coarse-grained, then it may fail to detect malicious behaviors, while if it is too granular, then it may stop benign behaviors with inconsequential differences from executing. In the work of Wan et al. [12], only the set of system calls are tracked and their contexts are ignored. In Jamrozik et al's work [9], the execution context is the GUI element that the user interacted with before an Android API call. In Le et al's work [10], the execution context is the sequence of other API calls before the given API call.

III. IoTBox

Our primary contribution is the proposal of IoTBox, a technique that mines a sandbox for a smart home. Key to our approach is to consider only causal information between events in a smart home. First, IoTBox connects events and actions to determine all possible paths leading to any action. If there is some unexplained difference between IoTBox's expectations and the actions in the real world, then it suggests that there is a behavior in the smart home that differs from the rules that IoTBox has learned. There are two phases to IoTBox, much like other techniques mining sandboxes [9]–[12], the exploration phase and the sandboxing phase.

In the exploration phase, IoTBox thoroughly explores the behaviors of the software system to determine the execution context of possible actions in the smart home. We consider an action's execution context as the set of execution paths causing the action's execution. IoTBox refines the execution context of a given action, finding all possible causes of the action, by utilizing the Alloy Analyzer to identify all paths linked to the action, including non-trivial interactions across multiple apps and timed events. This creates two guarantees. Firstly, only events that are linked to the given action are identified, and secondly, all such events are found. In contrast, existing work on mining sandboxes rely on test case generation, which may fail to comprehensively explore the entire search space of behaviors (In Section V, we discuss the tradeoffs of this choice). Abstracting the identified behaviors as rules, IoTBox uses these rules in the sandboxing phase to judge if there is any missing or disallowed action given recently observed events.

A. Exploration phase

To mine a sandbox, we use the Alloy Analyzer, a model checker, to thoroughly explore the behaviors of a bundle of apps encoded in the behavioral rule models, introduced by Alhanahnah et al. [6]. As described before in Section 2, these models are encoded in Alloy. The exploration phase can be viewed as answering the question *"What are all possible paths that lead to a given action?"*.

After we have produced formal models of all apps in the smart home, our first step is to identify all actions that the apps may execute. This is done by traversing all rules and picking out all actions that may be executed. Our next step is to find all execution paths to lead to any given action. In the example in Figure 1, one such path is ("Motion Detected" $\rightarrow$ "Home Mode" $\rightarrow$ "Unlock Door"). This entails finding out all events, $events$, that can trigger each action, a. First, we identify an event, $event$, that will trigger it (i.e., by simply using the triggers and conditions of the rules that the action is a command of). Next, we construct an Alloy assertion that checks that all events on a path leading to the action is either $event$ or is preceded by it. The following Linear Temporal Logic [18] fragment describes that any occurrence of a must be on an execution path triggered by $event$:

$$\neg a \, W \, event$$

$event$ has a device capability, an attribute, and a value associated with the attribute. This is used to initialize $events$, which initially contains just $event$. For example using Figure 1, given a trigger (*location*, *location_mode*, *HOME*), which matches an event on a motion sensor, we initialize the set, $events$, with a single event, (*location*, *location_mode*, *HOME*). An example of the assertion in Alloy is shown in Figure 2. It asserts that for all rules unlocking the door, all

```
assert {
  no r : IoTApp.rules, action : r.commands {
    action.attribute = lock
    action.value     = unlock
    (some predecessor : r.*(~connected),
    action' : predecessor.triggers
     {
      not {(
      {
          action'.attribute = location_mode
          action'.value = HOME

      }) or
      (some predecessor' : predecessor.*(~connected),
        action'' : predecessor'.triggers
         {
              predecessor ≠ r
              action''.attribute = location_mode
              action''.value = HOME
          }
      )}
    })
  }
}
```

Fig. 2. Example of an Alloy assertion

preceeding events are on the chain of events triggered by the transition to *HOME* mode.

We run the Alloy Analyser to check the assertion. If the assertion fails, the Alloy Analyser will generate a counterexample of an event, different from $event$, that transitively triggers the action. This new event, $event2$ (one of "motion detected", "switch on", or "App Touch" in our example), is added to $events$, and we modify the assertion to look for counterexamples of paths which are triggered by events not in $events$. Only $event2$, a single event that precedes all other events on the path, is added. Obtaining the possible paths from $event2$ is done afterwards. There may be multiple paths that are triggered by $event2$ that result in a.

$$\neg a \, W \, (event \vee event2)$$

Again, we execute the Alloy Analyser to check this assertion. If it fails, we once again use the counterexample it finds to modify the assertion. This process continues until the Alloy Analyser fails to find a counterexample, and that all chains of events that lead to a have been accounted for. At the end of this process, this assertion doubles as an interpretable security policy. The policy declares all possible events that can lead to a, and checks that other events do not transitively trigger a. This process produces the following assertion with n different triggering events:

$$\neg a \, W \, (\vee_1^n event_i)$$

An advantage of this technique is that it identifies only events that have a causal relationship with a based on the behavioral rule graph. Only events that are a root cause of the execution of a are identified and included in $events$. This avoids the problem of spurious associations had we applied data mining techniques on the large number of events.

Next, we traverse the behavioral rule graph, beginning with the members of $events$. We find all paths that lead to a. A path is a series of events and does not contain loops. From all paths that start with a member of $events$, we identify all subpaths that end with a, and in turn, these subpaths are paths. This gives us the set of paths, $paths$.

$$execution\_context(a) = paths$$

We treat $paths$ as the execution context of a. For an invocation of a, at least one path (e.g."Motion detected" $\rightarrow$ "Home Mode") within the execution context must be satisfied; all conditions and their predicates along the path must be satisfied, and all triggers in this path must have been triggered. $satisfied(path)$ is an implementation detail that will be described in Section III-B.

If an execution context of an action is satisfied, then an actuation of the action is expected. With this assumption, IoTBox looks for mismatches between the expected and actual actions in the smart home. Let us define two predicates: $expected(a)$ is true when the execution context of a has been satisfied. $actual(a)$ is true when the action a has been observed in the IoT environment. Based on these two predicates, the following gives a formal definition of IoTBox's assumption:

$$satisfied(execution\_context(a)) \leftrightarrow$$
$$\exists path(path \in execution\_context(a) \land satisfied(path))$$
$$expected(a) \leftrightarrow satisfied(execution\_context(a))$$

By the end of this phase, we have constructed for any given action, a, an execution context that comprises the set of all paths that lead to the triggering of a. The execution context allows for both the detection of new causes of an action and missing actions.

Given an action a that has taken place, IoTBox considers it as a disallowed action if its execution was not expected.

$$disallowed(a) \leftrightarrow \neg expected(a) \land actual(a)$$

Conversely, IoTBox considers an action to be missing if its execution was expected (e.g. "Unlock Door" is expected after observing "Motion Detected" and "HOME Mode") but is not observed in reality.

$$missing(a) \leftrightarrow expected(a) \land \neg actual(a)$$

B. Sandboxing phase

The objective of the sandboxing phase is to detect if there is a change in the behaviors of the smart home. This is done through monitoring executions at runtime. While the exploration phase was done through static analysis, it is insufficient to use static analysis to prevent malicious behaviors; malicious behaviors can be invoked through dynamic language features, such as call-by-reflection. Within IoTBox, the sandboxing phase answers the question: *"For all automated actions, are there changes in the paths that can trigger it?"*

If there is an unexpected action, then it implies that there is a new path that IoTBox is unaware of. If there is a missing action, then it implies that some paths has been removed.

When deployed, IoTBox communicates with the app before the execution of each action and whenever an event takes place. This requires the instrumentation of the smart apps to send events to IoTBox. We write a Groovy program transformer that modifies the Groovy smart apps at each event handler and at call sites of any action. The modified app calls out to a third-party server to either update IoTBox of new events or to request for permission to run an action. When an event handler runs, it logs the event to IoTBox. Before an action is invoked, the app waits for permission from IoTBox. Malicious behavior may be invoked through dynamic program features such as call-by-reflection, and we add guards to locations using Groovy's *GString* feature to perform dynamic method invocations. The action, resolved at runtime, waits for permission from IoTBox before execution, similar to statically invoked actions.

The traces of events and actions previously taken in the smart home prior to the action are used by IoTBox to make a decision to allow or reject an action. As events occur in a smart home, IoTBox updates its model of expected actions based on the execution contexts of the actions. With every new event that is reported, IoTBox determines if there is any missing event. IoTBox warns the user if an action is rejected or if there is a missing action.

We did not experience significantly increased latencies when we deployed our tool on the SmartThings Simulator. Our findings (less than 15% overhead) are similar with previous studies, which found that even after instrumentation, the latency for an action to execute is largely caused by communication between the IoT platform's server and the physical device [5].

IoTBox uses the most recent executed events to make decisions, much like DSM, the state-of-the-art technique to mine sandboxes [10]. Using these events, it makes a best-effort guess if each *path* in an action's execution context is satisfied. IoTBox is conservative about the triggers on the path, requiring that they must all be present before allowing an action, but liberal in checking the conditions; an action is denied only if there is evidence that a condition is not satisfied. Concretely, we propose the following algorithm to determine if $satisfied(path)$ holds for a given $path$.

In Figure 3, we iterate over the sequence of events in the traces, using `tracePointer` (initialized in Line 1, incremented in Line 16), and over the triggers on the path (Line 3). Each trigger has conditions associated with it, coming from the same rule (Line 4). Between the events that match subsequent triggers, if an event matches a condition on its device and attribute, but with a different value, then it causes the valuation of the condition to be false. If the event matches on the value, then it causes the valuation to be true. We track the conditions' valuation using `isConditionNegated` (initialized in Line 2), which maps a condition to true if an event caused the condition to have a negative valuation (set in either Line 10 or 12). Before the next trigger is matched, if a condition's valuation is false, then the path cannot be satisfied (Lines 18-20). `any(isConditionNegated, conditions)` returns true if any condition maps to `true`. A path is not satisfied if there is a missing trigger (Line 21) or if there is evidence that a required condition is false (Line

```
Input: A sequence of events, $trace$.
Input: A path, $path$.
Output: $satisfied(path)$, either true or false.
1:  $tracePtr = 0$, $event$ = null
2:  $isConditionNegated = \{\}$
3:  for $trigger \leftarrow$ triggersOf($path$) do
4:      $conds$ = conditionsAssociatedWith($trigger$, $path$)
5:      while $event$ != $trigger$ && $tracePtr$ <$trace.len$ do
6:          $event$ = $trace[tracePtr]$
7:          for $cond \leftarrow conds$ do
8:              if $cond.device$ = $event.device$ && $cond.attribute$ = $event.attribute$ then
9:                  if $cond.value != event.value$ then
10:                     $isConditionNegated[cond]$ = true
11:                 else
12:                     $isConditionNegated[cond]$ = false
13:                 end if
14:             end if
15:         end for
16:         $tracePtr$++
17:     end while
18:     if any($isConditionNegated$, $conds$) then
19:         return false
20:     end if
21:     if $tracePtr$ == $trace.len$ && $event$ != $trigger$ then
22:         return false
23:     end if
24: end for
25: return true
```

Fig. 3. Algorithm to determine if a given $path$ is satisfied (i.e., $satisfied(path)$) based on the events in $trace$.

18). Otherwise, the path is satisfied (Line 25).

This algorithm is best-effort and may not always be accurate. A smart home is stateful and local information from the most recent traces may not be enough to make the right decisions; it is not always possible to determine the truth value of a condition. For example, a condition that the home is in *HOME* mode cannot always be determined from the most recent traces, as *HOME* mode may have been set hours or even days ago. While it may be possible to simply store the all updates to state of the smart home, including state changes from multiple days ago, we surmise this poses a privacy risk if this monitoring service is ever compromised [19], [20].

IV. Empirical Evaluation

We are interested in answering 2 research questions:

- **RQ1: How frequently do handcrafted security policies lead to false positives?**
  Existing approaches detect security issues from the **joint** behavior of multiples apps. We claim that their handcrafted policies may produce many false positives. In this research question, we investigate if **individual** benign apps violate the security policies. As users are likely to understand and reason about individual apps, we assume that, if installed, their behaviors are intentionally introduced into a smart home.
- **RQ2: How effective is IoTBox?**
  In this research question, we investigate the effectiveness of IoTBox against DSM, previously proposed for Android apps, and a simple strawman sandbox.

A. RQ1: How frequently do handcrafted security policies lead to false positives?

This research question investigates the prevalence of false positives from the use of handcrafted security policies from prior work. We test a random sample of 500 apps from public repositories containing existing apps [21], [22]. For each app, we check the model produced by IoTCOM against the 36 policies used in IoTCOM [6]. These policies are similar to the policies used in other studies [3]–[5]. As we only test individual apps, all violations are not caused by interactions between apps, and are likely to be from behaviors that were intended if a user installed the app.

1) Findings: We find that out of the 500 apps, 326 of them (65%) violate at least one policy, with a total of 572 violations. On average, there is 1 violation per app. If deployed in a real setting, many violations of the security policies are false alarms. While the security policies can catch dangerous behaviors, they cannot be used out of the box.

One cause of the numerous false alarms is that the policies are too broadly specified, causing IoTCOM to detect violations even for legitimate uses. For each app, we describe the policy violated (based on the policies used in IoTCOM [6]), highlight the reason for the violation, and if the violation of the policy is intended. False alarms can come from either surprising uses of devices or from IoTCOM's overapproximation of execution paths that will not be taken in reality. While we discuss only 4 violations (out of 326) in Table I, we expect that, in the wild, there are many situations where IoT devices are used in unexpected ways. Many of these uses will violate security policies that do not anticipate surprising uses. This problem was also observed by Celik et al. [3]. For example, they reported anecdotes of users using flood sensors that produce alerts when water levels are low (contrary to its expected use in detecting floods) for reminders to water their plants.

Researchers have pointed out that users rarely use tools that produce many false positives, inhibiting its usage [26]–[28]. Users would need to apply significant effort to understand and tweak each security policy to their smart home. This finding motivates more research into automating this process.

B. RQ2: How effective is IoTBox?

To inveistate the effectiveness of IoTBox, we compare the number of bundles of apps in which IoTBox successfully identifies malicious changes in behavior. In all cases, we compare IoTBox against two baseline techniques, a strawman sandbox and DSM [10]. All techniques take traces of events as input. First, for each bundle, we produced events based on the formal models to simulate the smart home, triggering an average of 1208 actions. Next, to confirm our findings, we

TABLE I
BEHAVIORS OF INDIVIDUAL APPS THAT VIOLATE THE HANDCRAFTED SECURITY POLICIES. POLICIES PREFIXED WITH P INDICATES A SAFETY PROPERTY, WHILE T INDICATES A GENERAL COORDINATION THREAT.

Policy Violated (policy identifier in the IoTCOM paper)	**Reason for violation**
The heater should not be turned off when the temperature is low (P.7)	Intended; the app (use_outdoor_temp_to_turn_on_off_a_switch) switches off a switch (which may be the heater) based on the temperature outside, not inside the smart home.
Lights are not switched off if someone is at home (P.9)	Intended; the app (IlluminatedResponsetoUnexpectedVisitors [23]) toggles the lights on and off for illuminating someone snooping about the house (e.g. a burglar), switching the lights off if the burglar has left.
No rules with conflicting actions but same triggers/conditions (T7)	Intended; the app (LockDoorafterXminutes [24]) sets a variable for internal bookkeeping (to track that the door was automatically opened), but sets it to a different value (that it has been closed) after some time.
Location mode should be set to HOME when someone is at home (P.15)	Intended; while the user is home, the app (MotionModeChange [25]) may set the mode to NIGHT.

generate tests on the SmartThings Simulator and collect traces from the app executions, triggering 555 actions on average.

1) Experimental Setting: We use the flawed apps that were studied previously. We use the bundles of apps from IoTMAL, used in the evaluation of previous studies (Bundle 1-6) [3], [6]. Furthermore, we proposed new bundles of apps (Bundles 7-16), constructed with individual, flawed apps proposed in prior studies [3] and combining them with other benign apps. We study the same apps used in the evaluation of IoTCOM [6]. Bundle 17 is the example from Figure 1, and we constructed a malicious variant by removing a transition to *HOME* mode.

In each bundle, we locate the app with malicious behavior and create a variant of the bundle by removing the unsafe logic. To evaluate the ability of IoTBox to detect missing behavior, we constructed variants of several bundles by removing a piece of behavior required for it to function correctly. We pass the benign bundles of apps as input to IoTBox, which explores their behaviors and constructs rules for the execution of each action. Thus, in this work, a benign bundle of apps is the modified bundle of apps without malicious behavior. A malicious bundle of apps is either 1) the original bundle containing an app with a malicious behavior, or 2) a bundle of apps modified from a benign bundle to remove a necessary behavior. A total of 17 benign bundles of apps were explored, and 20 variants of these bundles of apps with either additional malicious or missing necessary behavior were constructed.

Next, we statically produce execution traces of the bundles of apps. The executions of the benign bundles of apps are used for learning the sandbox for both the strawman sandbox and DSM. The traces are produced from the models to allow for a fair comparison of existing techniques and IoTBox, as only the parts of the IoT app captured by the formal models are used to produce traces. To produce events from the formal models, we identify two types of event for each trigger and predicate; one event will set the predicate to true, and another will falsify the predicate. We track the state of the environment as a mapping of the attributes of every device located in the environment to one value. We model time using a counter, which increases for every event. At each time step, a random event will be selected for execution. As an event is executed, the state of the environment is modified. Every time an event is executed, we iterate over all the rules in the formal model and determine for each rule, if its trigger matches the event, and if its conditions have been met by the modified state. If so, then an event that matches the command will be executed, either immediately or after some time.

The execution traces from the malicious bundles of apps are input to all techniques. If a technique rejects more than 1% of actions, then we consider that it is able to detect the malicious behavior in the bundle.

In simulating the apps' executions, we constructed a challenging experimental setup. We produced a large number of events while simulating the possibility of race conditions between the apps. The triggering of apps may be delayed to reflect real-world network conditions. Events from the execution of an app may be interleaved between executions of other apps. Thus, the threshold of 1% permits some degree of false positives caused by these challenging conditions. Without this threshold, DSM will find two times more false positives. In practice, we expect that these challenging conditions will not frequently occur. Also, we believe that users will find reasonably rare false alarms to be acceptable. We manually inspected the statically generated traces to verify if the traces are plausible based on the source code. Of 50 randomly sampled traces, none were infeasible.

The strawman sandbox detects if there is any action executed by the smart home that was not seen during the exploration phase. Any previously unseen action is rejected. DSM [10] takes the sequence of events and actions that occurs before an action as input. DSM uses an automaton model, inferred through a Recurrent Neural Network [10]. If the events are rejected by this model, the action is prohibited.

We use IoTBox as described in Section III. Rules of the sandbox are first learned through the exploration of the modified bundles of apps without any malicious behavior. Next, we test the sandbox against both the original bundle of apps containing the malicious behavior, and the modified bundles without malicious behavior. We count the number of

TABLE II
BUNDLES WITH MALICIOUS BEHAVIOR THAT WERE DETECTED. T INDICATES THAT THE MALICIOUS BEHAVIOR WAS DETECTED, F OTHERWISE. FP INDICATES THAT THE BENIGN BUNDLE WAS CLASSIFIED AS MALICIOUS. (M) INDICATES BUNDLES WHERE THE MALICIOUS CHANGE IS A REMOVAL OF EXISTING BEHAVIOR.

Bundle	**Strawman**	**DSM**	**IoTBox**
1. Bundle 1	F	F	**T**
2. Bundle 2	T	T	T
3. Bundle 3	T	T, FP	T
4. Bundle 4	T	T, FP	T
5. Bundle 5	T	T	T
6. Bundle 6	T	T	T
7. MaliciousBatteryMonitor [16], [31]	T	T	**T**
7 (M)	F	F	F
8. ID1BrightenMyPath	F	T	**T**
9. ID2SecuritySystem	F	T	**T**
10. ID3SmokeAlarm	F	T	F
11. ID4PowerAllowance	F	F,FP	F
12. ID5FakeAlarm	F	T	**T**
12 (M)	F	F	**T**
13. ID6TurnOnSwitchNotHome	T	T	T
13 (M)	F	F	**T**
14. ID7ConflictTimeandPresence	F	**T**	F
15. ID8LocationSubscribeFailure (M)	F	F	**T**
16. ID9DisableVacationMode	T	T	T
17. Figure 1 (M)	F	F	T
# True Positives	8	13	**16**
# False Positives	**0**	3	**0**
Recall	40%	70%	**80%**
Precision	**100%**	75%	**100%**
F1	57%	72%	**88%**

malicious bundles of apps that were correctly identified (**T**rue **P**ositives), the number of benign bundle of apps identified as malicious (**F**alse **P**ositives), and the number of malicious bundle of apps identified as benign (**F**alse **N**egatives).

We evaluated the effectiveness of the tools using Precision, Recall, and F1. Precision and Recall are computed as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}} \ \text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$$

A precision of 100% indicates that the absence of false positives, while a recall of 100% indicates that every malicious behavior was caught. Finally, F1 is the harmonic mean of precision and recall. These metrics are widely used in the literature of both mining sandboxes [10], [11], as well as other related domains such as the detection of malware [29], [30].

2) Experimental Results from Statically Produced Traces: Table II summarizes our results on the static traces, for 16 out of 20 bundles, IoTBox detected a change in behavior that should be disallowed. In contrast, DSM and the strawman detected only 13 and 8 bundles with changed behavior.

We evaluated the techniques on false alarms and found that DSM produced false positives on 3 bundles. Both the strawman and IoTBox do not produce false positives.

As we produced over a thousand traces, we rule out a lack of training data as a reason for DSM's relative ineffectiveness. We hypothesize that its poor performance is caused by the tight coupling of its decisions to the local context, requiring that all necessary information to make the right decision appear in the most recent traces. In fact, the relevant events to make the right decision can occur far apart from one another. While Le et al. [10] suggests that DSM's use of Recurrent Neural Networks may help in capturing long-term dependencies between events, still, an IoT system is stateful and the relevant state changes may have occurred before the collection of the most recent traces. Ultimately, DSM is fooled by spurious patterns in the most recent traces. On the other hand, IoTBox encodes domain knowledge of IoT apps, only loosely enforcing conditions to check if they are satisfied.

Only IoTBox can detect changes in behavior that result in missing actions (indicated with (M) in Table II). Both DSM and the strawman do not detect missing actions. On the other hand, IoTBox detects 4 out of 5 cases with missing actions.

Both the strawman sandbox and IoTBox have 100% precision. Overall, IoTBox performs best, in terms of Recall and F1 score. Even if we omit the cases with missing actions, then IoTBox has a Recall of 60% and an F1 of 75%. Overall, we believe that there is sufficient evidence to suggest that IoTBox is more effective than existing sandbox mining techniques.

3) Experimental Results on Traces from Test Case Generation: Next, we investigate the effectiveness of IoTBox on traces produced from test case generation. This approach uses the SmartThings simulator [32]. We instrument the apps, adding log statements to them.This allows us to collect information about the state of the smart home without modification to the platform that the apps runs on. Our objective is to elicit enough traces to evaluate IoTBox to determine if our findings are likely to hold in practice.

Our test generation strategy identifies relevant event types by detecting which devices are present in the smart home. After that, it randomly generates events of these types through the simulated devices in the SmartThings IDE. As some apps are time-sensitive, we transform the Groovy programs, replacing time-related functions with mocks. Our test generator simulates the passing of time. Instead of using the current real-world time, the app fetches the mocked time from our server. Time monotonically increases with the number of executed events; every time an event is executed, the test generator advances time by a random number of minutes. For functions scheduled to run after some time (i.e., using *runIn*), we set its waiting duration to between 1 to 3 minutes, giving them a high chance of executing within the experimentation duration.

Execution traces are collected from the logs. Each trace is a sequence of events and actions executed by the apps. For each bundle, we generate tests for over an hour. The same execution contexts are used, unmodified, from the previous experiments.

Due to the limitations of the SmartThings simulator, we restrict our analysis to only a few bundles of apps. The simulator cannot simulate physical channels of interactions (e.g. an app switching on a lamp may trigger an app reading from a light sensor), and does not support simulating every device type, a known limitation of test generation on the SmartThings simulator [16]. Therefore, we cannot create an accurate simulation of several bundles (Bundles 4-7, 12-14) in our evaluation dataset. We also omit Bundles where IoTBox was ineffective on the static traces (Bundles 10, 11).

TABLE III
EFFECTIVENESS OF IOTBOX ON TRACES FROM TEST GENERATION. NOT ALL BUNDLES CAN RUN ON THE SMARTTHINGS SIMULATOR AS THEY USE DEVICES UNSUPPORTED BY THE SIMULATOR.

Bundle	Malicious Behavior Detected
1. Bundle 1	T
2. Bundle 2	T
3. Bundle 3	T
8. ID1BrightenMyPath	T
9. ID2SecuritySystem	T
16. ID9DisableVacationMode	T
17. Figure 1 (M)	T

Our results are shown in Table III. IoTBox detects the malicious changes in behaviors on all seven bundles of apps. As before, there are no false positives. Overall, these results agrees with the evaluation results using the statically produced traces in Table II, and supports our findings that IoTBox is effective in mining sandboxes for a smart home.

V. DISCUSSION

A. Risk of encoding malicious behavior in the sandbox

In any technique mining sandboxes, there is a risk that the mined rules encode malicious behavior that was already present in the smart home [9], [12]. If so, the sandboxing phase does not prevent its execution as it is an expected behavior. This is one source of false negatives [9], as the malicious behavior would be considered benign. On the other hand, this implies that *the malicious behavior is already explicitly described in the security policy mined during the exploration phase* and can be checked by a human user.

We analyze the quality of the rules produced by IoTBox. The rules mined by IoTBox are precise, expressed as Alloy assertions. Furthermore, IoTBox can visualize the paths from any unexpected trigger to a given action. This lends the rules to inspection by human users of an IoT system.

We investigate what a malicious behavior encoded in the rules may look like. We use the running example involving the unlocking of doors introduced in Figure 1, and now, we point out it was crafted with a flaw in mind. While it correctly unlocks the door when the smart home transits to *HOME* mode, in fact, it unlocks the door in any mode change, including mode changes away from *HOME* (e.g. to *AWAY*) [33]. This behavior is unexpected to our hypothetical user who did not carefully inspect the app.

In Figure 4, we show a more complete version of the graph from Figure 1 (in Section 1) with this surprising behavior. An excerpt of the security policy mined by IoTBox is shown in Figure 5. Both the Alloy assertion and accompanying code comments, generated by IoTBox, makes it simple to determine the events that led to the smart home automatically unlocking the app. Also, IoTBox can present all paths that lead to a given action from a given event, providing a visualization identical to the graph shown in Figure 4. In this case, our hypothetical user may inspect the security policy and will be surprised that toggling a particular switch will always trigger the unlocking

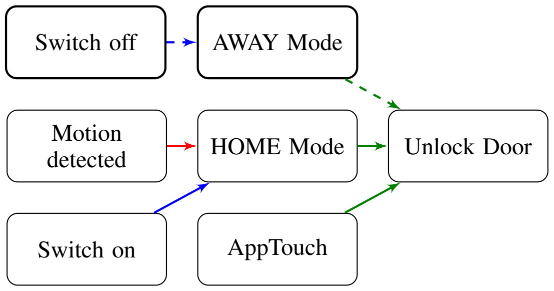

Fig. 4. An undesired path (from the interaction of App2 and App3 in dotted lines) to unlock the door is already in the smart home before the exploration phase. Undesired paths may not be easily noticed, but IoTBox helps detect such paths through its interpretable rules.

of the door. Thus, the user will be able to identify existing undesired behavior with the help of IoTBox.

```
assert {
  // if the lock is automatically unlocked,
  // it is caused by ...
  no r : IoTApp.rules, action : r.commands {
    action.attribute = lock
    action.value     = unlock
    (some predecessor : r.*(~connected),
        action' : predecessor.triggers {
      not {
       ...// omitted code
        // the switch turning on OR off
        action'.attribute = switch
        (action'.value = switch_on
           or
        action'.value = switch_off)
      }
    )
  }
}
```

Fig. 5. Excerpt of the Alloy assertion revealing possible triggers.

IoTBox allows malicious behaviors to be executed provided that they were encoded in the execution context. While this seems to be a limitation, this is, in fact, a strength of sandbox mining. Writers of malicious apps are forced to “disclose-or-die” [9], as any malicious behavior must be made explicitly detectable for analysis. Otherwise, the sandbox will prevent its execution. If so, a user of an IoT environment can detect the malicious behavior through inspection of the assertion and the visualization of paths leading to an action, both produced by IoTBox, to locate the undesired behavior.

B. Limitations and Tradeoffs

The primary difference between IoTBox and existing techniques that mine sandboxes is its use of models produced from static analysis, instead of using test generation. Fundamentally, the previous techniques use *test complement exclusion* [17]. This relies on the incompleteness of test case generation; test generation cannot explore all possible behaviors. If only normal behaviors were observed from testing during the exploration phase, then it guarantees that the sandbox allows only normal behaviors to run. IoTBox relies on a variant of this guarantee; only behaviors captured in the formal model can be executed. Conversely, *potentially dangerous behaviors that were not analyzed are disallowed from running*.

We do not learn from normal executions but include all behaviors that were abstracted into an app's model. By doing so, we gain the advantage of fewer false alarms as more behaviors are covered compared to test generation. On the other hand, static analysis usually overapproximates possible behaviors, introducing the possibility of including malicious behavior in a model. This is mitigated in IoTBox through non-opaque rules that can be interpreted by human users.

C. Threats to Validity

We mitigate threats to internal validity by relying on the models and tool used by other researchers [6], with a formalism of an IoT app (abstracted into triggers, conditions, and commands) that is similar to that of other studies [3]–[5], [16]. In some cases, we find that the Alloy models produced by IoTCOM are not directly usable (e.g. as they do not compile), so we modified them. Our models are publicly available [34].

To minimize threats to construct validity, we have used the evaluation metrics from previous studies [10], [11]. We evaluated the risk of false negatives, similar to Jamrozik et al. [9]. Moreover, we studied the same flawed IoT apps from a previous study [6], and many of these apps have also been studied in other works [3], [5], [16], [35]. We studied 20 bundles of apps, similar to prior studies. Jamrozik et al. [9] and Le et al. [36] studied 13 and 25 Android apps, and Wan et al. [12] studied 8 Linux containers.

A threat to external validity is that our experiments focuses only on the SmartThings and IFTTT platforms. Other smart home platforms include Apple's HomeKit [37], Google Home [38], Zapier [39], and Home Assistant [40]. However, as SmartThings support more devices than competing platforms [13] and IFTTT has over 11 milion users, we expect our findings to generalize. Our process of instrumenting apps is limited only to the SmartThings platform.

VI. Related Work

Mining sandboxes. Compared to existing studies on mining sandboxes [9]–[12], IoTBox does not use test case generation due to the difficulties of generating comprehensive test cases for an entire IoT system. Instead, IoTBox explores a formal model of the smart home, identifying the execution context of an action through the counterexamples found by the Alloy Analyser [15]. This has the advantage of identifying only events that have a causal relationship with a given action through analysis of the behavioral rule graph.

Related to mining sandboxes, Acar et al. [41] suggest that the automatic generation of security policies may help to address the permission comprehension problem on Android. Provos [42] proposed learning policies for system calls in UNIX systems. IoTBox is the first approach to account for context that spans multiple apps in a smart home.

Threats in an IoT system. Researchers have studied threats at the application-level on IoT platforms [1], [3]–[7], [31], [43]–[46]. ContexIOT [16] considers the context of an app in isolation, missing out on threats spanning multiple apps. Some studies have shown the prevalence of incorrect behavior in IoT applications [6], [22], [45], [46]. Compared to IoTMon [1], Soteria [3], IoTGuard [5], HOMEGUARD [44], IoTSan [4], IoTCOM [6], iRuler [7], our work shares the goal of identifying malicious behavior from the interaction of apps, but different from these studies, has the objective of inferring rules that help to detect behavioral changes.

IoTBox and ProvThings [43] share similarities. Both trace the origins of events, traversing graphs of apps connected by how they trigger and influence the execution of one another. However, they have different goals. IoTBox mines rules for a sandbox to detect behavioral changes while ProvThings collects information for debugging.

Other aspects of IoT systems have been studied [20], [47]–[50]. Researchers studied the impact of platform compromise [20] and suggested the need to minimize potential damage from an adversary that has compromised an IoT platform. Various security aspects, including the misuse of devices for botnets [48], [51] and firmware security [49], [50], have been studied but have different goals from our work.

Other researchers have shown that developers face difficulties interpreting behaviors in IoT apps [19], [52]–[54]. IoTBox may help in debugging, as it can reveal unexpected changes in behavior. Researchers have suggested the need to simplify the configuration of privacy policies [55]. There are many concerns, such as privacy concerns of other users in the smart home [19]. Our work may find application in interpreting the automation in a smart home. Indeed, other researchers have pointed out the need for tools that present the effects of enabling a new app and to identify unforeseen consequences [56].

VII. Conclusion and Future Work

In this work, we show that handcrafted security policies for smart home may produce many false alarms and suggest the need to automate the specialization of these policies for a smart home. We propose IoTBox to mine sandboxes of IoT systems, which detects behavioral changes in IoT systems. A sandbox with rules mined from an existing smart home will produce few false positives. IoTBox produces rules that can be inspected by a human user. IoTBox identifies the complete execution context of any action, producing rules that can be enforced to detect both disallowed and missing actions.

IoTBox comprehensively explores the behaviors of a smart home to precisely identify rules. We evaluated IoTBox on app bundles containing flawed behavior, including incorrect behaviors caused by interactions between apps, that were studied in prior work. We find that it effectively detects changes that introduce malicious behavior. A replication package is available [34]. In the future, we hope to evaluate IoTBox on other IoT platforms.

Acknowledgment

This research was supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant from SIS at SMU.

References

[1] W. Ding and H. Hu, "On the safety of IoT device physical interaction control," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 832–846.

[2] E. Ronen and A. Shamir, "Extended functionality attacks on IoT devices: The case of smart lights," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 3–12.

[3] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 147–158.

[4] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "IoTSan: Fortifying the safety of IoT systems," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 191–203.

[5] Z. B. Celik, G. Tan, and P. D. McDaniel, "IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT." in *NDSS 2019*.

[6] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in IoT systems," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 272–285.

[7] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action IoT platforms," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1439–1453.

[8] Vacation Light Director. Accessed: 2020-10-10. [Online]. Available: https://community.smartthings.com/t/new-app-vacation-light-director/7230

[9] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 37–48.

[10] T.-D. B. Le and D. Lo, "Deep specification mining," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 106–117.

[11] L. Bao, T.-D. B. Le, and D. Lo, "Mining sandboxes: Are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 445–455.

[12] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for Linux containers," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 92–102.

[13] Samsung SmartThings. Accessed: 2020-10-10. [Online]. Available: https://www.smartthings.com/

[14] IFTTT. Accessed: 2020-10-10. [Online]. Available: https://ifttt.com/

[15] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[16] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. Unviersity, "ContexIoT: Towards providing contextual integrity to appified IoT platforms." in *NDSS*, 2017.

[17] A. Zeller, "Test complement exclusion: Guarantees from dynamic analysis," in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 1–2.

[18] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal analysis of fairness," in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1980, pp. 163–173.

[19] E. Zeng, S. Mare, and F. Roesner, "End user security and privacy concerns with smart homes," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, 2017, pp. 65–80.

[20] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action IoT platforms," in *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[21] IoTBench. Accessed: 2020-10-10. [Online]. Available: https://github.com/IoTBench/IoTBench-test-suite

[22] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what? controlling flows in IoT apps," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1102–1119.

[23] Illuminated Response to Unexpected Visitors. Accessed: 2020-10-10. [Online]. Available: https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/IlluminatedResponsetoUnexpectedVisitors.txt.groovy

[24] Lock Door after X minutes. Accessed: 2020-10-10. [Online]. Available: https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/LockDoorafterXminutes.txt.groovy

[25] Motion Mode Change. Accessed: 2020-10-10. [Online]. Available: https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/MotionModeChange.txt.groovy

[26] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 234–245.

[27] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[28] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[29] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[30] D. Arp, M. Spreitzenbarth, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." 2014.

[31] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 636–654.

[32] SmartThings Simulator. Accessed: 2020-10-10. [Online]. Available: https://graph.api.smartthings.com/ide/apps

[33] Unlock Door. Accessed: 2020-10-10. [Online]. Available: https://github.com/IoTBench/IoTBench-test-suite/blob/master/smartThings/smartThings-contexIoT/smartThings-contextIoT-official-and-third-party/Unlockdoor.txt.groovy

[34] IoTBox artifact website. [Online]. Available: https://github.com/iotboxdeveloper/iotbox

[35] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity IoT," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1687–1704.

[36] T.-D. B. Le, L. Bao, D. Lo, D. Gao, and L. Li, "Towards mining comprehensive Android sandboxes," in *2018 23rd International conference on engineering of complex computer systems (ICECCS)*. IEEE, 2018, pp. 51–60.

[37] Apple HomeKit. Accessed: 2020-10-10. [Online]. Available: https://www.apple.com/ios/home/

[38] Google Home. Accessed: 2020-10-10. [Online]. Available: https://store.google.com/sg/category/connected_home

[39] Zapier. Accessed: 2020-10-10. [Online]. Available: https://zapier.com/

[40] Home Assistant. Accessed: 2020-10-10. [Online]. Available: https://www.home-assistant.io/

[41] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *2016 IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 433–451.

[42] N. Provos, "Improving host security with system call policies."

[43] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *Network and Distributed Systems Symposium*, 2018.

[44] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 411–423.

[45] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, G. H. Xu, and S. Lu, "Understanding and automatically detecting conflicting interactions between smart home IoT applications," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[46] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 1501–1510.

[47] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based IoT deployments," in *2019 IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 1362–1380.

[48] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the Mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[49] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1099–1114.

[50] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing." in *NDSS*, 2018.

[51] https://securityledger.com. Report: Millions (and millions) of devices vulnerable in latest Mirai attacks.

[52] W. Brackenbury, A. Deora, J. Ritchey, J. Vallee, W. He, G. Wang, M. L. Littman, and B. Ur, "How users interpret bugs in trigger-action programming," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[53] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur, "AutoTap: synthesizing and repairing trigger-action programs using LTL properties," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 281–291.

[54] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2015, pp. 215–225.

[55] E. Zeng and F. Roesner, "Understanding and improving security and privacy in multi-user smart homes: a design exploration and in-home user study," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 159–176.

[56] W. He, J. Martinez, R. Padhi, L. Zhang, and B. Ur, "When smart devices are stupid: negative experiences using home smart devices," in *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2019, pp. 150–155.