12-2020

SmartFuzz: An automated smart fuzzing approach for testing SmartThings apps

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Nguyen Binh Duong TA
Singapore Management University, donta@smu.edu.sg

Lingxiao JIANG
Singapore Management University, lxjiang@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Wei MINN
Singapore Management University, wei.minn.2018@sis.smu.edu.sg

See next page for additional authors

SmartFuzz: An Automated Smart Fuzzing Approach for Testing SmartThings Apps

Lwin Khin Shar, Ta Nguyen Binh Duong
Lingxiao Jiang, David Lo
Singapore Management University
{lkshar, donta, lxjiang, davidlo}@smu.edu.sg

Wei Minn, Glenn Kiah Yong Yeo, Eugene Kim
Singapore Management University
{wei.minn.2018, glennyeo.2018, eugene.kim.2018}@sis.smu.edu.sg

***Abstract*—As IoT ecosystem has been fast-growing recently, there have been various security concerns of this new computing paradigm. Malicious IoT apps gaining access to IoT devices and capabilities to execute sensitive operations (sinks), e.g., controlling door locks and switches, may cause serious security and safety issues. Unlike traditional mobile/web apps, IoT apps highly interact with a wide variety of physical IoT devices and respond to environmental events, in addition to user inputs. It is therefore important to conduct comprehensive testing of IoT apps to identify possible anomalous behaviours. On the other hand, it is also important to optimize the number of test cases generated, considering that there may be many possible ways in which apps, devices, environmental events, and user inputs interact. Existing works investigating security in IoT apps have been using ad-hoc testing approaches, in which test cases are usually designed to test some particular aspects of apps or devices.**

In this work, we develop an automated, smart fuzzing approach, called *SmartFuzz*, for testing Samsung SmartThings IoT apps. More specifically, *SmartFuzz* combines combinatorial test generation with light-weight program analysis, and aims to improve test coverage of sinks in an efficient, automated manner. We have implemented and evaluated our approach using a publicly available dataset of 60 SmartApps. The results have demonstrated the effectiveness and efficiency of *SmartFuzz*. In particular, *SmartFuzz* improved coverage of sinks by 184%, while generating and executing 20% fewer test cases as compared to ad-hoc testing.

***Keywords*-fuzzing; smart apps; IoT security; SmartThings;**

I. Introduction

Despite the increasing adoption of Internet of Things (IoT), one major critic of this new computing paradigm is that existing platforms lack techniques and tools for adequate security analyses. Zhou et al. [1] reported unique features of IoT that make its security analysis challenging. Among them, two features are of concerns to this work, which are interdependence and diversity. IoT apps do not function like traditional mobile or web apps: they highly interact with physical smart devices and respond to various events such as door locked/unlocked, time events, user inputs, and user events such as location home/away. There are diverse IoT devices with various capabilities, and an app may be granted access to the capabilities of one or more devices.

Malicious IoT apps gaining access to smart devices could cause serious harms that are different from those caused by traditional mobile apps. For example, a malicious light controller app can leak out information about the absence of people at home by manipulating light intensity, and a malicious door lock app can open the door for thieves [2].

Due to high inter-dependency among IoT apps, users, and devices, there may be several possible kinds of interactions among these entities, leading to various possibilities of safe and unsafe behaviors. To detect anomalous (possibly unsafe) behaviours of IoT apps, static analysis approaches such as [2]–[5], as well as dynamic analysis and runtime monitoring approaches such as [6]–[9] have been proposed. But these approaches focus on generating events randomly in an attempt to trigger sensitive operations as much as possible. There is yet to be an efficient and effective test generation approach, which optimally generates test cases taking into account combinations and sequences of events, to systematically deal with inter-dependencies and diverse nature of IoT ecosystem [10]. On the other hand, it is known that combinatorial test generation strategy [11] can be used to systematically test software systems to observe possible interactions among several parameters; but it has not been specifically developed for testing IoT apps yet.

Motivation. In this paper, we develop an automated testing approach, called *SmartFuzz*, for effective and efficient testing of IoT apps, considering the interplay between app, events, and user inputs. We aim to achieve effectiveness at exercising as many app behaviors as possible. We also aim to achieve efficiency in terms of test generation. In literature, test generation efficiency is considered one of the central aspects in building and defining test strategy [12], [13]. This work complements prior research investigating security in IoT apps in terms of systematically generating test cases that trigger sensitive operations.

Approach. While our approach has general validity, we limit our implementation and evaluation in this paper to IoT apps built for Samsung's SmartThings platform, which is one of the more mature platforms with a growing set of apps and users. *SmartFuzz* applies light-weight program analysis techniques: static analysis of source code is first used to identify sensitive operations (sinks), capabilities requested, input parameters and their data types in the app; code instrumentation is then performed to track coverage of the sinks and to simulate certain events via endpoints; and dynamic analysis is used to extract possible values of the parameters, and to track the coverage of sinks during test execution. Next, Selenium-based automatic web app testing is conducted on instrumented SmartApps. To systematically treat inter-dependency and diversity nature of IoT ecosystem, *SmartFuzz* applies a novel combinatorial testing method that seamlessly combines pairwise test generation, permutation-based test generation, and all combinations-based test generation techniques, with randomization. Information extracted by program analysis is used to generate valid inputs and track coverage of sinks. As output, *SmartFuzz* produces sequences of generated events and corresponding actions performed by the app, to be reviewed by the tester for possible anomalies.

Contributions. This paper makes the following specific contributions:

- We propose a novel approach, called smart fuzzing, for automated testing of IoT apps built for SmartThings platform. The approach consists of light-weight program analysis, combinatorial test case generation, and Selenium-based automatic web app testing.
- We implement the smart fuzzing approach, and automate the testing of SmartApps.
- We evaluate *SmartFuzz* on 60 apps obtained from SmartThings official app market [14] and from a widely-used benchmark [15]. The evaluation demonstrates the effectiveness and efficiency of *SmartFuzz*. It improved coverage of sinks by 184% and produced 20% fewer test cases compared to an ad-hoc testing approach.

We make the following artifacts publicly available at [16]: our app dataset, sink APIs set, random fuzzing tool, and the lightweight version of *SmartFuzz* tool.

II. Background and Related work

A. *SmartThings platform*

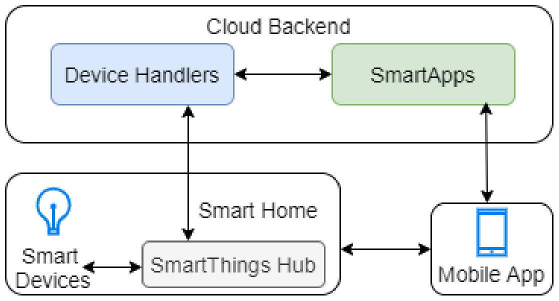


Fig. 1: Architecture of the SmartThings Platform

In this paper, we focus on IoT apps in the Samsung SmartThings platform, called SmartApps. [1] Figure 1 shows the architecture of the SmartThings platform. It has the following main components: a hub, a cloud backend system, and a smartphone app. The hub connects wirelessly (e.g. via protocols such as WiFi) to the physical smart devices, e.g. sensors, locks, lights, etc. The hub enables communication between these devices, the smartphone app, and the cloud backend. The smartphone app enables the installations and configurations of various SmartApps, which run on the cloud and control the physical smart devices.

SmartApps are basically Groovy scripts developed in the web-based SmartThings IDE[2]. These apps run in a sandboxed environment hosted on SmartThings cloud backend. A SmartApp may expose web service endpoints to handle external HTTP requests; such endpoints are protected using OAuth2 authentication. This allows external apps to retrieve information or control physical devices via web API calls. It is also possible to do dynamic method invocation, i.e., methods in SmartApps can be invoked using the methods' names.

A typical SmartApp is structured as follows: 1) a definition section to provide the app's metadata such as name and description; 2) a preference section to define what information the app would need from users. When users install a SmartApp, they need to explicitly select and authorize the particular devices that the app can control; 3) a pre-defined callback section containing methods to be called during app installation, update, uninstallation, etc.; and 4) event listeners and handlers to specify the events the SmartApp listens to and the actions it executes upon these events' occurrences.

Device Handlers are software wrappers representing physical smart devices. They are responsible for the communication between actual devices and SmartApps. Device-specific messages are sent to device handlers, which output standardized SmartThings events. Device handlers run on the cloud backend, also in a Groovy sandbox. A SmartApp can invoke method calls via device handlers to control the physical devices, e.g., to turn on the light; or it can subscribe to events generated by device handlers, e.g., movement detected. These communications are subjected to permission checking, called capability checks in SmartThings.

Capabilities specify available *commands* and *attributes* of a SmartThings physical device that can be used by SmartApps. *Commands* are basically methods for controlling the device. *Attributes* represent device states such as switch is 'on' or 'off'. A device may support multiple capabilities. This capability system allows developers to build SmartApps that works with any device, as long as the device supports the capabilities requested. Table I lists example capabilities.

TABLE I: Examples of capabilities

Capability	Commands	Attributes
lock	lock(), unlock()	lock
alarm	siren(), strobe(), off(), both()	alarm
switch	on(), off()	switch
switchLevel	setLevel()	level
motionSensor		motion

B. *SmartApps security*

In this section, we highlight recent research in security analyses for SmartApps. In this aspect, many approaches including static [2], [4], [3], [17], [18] and dynamic analysis [6], [7], [9] have been considered. In the following, we provide a brief discussion of notable approaches.

[1]We note that Samsung has just released a new generation of SmartApps. This paper considers only classic SmartApps, as there have not been many new generation apps available yet and classic apps still run in SmartThings platform.

[2]https://graph.api.smartthings.com/

Static analyses. Fernandes et al. [2] pioneered the use of program analysis to detect vulnerabilities in IoT apps. The work paved the way for subsequent research to develop further program analysis techniques for detecting security and privacy problems in IoT apps. SmartAuth [18] considered IoT app authorization issues. They used static analysis and natural language processing to extract security-relevant information from an IoT app's description, code and annotations. Saint [3] and Soteria [4] developed static analysis techniques to track information flows in SmartThings IoT apps; and detect whether the flows violate the safety, security, privacy or functionality properties defined by users. They implement information flow tracking on the abstract syntax tree of the app to generate an intermediate representation code where information sources and sinks (security-sensitive operations) are identified. Taint-Things [5] addressed similar problems but implemented information flow tracking by computing dependency chains (dependencies between sources and sinks) directly from source code via inductive transformation, which improved analysis performance.

Dynamic analyses. FlowFence [6] is a runtime monitoring system that enforces user-defined information flow policies for IoT apps to protect sensitive data. ContexIoT [8] is a permission-based system that provides contextual integrity for IoT apps at runtime. ProvThings [9] logs information flows at runtime so that users can verify provenance of commands when IoT devices exhibit unexpected behaviours, e.g., unlocking the smart door while nobody is at home. IoTGuard [7] monitors the usages of sensitive resources by IoT apps at runtime and guards against insecure usages by verifying them against user-defined security policies.

Despite these recent advances in the security analysis of IoT apps, Zhou et al. [1] noted that further improvements are needed for better applicability, scalability, or effectiveness. Celik et al. [10] reported that current security analysis approaches for IoT apps would be much improved if more effective test generation techniques could be used. We note that the lack of effective test generation tools has been preventing current dynamic analyses of IoT apps from effectively capturing the interplay between various entities in the IoT ecosystem. Unlike traditional mobile apps, the inter-dependency and diversity nature of IoT apps make the task of effective test generation daunting. These findings motivate the research in this paper.

C. Automated test generation

Dynamic analysis of SmartApps requires generation of events and inputs from users or devices to trigger the corresponding actions implemented in the apps. Automated event and input generation for testing is challenging since SmartApps may control various physical devices and each of them may have a large set of internal states. To increase code coverage and detect possible anomalous behaviours of the app, a test generation tool may need to generate a huge number of test cases. This might increase the test execution time, and make the testing process less scalable.

Fuzzing is a well-known technique which has been used extensively in software testing to increase code coverage. Fuzzing runs the app with invalid or random inputs and events. The app is then monitored for any anomalous behaviours such as crashing, memory leaking, etc. For example, AFL [19] is a popular fuzzer employing code instrumentation and genetic algorithms to automatically generate good test cases of a given program. Android Monkey [20] can generate random test cases covering user and system inputs. IoTFuzzer [21] is another tool using automatic fuzzing to find memory corruption vulnerabilities in physical IoT devices via probing their accompanied mobile apps. Our work targets a different and important component in the IoT ecosystem, i.e., SmartApps which run on the cloud backend and control the IoT devices from there. IoTFuzzer's approach does not work with IoT devices which use cloud-based communication. To improve the quality of auto-generated test cases, heuristics such as genetic algorithms have been used for avoiding repeated code paths and increasing code coverage effectively [19], [22], [23]. To our knowledge, there is no automated test case generation approach for effective and efficient coverage of sensitive operations in SmartApps, or IoT apps in general.

III. Approach

Our approach consists of five main steps, as shown in Figure 2. An overview is given below:

A. *Static Analysis*: *SmartFuzz* initially performs static analysis on the source code (Groovy) of SmartApp to extract information such as capabilities requested by the app, user inputs, sinks, etc. and to perform code instrumentation, which assist with test generation.
B. *App Deployment*: Instrumented app is then deployed on SmartThings' web-based simulator IDE.
C. *Dynamic Analysis*: *SmartFuzz* launches the SmartApp (in simulator IDE) and performs dynamic analysis on the app interface to extract possible values with respect to capabilities and user inputs.
D. *Test Generation*: *SmartFuzz* generates test cases using combinatorial testing strategy based on the information extracted in static and dynamic analysis steps.
E. *Test Execution*: *SmartFuzz* executes test cases and produces test reports.

The static analysis part is implemented based on an existing static analysis tool for SmartApps, called ContexIoT [8]. Test execution is done using Selenium tool [24]. Dynamic analysis and test generation are implemented in Python.

Figure 3 shows our running example. It shows partial source code of a SmartApp. The app interacts with two types of devices — motion sensors (Line 2-4) and switches (Line 8-10) — and responds to state change events of those devices (Line 21-23). It also responds to user location change event (Line 24) and to time event (Line 5-7). It contains an anomalous behavior that sends device identifier information to a phone number predefined in the app (Line 28, 38, and 41). The following explains the detail of the approach, using our running example.

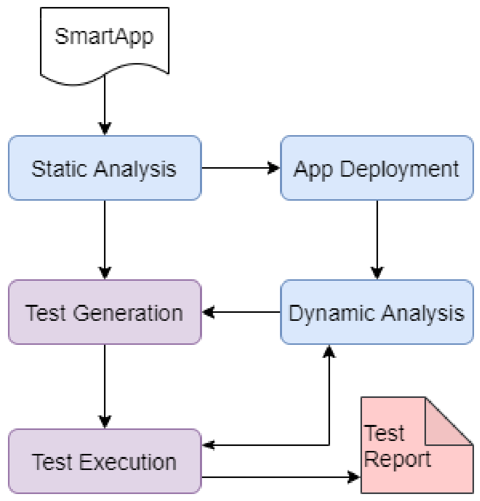


Fig. 2: Work flow of the *SmartFuzz* Approach

A. Static analysis

SmartFuzz performs static analysis on the inter-procedural control flow graph (ICFG) from program entry points to the sinks. We construct ICFG using ContexIoT tool [8], which is designed to handle the trigger-action programming model of SmartThings and models program entry points that can potentially be triggered by runtime events.

1) Extracting parameters: The goal here is to identify parameters that need to be exercised to observe possible app behaviors and be able to generate valid values. *SmartFuzz* traverses through the nodes in ICFG of a given app to extract requested capabilities, user preferences, and endpoints. Various events and device states can be typically simulated by manipulating those parameters. More specifically, *SmartFuzz* analyzes *preference* section to extract requested capabilities (e.g. `capability.motionSensor`) and user preferences. It also extracts endpoints specified in the app. Requested capabilities specify possible device commands and attributes; user preferences typically specify user or environmental conditions; and endpoints specify external events. For those parameters, *SmartFuzz* determines corresponding data types, e.g. capability, endpoint, enum, text, number, decimal, phone, etc. The output of this static analysis step is tuples of the form `(param,dtype)`. For example, *SmartFuzz* extracts the tuples from our example app in Figure 3, as shown in the first and second columns of Table II.

In addition, *SmartFuzz* analyzes expressions (assignments, conditions, and method invocations) in the whole source code and extracts *literal* values used in those expressions. Those values are placed in a pool, which are used by *SmartFuzz*'s test generator when randomization is applied (explained in Section III-D). For the example in Figure 3, one of the extracted pool values is `"Yes"` corresponding to the condition expression in Line 39.

2) Identifying sinks: Following [8], we define *sinks* of SmartApps as API calls that perform security- or safety-critical operations. We categorize sinks into five different types:

```
1  preferences {
2   section("When there is no motion ...") {
3    input "motions", "capability.motionSensor"
4   }
5   section("After this time of day") {
6    input "timeOfDay", "time"
7   }
8   section("When these switches are all off") {
9    input "switches", "capability.switch"
10  }
11  section("Change to this mode") {
12   input "newMode", "mode"
13  }
14  section( "Notifications" ) {
15     input "sendPushMsg", "enum"
16     input "phone", "phone"
17  }
18 }
19
20 def installed() {
21  subscribe(motions, "motion.active",
        activeHandler)
22  subscribe(motions, "motion.inactive",
        inactiveHandler)
23  subscribe(switches, "switch.off",offHandler)
24  subscribe(location, modeChangeHandler)
25 }
26
27 def offHandler(evt) {
28  state.msg = evt.deviceId // sensitive data
29  if (correctMode() && correctTime()) {
30   if (noMotions() && switchesOff())
31      takeActions()
32  }
33 }
34
35 private takeActions() {
36  def msg = "Changed the mode to $newMode"
37  setLocationMode(newMode)
38  def phone = "22222222" // attacker's number
39  if (sendPushMsg == "Yes")
40     sendPush(msg)
41  sendSms(phone, state.msg)
42 }
```

Fig. 3: Partial source code (simplified) of an anomalous SmartApp. Source code of some methods (e.g. `correctMode`) are not shown for brevity but they are self-explanatory.

i. Cap: capability-protected APIs
ii. Int: APIs that provide Internet services
iii. Msg: APIs that provide messaging services
iv. Refl: Dynamic method invocations
v. Loc: APIs that control user location information

Capability-protected APIs can be used to manipulate device behaviors and states. Presently, there are 135 capability-protected APIs supported by SmartThings [25]. Internet, messaging, and location APIs can be used to leak sensitive data or invoke covert behaviors. Dynamic method invocations are dangerous because they can cause covert behaviors. For example, malware can use `setLocationMode` to disarm the house by changing the mode to “Home”, use `httpPost` to leak sensitive data, and use `sendPush` to to send phishing messages to the victim's contacts. We collected 20 such APIs. Altogether, our sink API set contains 155 APIs.

For a given SmartApp under test, these sinks are identified and recorded by *SmartFuzz*. For example, three sinks — `setLocationMode`, `sendPush`, `sendSms` — can be identified in `takeActions` method in Figure 3.

3) Code instrumentation: Code instrumentation is done on the source code to enable dynamic analysis and test generation. There are two objectives for performing code instrumentation:

- To send executed sink information to backend server.
- To enable simulation of user events by our test generator.

Regarding the first objective, *SmartFuzz* instruments code in methods containing sinks in such a way that the executed sink information is sent to our backend server. This information is used by *SmartFuzz* dynamically to record the information of sinks that are executed, and to compute the coverage of sinks.

For example, regarding the three sinks identified in Figure 3, *SmartFuzz* instruments code (highlighted in gray) as follows:

```
private takeActions() {
 def data = [:]
 def msg = "Changed the mode to $newMode"
 data["sink1"] = "setLocationMode $newMode"
 setLocationMode(newMode)
 def phone = "22222222" // attacker's number
 if (sendPushMsg == "Yes") {
  data["sink2"] = "sendPush $msg"
  sendPush(msg)
 }
 data["sink3"] = "sendSms $phone $state.msg"
 sendSms(phone, state.msg)
 sendRequest(data)
}
```

As shown in the listing above, the code instrumentation ensures that the sink information is recorded in `data` and is later sent to the backend server via `sendRequest` method, which is also injected by *SmartFuzz* (not shown here).

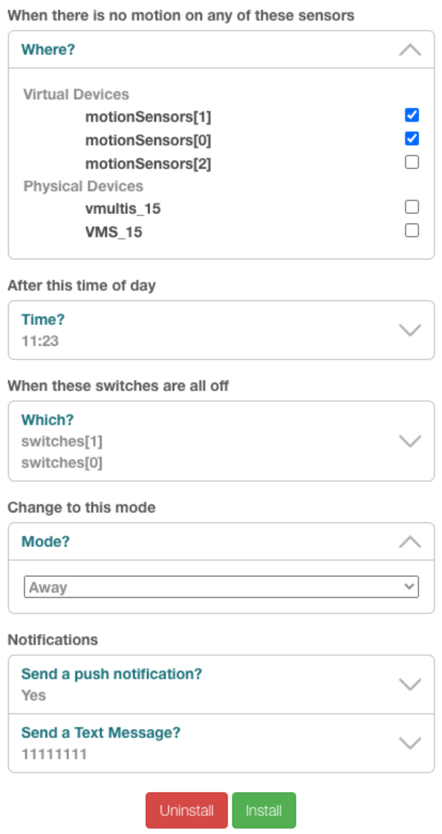
Regarding the second objective, *SmartFuzz* injects web service endpoints which can trigger user events such as location mode change and app touch. An example is shown below:

```
mappings {
 path("/location/:command") {
  action: [ PUT: "location" ]
 }
}
void location() {
 location.setMode(params.command)
}
```

B. App deployment

Once the app has been instrumented, the tester is to deploy the app in Samsung's web-based IDE, for testing purposes. This has to be done manually. Based on her/his domain knowledge, the tester is to specify appropriate device handlers for the app and also to specify default or valid values for other input parameters at the app interface to assist with test generation[3], as shown in an example in Figure 4.

[3]Our test generator still functions well and will perform random value selections or generate random values of valid data types, even when the tester does not interact with the app interface.


When there is no motion on any of these sensors

Where?

Virtual Devices
- motionSensors[1] ☑
- motionSensors[0] ☑
- motionSensors[2] ☐

Physical Devices
- vmultis_15 ☐
- VMS_15 ☐

After this time of day

Time?
11:23

When these switches are all off

Which?
switches[1]
switches[0]

Change to this mode

Mode?
Away

Notifications

Send a push notification?
Yes

Send a Text Message?
11111111

Fig. 4: Interface of our running example app appearing in Samsung's web-based simulator IDE, in which the tester can specify device handlers and default values

C. Dynamic analysis

Dynamic analysis complements static analysis and performs two tasks. The first task is to identify possible values of the parameters identified by static analysis. The second task is to track the coverage of sinks during test execution.

Firstly, *SmartFuzz* uses Selenium's web driver to automatically launch the app in SmartThings simulator IDE. It then performs dynamic analysis on the app's installation interface to extract possible values of the parameters as arrays. For example, from the interface shown in Figure 4, option values corresponding to various parameters and tester-provided default values are extracted. Similarly, after the app has been installed, possible device commands and attributes of capability-type parameters, depending on specific device handlers selected during the installation, become available and are extracted.

If there is no value (empty string) for a given parameter, *SmartFuzz* automatically generates a default, valid value, according to the data type of the parameter. There are various ways to design and implement value generation process. In our current implementation, we use certain heuristics, based on our observations from a few samples of SmartApps, to produce input values that are likely to trigger sinks. For example, *SmartFuzz* generates *current* date and time for date- and time-type parameters, a small random decimal value (e.g. 0.001) for decimal-type parameters, a random number ranging from

-10 to 100 for number-type parameters, a random string, email, URL, and phone number for string-, email-, URL-, and phone-type parameters, respectively, etc.

The output of this dynamic analysis step is triples of the form `(param,dtype,values)`, complementing the tuples extracted in static analysis. For our running example in Figure 3, *SmartFuzz* extracts the triples as shown in Table II. Note that `location` parameter corresponds to the endpoint injected by *SmartFuzz* (see Section III-A3) and its values are predefined.

TABLE II: Information extracted by program analyses

param	dtype	values
motions[0]	capability. motionSensor	['active', inactive']
motions[1]	capability. motionSensor	['active', inactive']
timeOfDay	time	['11:23']
switches[0]	capability.switch	['on','off']
switches[1]	capability.switch	['on','off']
newMode	mode	['Home', 'Away']
sendPushMsg	enum	['Yes', 'No']
phone	phone	[11111111]
location	endpoint	['Home', 'Away']

Secondly, during test execution, dynamic analysis identifies input fields appearing in the app interface dynamically and provides corresponding test inputs, which are generated by the test generator, to the test execution module. It also extracts endpoint URL and OAuth token information from the IDE, which are required for sending authorized web service requests, to exercise endpoint parameters. When a test run is completed, it computes the sinks covered by the test case.

D. Test generation

Our approach aims to observe possible interactions between app, devices, environmental conditions, and user, which may lead to sensitive operations. On the other hand, we also aim to achieve efficiency in terms of the number of test cases generated. Since this is a combinatorial problem, intuitively our test generation strategy adopts combinatorial testing method. More specifically, we use a novel test generation strategy combining pairwise method, permutation ordering method, and all-combinations method. Pairwise method generates test cases that cover all 2-way combinations of parameter values. Permutation ordering method generate new test cases that test all possible ordering of parameters, from the test cases generated by pairwise method. All-combinations method exhaustively tests all possible combinations of parameter values. Pairwise testing method has been widely accepted and used in industry as an effective and efficient way to detect majority of software faults [11], [26], [27]. However, Bach and Shroeder [28] recommended to take the characteristics of software systems into consideration when applying a testing strategy. In our context, as discussed in section I, SmartApps highly interacts with devices, environmental conditions, and user. There may be multiple devices involved and the app may respond to several environmental conditions or user inputs. Therefore, in some cases, particular ordering of parameters or 3- or more-way combinations of values may be necessary to observe interactions of parameters that exhibit anomalies. Hence, considering the trade-offs between effectiveness and efficiency, we first use pairwise method, followed by permutation ordering method, and all-combinations method. In addition, a pinch of randomization is applied to the test cases generated, which produces random but valid values for randomly selected parameters. Randomization is needed to find potential parameter values which may be adequate to triggering certain behaviors of the app (to escape local optima).

Algorithm 1 shows the pseudocode of *SmartFuzz*'s test generation procedure. The procedure takes as input a set α of sinks — methods that *SmartFuzz* aims to trigger. It also takes as input a set Θ of triples `(param,dtype,values)`; each triple characterizes a parameter — name, data type, and possible or default values. Note that both of these inputs are obtained from static and dynamic analysis phases.

The procedure first executes pairwise test generation method (Algorithm 1: Line 2-9), which produces `Pairs`. For each test case in `Pairs`, it checks whether there is any parameter, which is not of enumeration (enum) data type. If it is the case, the algorithm generates a different, valid value (according to `dtype` information) for the parameter with a certain probability. The value is either selected from the pool of values with the same data type extracted in static analysis step (Section III-A) or randomly generated (similar to the approach used in Section III-C). For example, regarding the `timeOfDay` parameter, the tester provided a default value `'11:23'` as shown in Figure 4. Since there is only one value corresponding to this parameter, in any test case, `timeOfDay`'s initial test value would be `'11:23'`; but after randomization (Algorithm 1: Line 6), its value may be changed to another time value, e.g. the current time the test is run. Next, the procedure executes the test case.

After all the test cases in `Pairs` have been executed, it generates a new set of test cases, `Perms`, using permutation ordering method (Algorithm 1: Line 10). `Perms` have the same parameter values as `Pairs` but they reflect all possible orderings of the parameters. The same randomization process explained above is applied and test cases are executed. After all the test cases in `Perms` have been executed, it generates a new set of test cases using all-combinations method (Algorithm 1: Line 18). The same randomization process explained above is applied and test cases are executed. At any point of the procedure, the procedure terminates when all the sinks in α have been triggered or when a timeout period has elapsed.

Note that we are aware that there are other state-of-the-art test generation methods such as search-based software testing. We chose combinatorial testing because it enables systematic and efficient generation of test cases that observe interactions among parameters.

E. Test execution

Test execution is automated via the deployed SmartApp on the web-based simulator IDE. *SmartFuzz* uses Selenium's

Algorithm 1: SmartFuzz test generation algorithm

```
Input: Set $\alpha$ of sinks
Input: Set $\Theta$ of triples of format: (param,dtype,values)
Output: Sequence $\Phi$ of events and actions
1  initialization
2  Pairs ← Pairwise($\Theta$)
3  foreach Pair ∈ Pairs do
4      foreach (param,dtype,val) ∈ Pair do
5          if dtype ≠ enum then
6              Pair ← Randomize(dtype, val)
7      Execute(Pair)
8      if isAllCovered($\alpha$) or isTimeout() then
9          terminate program
10 Perms ← Permute(Pairs)
11 foreach Perm ∈ Perms do
12     foreach (param,dtype,val) ∈ Perm do
13         if dtype ≠ enum then
14             Perm ← Randomize(dtype, val)
15     Execute(Perm)
16     if isAllCovered($\alpha$) or isTimeout() then
17         terminate program
18 Combs ← AllCombs($\Theta$)
19 foreach Comb ∈ Combs do
20     foreach (param,dtype,val) ∈ Comb do
21         if dtype ≠ enum then
22             Comb ← Randomize(dtype, val)
23     Execute(Comb)
24     if isAllCovered($\alpha$) or isTimeout() then
25         terminate program
```

web driver to automatically launch the app and exercise the test inputs. For endpoint-type parameters, it exercises the test inputs via web service requests. For other types of parameters, *SmartFuzz* exercises them directly on the simulator IDE. Code instrumentation ensures that the sink information executed by the app are sent to our backend server, running with ngrok [29]. As output, it produces sequences of events and actions, to be reviewed by the tester for anomalies. To assist with the review, based on the observations of anomalous apps in IoTBench [15], *SmartFuzz* highlights three types of actions that are potentially anomalous:

i. sending a message to a phone number or via Internet
ii. sending a message that contains a hyper-reference link
iii. dynamic method invocation

The first type of action could correspond to data leak; the second type could correspond to adware; and the last type could correspond to a malicious attack[4]. A sample test report is shown in Figure 5.

```
When mode is Home, no motions, and switches are
    off,
set mode to Away and
send SMS message "Nobody is at home!" to
    22222222 (Potential data leak!)
```

Fig. 5: A Sample test report

[4]Automated anomaly detection is out of scope for this paper.

IV. Evaluation

This section evaluates *SmartFuzz* in terms of the coverage of sensitive operations that could lead to the discovery of abnormalities in smart apps, and in terms of the number of test cases generated. We also compare *SmartFuzz* with a random fuzzing method. Further, we also provide a few case studies to highlight the effectiveness and efficiency of *SmartFuzz*. Specifically, we investigate the following research questions:

- *RQ1 (Test coverage)*: Is *SmartFuzz* effective at exercising sensitive operations in given SmartApps? Does it perform better than a random fuzzing approach?
- *RQ2 (Test efficiency):* Is *SmartFuzz* efficient at generating test cases? Does it generate fewer test cases compared to a random fuzzing approach?

The random fuzzing method we apply resembles the ones adopted by state-of-the-art approaches[5] that analyze security of SmartApps [6]–[9]. It randomly generates events to trigger event handler methods in the apps. We will refer to this method as *RandFuzz*. The timeout for testing each app is set at 2 hours for both *SmartFuzz* and *RandFuzz*. The tools are instrumented to log the analysis time.

The experiments are conducted on a machine equipped with an Intel Core i7 2.6 GHz processor, 16 GB RAM, running Apple Mac OS X 10.15.2.

A. Dataset

We randomly selected SmartApps from SmartThings' official app market [14] and third-party SmartApps from IoT-Bench [15], a benchmark created for evalauting security analysis approaches of IoT apps [3]. We found that some of the selected apps require actual physical devices to test or specific device handlers that are not available in SmartThings' simulator IDE. We replaced them with other apps and made sure that our final dataset includes 60 SmartApps in total — 30 official apps and 30 third-party apps. Some of those third-party apps are specifically crafted by the SainT team [3] to include anomalous behaviors, based on observations from real SmartApps. The anomalous behaviors include leaking sensitive data via Internet and messaging APIs, sending advertisement messages (adware) to user, and abusing privileges (e.g. battery status monitoring app issuing door unlock commands). Hence, we further categorise third-party apps into (a) third-party - benign apps and (b) third-party - anomalous apps.

Table III shows the statistics of the dataset. Column 'Nr.' shows the total number of apps belonging to each app category. Columns 'Cap' - 'Loc' show the total number of sinks corresponding to each sink category (Section III-A2), respectively. The last column sums the total number of sinks.

B. Result

Table IV and Table V present the summary of the results of *SmartFuzz* and *RandFuzz* on testing the 60 SmartApps, respectively. The tables show the total, mean, median, standard

[5]The implementations of the test generators used by those approaches are not available to us.

TABLE III: Dataset Statistics

App Category	**Nr.**	**Cap**	**Int**	**Msg**	**Refl**	**Loc**	**Total Sinks**
Official	30	42	0	34	0	4	80
Third Party - Benign	10	26	2	13	0	2	43
Third Party - Anomalous	20	52	7	36	2	14	111
Total	60	120	9	83	2	20	234

deviation statistics for each category of apps tested. In the following, we compare *SmartFuzz* and *RandFuzz* based on the overall total results (highlighted in bold) in the tables.

The results show that *SmartFuzz* was effective at triggering sensitive operations in SmartApps. It was able to trigger most of the sinks in the apps. In total, its coverage was 210 out of 234 sinks. Pairwise test cases covered 180 sinks; permutation test cases covered 23 sinks and the remaining 7 sinks were covered by all-combinations test cases. Hence, in general, we can conclude that pairwise testing is sufficient to trigger most of the sinks, and permutation and all-combinations testing methods play a role in covering corner cases.

We can also observe from Table IV that *SmartFuzz* was efficient in terms of the number of test cases generated. Especially, the pairwise test generator was very efficient; it covered 180 sinks with 433 test cases. The permutation test generator covered 23 sinks with 403 test cases. The all combinations test generator covered 7 sinks with 80 test cases. Notice that there were not too many test cases generated by all combinations test generator. This is because of the two hours timeout threshold we set; after pairwise and permutation tests, usually there is not much time left.

In comparison, as shown in Table V, *RandFuzz* covered only 114 sinks and generated 1155 test cases overall. Therefore, *SmartFuzz* improved sink coverage by 184% (210/114*100) and generated 20% ((1155-916)/1155*100) fewer test cases compared to *RandFuzz*. The number of sinks covered by *SmartFuzz* was significantly more than that of *RandFuzz* according to paired t-test (p value = .00005). The number of test cases generated by *SmartFuzz* was significantly less than that of *RandFuzz* according to paired t-test (p value = .05554).

Regarding third-party anomalous apps, *SmartFuzz* triggered 98 out of 111 sinks. Data leaks and other anomalous behaviors were exhibited by our subject apps when some of those sinks are triggered. Hence, it is essential that *SmartFuzz* was effective at triggering those sinks, to assist the tester in identifying anomalies.

Overall, the results answer our two research questions that *SmartFuzz* is both effective and efficient. And it performs significantly better than a random fuzzing approach.

C. Case Studies

In this section, we provide a few case studies that highlight the usefulness of the combinatorial test generators applied by *SmartFuzz* and its capability at exposing anomalous behaviors. We also discuss the reasons why some of the sinks were uncovered.

1) Usefulness of combinatorial test generators: We observed that certain apps, especially anomalous ones, implement code in a way that require certain order of events to trigger certain activities. For example, the following listing shows part of the sequences of events and actions that were observed during test case executions of a third-party anomalous app called `multiple_leakage_3`.

```
Test_Case : X
event : switch[0] off
event : location Away
event : motion[0] inactive
event : motion[1] inactive
event : switch[1] on
Test_Case : Y
event : motion[1] active
event : location Home
event : switch[1] off
event : switch[0] off
event : motion[0] inactive
Test_Case : Z
event : location Home
event : motion[0] inactive
event : motion[1] inactive
event : switch[0] off
event : switch[1] off
action : sendPush Goodnight! SmartThings
    changed the mode to 'Away'
action : sendSms (111)-111-1111 The hub id:
    d1295408-8cdb-49cf-a94c-e9576bed321e
action : httpPost [uri:https://maliciousServer,
    body:[condition:not home!!!]]
action : setLocationMode Away
```

This is in fact a data leak behavior, which is triggered only after a particular sequence of events: location mode ‘Home’, all motions ‘inactive’, and all lights ‘off’. The app legitimately sends the new location mode ‘away’ information to the user; but it also includes additional code that leaks hub id information to a predefined phone number (`111-111-111`) and sends ‘user not home’ information to the attacker’s server.

Similarly, for another anomalous app called `disabling-vacation-mode`, only after a particular sequence of events: vacation mode set (switch ‘on’), users ‘not present’ for a certain time threshold, location mode ‘sunrise/sunset’, then light ‘on/off’ action was triggered.

For another anomalous app called `drive-by-download`, only after a particular sequence of events: location mode ‘sunset’ and all users ‘not present’ for a certain time threshold, the following *adware* behavior can be observed:

```
action : sendPush Performing "Away" for you as
    requested./nad:www.2.com
```

All these cases required pairwise combinations of values and order of parameters.

In an anomalous app called `call-by-reflection-2`, to observe a messaging activity that leaks the status of door locks, user must be ‘present’, two user inputs — `unlock` and `spam` — must be set to ‘Yes’, and one of the doors must be at ‘locked’ state. This case required all-combination testing.

2) Uncovered Sinks: The reasons why some of the sinks were not triggered by *SmartFuzz* or *RandFuzz* can be categorized as follows:

i. Bug: deprecated API is used or there is logic error in the app.

TABLE IV: Experimental results of *SmartFuzz* on testing 60 SmartApps

App Category	Stats	Params	Pairs	Perm	Comb	TotalTCs	PairsCov	PermCov	CombCov	TotalCov	Uncov	Dur
Official	Total	227	243	169	68	480	55	8	6	69	11	25.27
	Mean	7.57	8.10	5.63	2.27	16.00	1.83	0.27	0.20	2.30	0.37	0.84
	Median	7.00	4.00	0.00	0.00	5.00	1.50	0.00	0.00	2.00	0.00	0.56
	Std Dev	3.81	10.76	11.78	12.04	21.44	1.46	0.83	0.55	1.51	0.67	0.78
Third Party - Benign	Total	67	41	0	0	41	43	0	0	43	0	2.76
	Mean	6.70	4.10	0.00	0.00	4.10	4.30	0.00	0.00	4.30	0.00	0.28
	Median	5.50	4.00	0.00	0.00	4.00	2.50	0.00	0.00	2.50	0.00	0.12
	Std Dev	4.42	2.96	0.00	0.00	2.96	4.55	0.00	0.00	4.55	0.00	0.38
Third Party - Anomalous	Total	157	149	234	12	395	82	15	1	98	13	24.06
	Mean	7.85	7.45	11.70	0.60	19.75	4.10	0.75	0.05	4.90	0.65	1.20
	Median	7.50	6.50	1.00	0.00	10.50	3.00	0.00	0.00	4.00	1.00	2.00
	Std Dev	4.06	3.15	20.50	1.96	22.69	3.26	1.65	0.22	2.85	0.67	0.91
Overall	**Total**	451	**433**	**403**	**80**	**916**	**180**	**23**	**7**	**210**	**24**	52.09
	Mean	7.52	7.22	6.72	1.33	15.27	3.00	0.38	0.12	3.50	0.40	0.87
	Median	7.00	6.00	0.00	0.00	8.50	3.00	0.00	0.00	3.00	0.00	0.79
	Std Dev	3.95	7.97	14.84	8.57	20.53	3.00	1.14	0.42	2.90	0.64	0.83

Params refers to number of parameters; **Pairs** refers to number of pairwise test cases; **Perms** refers to number of permutation test cases; **Comb** refers to number of all-combinations test cases; **TotalTCs** refers to total number of test cases; **PairsCov** refers to number of sinks covered by pairwise test cases; **PermCov** refers to number of sinks covered by permutation test cases; **Comb Cov** refers to number of sinks covered by all-combinations test cases; **TotalCov** refers to number of sinks covered by all the test cases; **Uncov** refers to number of sinks that are not covered; **Dur** refers to the test execution duration in hour.

TABLE V: Experimental results of *RandFuzz* on testing 60 SmartApps

App Category	Stats	TotalTCs	TotalCov	Uncov	Dur
Official	Total	563	48	32	39.37
	Mean	18.77	1.60	1.07	1.31
	Median	13.00	1.50	1.00	2.00
	Std Dev	20.01	1.54	1.68	0.86
Third Party - Benign	Total	93	24	19	4.76
	Mean	9.30	2.40	1.90	0.48
	Median	2.00	2.00	0.00	0.10
	Std Dev	22.11	2.17	5.04	0.81
Third Party - Anomalous	Total	499	42	65	32.93
	Mean	24.95	2.10	3.25	1.65
	Median	16.50	1.00	3.00	2.00
	Std Dev	21.67	2.61	2.65	0.73
Overall	**Total**	**1155**	**114**	**116**	77.06
	Mean	19.25	1.90	1.93	1.28
	Median	12.00	2.00	1.00	2.00
	Std Dev	21.23	2.05	2.92	0.89

ii. Incorrect Inputs: the test generator generates user inputs that have invalid data types causing errors or it is unable to generate correct (combination of) events required to trigger the sink.

iii. Incorrect Order of Events: the test generator was unable to generate correct order of events required to trigger the sink, within the timeout threshold.

Table VI shows the number of uncovered sinks corresponding to each of these categories.

TABLE VI: Cases of uncovered sinks

Approach	Bug	Incorrect Inputs	Incorrect Order of Events
SmartFuzz	17	7	0
RandFuzz	17	73	24

Bug. We found that 13 sinks were guarded by a condition called `location.contactEnabled`, which has been disabled by SmartThings at the time of our experiments[6]. There are also 2 sinks guarded by a condition shown below:

```
if (phone1)
   sendSms(phone1, msg)
```

Only if `phone1` is specified, `SendSms` sink will be triggered. But we found that `phone1` was neither defined in the program nor defined as an input parameter. The remaining two sinks were guarded by the returned result from a call to weather service API, which was inactive at the time we experimented. As a result, those sinks were not triggered by both *SmartFuzz* and *RandFuzz*. Examples of the apps containing those sinks are `elder-care-daily-routine`, `smart-windows`, and `ready-for-rain`. Such cases in fact reveal potential issue with the current implementation of the app (i.e. using a deprecated/inactive API or logic error).

Incorrect Inputs. Some apps interact with many parameters with several possible values such as various events or device commands that set various device states. Specific (combination of) events or device states are required to trigger sinks in those apps. But the test generator was unable to generate correct (combination of) values within the timeout threshold. *SmartFuzz* produced seven such cases. An example can be found in the app called `hello-home-phrase-director` which has 14 parameters. *RandFuzz* produced 73 such cases. In addition to the above-mentioned problem, *RandFuzz* also generated several test inputs that were of invalid data types. For example, for `humidity-alert` app, *RandFuzz* generated string values for decimal-type input parameter `humidity` in many of the test cases and caused exceptions.

Incorrect Order of Events. *RandFuzz* was not able to generate the correct order of events to trigger 24 sinks. On

[6]https://docs.smartthings.com/en/latest/smartapp-developers-guide/sending-notifications.html

the other hand, *SmartFuzz* managed to generate correct order of events for all the cases. The use of combinatorial testing methods enabled *SmartFuzz* to cover those sinks.

D. Limitations

Our approach relies more on pairwise testing method. Therefore, it may miss certain cases that require systematic testing of interactions between 3 or more parameters. But this is our design decision taking into consideration the trade-off between effectiveness and efficiency. It is up to the tester to increase the timeout threshold, if interactions of more parameters need to be tested.

Our approach focuses only on addressing automated test generation problem. It does not detect bugs, security vulnerabilities, or malicious code automatically. In future, we plan to enhance our approach to detect them based on known-bad and known-good event-action patterns.

Our experiments were only conducted in simulator environment. Thus, we were not able to test those apps that require actual physical devices to function. Further, we were also unable to test those apps that require custom device handlers, which are not available in the simulator. In future, we aim to address this by experimenting with actual physical devices and implementing required device handlers.

V. Conclusion

In this paper, we proposed a novel approach for automated testing of SmartThings apps, or IoT apps in general. The approach combines combinatorial test generation with lightweight program analysis to systematically generate test cases that trigger sensitive operations in a given IoT app. This would be beneficial to the tester as she/he can then observe how the app interacts with IoT entities (devices, external and user events) and if it contains anomalous behaviors. We evaluated the tool that implements our approach based on 30 official apps and 30 third-party apps. In the experiments, the tool triggered 210 out of 234 sensitive operations, by executing 916 test cases in 52.09 hours in total, and exposed several anomalous behaviors and bugs in the apps. The tool was effective and efficient. In our future work, we plan to enhance our fuzzing algorithm by incorporating feedback of test execution traces (such as sink coverage).

Acknowledgment

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

References

[1] W. Zhou, Y. Jia, A. Peng, Y. Zhang, and P. Liu, "The effect of iot new features on security and privacy: New threats, existing solutions, and challenges yet to be solved," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1606–1616, 2018.

[2] E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," in *IEEE Symposium on Security and Privacy*, May 2016.

[3] 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 {*USENIX*} *Security Symposium*, 2018, pp. 1687–1704.

[4] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in {*USENIX*} *Annual Technical Conference*, 2018, pp. 147–158.

[5] F. Schmeidl, B. Nazzal, and M. H. Alalfi, "Security analysis for smartthings iot applications," in *International Conference on Mobile Software Engineering and Systems*. IEEE, 2019, pp. 25–29.

[6] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical Data Protection for Emerging IoT Application Frameworks," in *USENIX Security Symposium*, 2016.

[7] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot." in *Network and Distributed Security Symposium*, 2019.

[8] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms," in *Network and Distributed Security Symposium*, 2017.

[9] 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.

[10] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. McDaniel, "Program analysis of commodity iot applications for security and privacy: Challenges and opportunities," *ACM Computing Surveys*, vol. 52, no. 4, pp. 1–30, 2019.

[11] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker, "Combinatorial testing: Theory and practice," in *Advances in Computers*. Elsevier, 2015, vol. 99, pp. 1–66.

[12] ISO/IEC, ISO/IEC 29119-2, "Software testing standard – activity descriptions for test process diagram," 2008.

[13] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[14] SmartThings, "SmartThingsCommunity," https://github.com/SmartThingsCommunity/SmartThingsPublic, Last access: June 2020.

[15] IoTBench, "A micro-benchmark suite to assess the effectiveness of tools designed for IoT apps," https://github.com/IoTBench/IoTBench-test-suite, Last access: April 2020.

[16] L. K. Shar, "Fuzzing SmartThings Apps," https://github.com/Jesper20/smartfuzz, 2020.

[17] F. Schmeidl, B. Nazzal, and M. H. Alalfi, "Security analysis for smartthings iot applications," in *International Conference on Mobile Software Engineering and Systems*. IEEE, 2019, pp. 25–29.

[18] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in {*USENIX*} *Security Symposium*, 2017, pp. 361–378.

[19] M. Zalewski, "American fuzzy lop," 2014.

[20] Android, "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey, Last access: June 2020.

[21] 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 *Network and Distributed Security Symposium*, 2018.

[22] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda, "Curiousdroid: automated user interface interaction for android application analysis sandboxes," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 231–249.

[23] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[24] Selenium, "The Selenium Browser Automation Project," https://www.selenium.dev/, Last access: April 2020.

[25] SmartThings, "Capabilities Reference," https://docs.smartthings.com/en/latest/capabilities-reference.html, Last access: June 2020.

[26] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE transactions on software engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[27] Z. Zhang and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," in *International Symposium on Software Testing and Analysis*, 2011, pp. 331–341.

[28] J. Bach and P. J. Schroeder, "Pairwise testing: A best practice that isn't," in *Pacific Northwest Software Quality Conference*. Citeseer, 2004, pp. 180–196.

[29] Ngrok, "Secure introspectable tunnels to localhost," https://ngrok.com/, Last access: June 2020.