WESTWORLD: Fuzzing-Assisted Remote Dynamic Symbolic Execution of Smart Apps on IoT Cloud Platforms

Lannan Luo
University of South Carolina
Columbia, USA
lluo@cse.sc.edu

Qiang Zeng
University of South Carolina
Columbia, USA
zeng1@cse.sc.edu

Bokai Yang
University of South Carolina
Columbia, USA
bokai@email.sc.edu

Fei Zuo
University of South Carolina
Columbia, USA
fzuo@email.sc.edu

Junzhe Wang
University of South Carolina
Columbia, USA
junzhe@cec.sc.edu

ABSTRACT

Existing symbolic execution typically assumes the analyzer can control the I/O environment and/or access the library code, which, however, is not the case when programs run on a remote proprietary execution environment managed by another party. For example, *SmartThings*, one of the most popular IoT platforms, is such a cloud-based execution environment. For programmers who write automation applications to be deployed on IoT cloud platforms, it raises significant challenges when they want to systematically test their code and find bugs. We propose *fuzzing-assisted remote dynamic symbolic execution*, which uses dynamic symbolic execution as backbone and utilizes fuzzing when necessary to automatically test programs running in a remote proprietary execution environment over which the analyzer has little control. As a case study, we enable it for analyzing smart apps running on SmartThings. We have developed a prototype and the evaluation shows that it is effective in testing smart apps and finding bugs.

ACM Reference Format:
Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. 2021. WESTWORLD: Fuzzing-Assisted Remote Dynamic Symbolic Execution of Smart Apps on IoT Cloud Platforms. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3485832.3488022

1 INTRODUCTION

The rapid proliferation of Internet-of-Things (IoT) devices has advanced the development of smart homes and factories. By installing *automation apps* (also called *IoT apps* or *smart apps*) on IoT platforms, users can integrate heterogeneous IoT devices for convenient automation. Popular IoT platforms include Samsung SmartThings, Amazon Alexa, and Google Home.

On platforms such as SmartThings, there is a lengthy process to *manually review* an official smart app [64], which is incomplete, error-prone and time-consuming. On the other hand, many developers enjoy writing custom smart apps and share them on the SmartThings community forum so that others can use them [62, 63, 70, 71], which however does not enforce code review, causing even more bugs to exist. This points to a critical need for *automated testing* of smart apps for bug discovery.

A promising testing technique is symbolic execution [34], an automated path exploration approach that is powerful for finding bugs. While many symbolic executors have been proposed for analyzing *Windows programs* [34], *Linux programs* [11, 12, 23, 49, 50] and *Java programs* [51, 52, 59], none support the analysis of IoT apps. Due to unique characteristics of IoT platforms, there are multiple challenges for symbolic execution analysis of IoT apps.

Challenge 1: Closed-source platform proprietary APIs. Existing classic symbolic execution often assumes the analyzer and the execution environment reside together locally, and the I/O environment and the API layer can be modeled conveniently. However, in the case of remote proprietary computing platforms, such as IoT cloud platforms, very often the assumptions are not valid.

IoT apps frequently interact with the platform by invoking APIs, e.g., to retrieve environment data (such as temperature and device status), which are *proprietary* with *no code* released to the public. For API calls, classical symbolic execution either sets the return variables as *new* symbolic inputs [54], causing imprecision, or applies function modeling [9, 32], which requires access to the API code or detailed documentation.

In order to analyze IoT apps running in a remote proprietary environment, we propose *remote dynamic symbolic execution (remote DSE)* to remotely and symbolically execute them. To enable remote DSE, our insight is that IoT platforms usually support logging and messaging [73], which makes information collection viable. Specifically, we leverage logging service to collect information needed for path exploration and send it back as messages to the local analyzer for making strategic exploration decisions.

Challenge 2: Achieving both precision and completeness of remote DSE. Although dynamic symbolic execution can recover from imprecision of classic symbolic execution caused by API calls, it often sacrifices *completeness*[1] for missing execution paths. While missing some paths might not be a big issue for large-sized programs, such as web applications [3, 5, 55] and Android apps [31, 86],

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488022

[1] *Completeness* here refers to full path coverage, following the definition in [13] and [23].

it has a significant impact on small-sized programs, such as IoT apps having a relatively small number of paths. *How to achieve both precision and completeness for analyzing IoT apps is challenging.*

To tackle this challenge, we propose ***selective code-segment fuzzing*** to assist DSE. It (i) automatically identifies the code snippet of a smart app that causes missing execution paths, and (ii) fuzzes only this part of code to complement symbolic execution. The result from selective code-segment fuzzing is combined with the symbolic path constraints from DSE to explore paths.[2] Our insight is that symbolic inputs (e.g., temperature, home mode, switch state) of smart apps usually have a small to moderate number of discrete values. Thus, selective code-segment fuzzing that fuzzes a code segment by iterating over values of the symbolic inputs is feasible (see Section 6). This is different from general programs, where their inputs usually have an infinite or huge number of possible values.

Challenge 3: Communication cost due to remote execution. As IoT apps run *remotely*, the request handling and communication cost between the local analyzer and remote cloud cannot be omitted. We propose ***boosted generational search***, which speeds up the analysis by wrapping multiple test inputs in one test request.

There are some other challenges. Smart apps frequently interact with the platform to retrieve environment data, which may be involved in conditional statements to determine whether a branch should be taken. However, *it is not allowed to vary environment settings on a platform once an app starts.* Our solution is to precisely identify variables storing environment data and set them as symbolic inputs. Moreover, smart apps have grammar peculiarities, such as closure [69]. *How to deal with the peculiarities is challenging.*

We have overcome these challenges, and implemented a system named Westworld, which enables fuzzing-assisted dynamic symbolic execution of IoT apps running on a remote platform. To make the work concrete, we showcase the ideas and techniques on SmartThings, one of the most popular IoT device integration platforms. The source code for Westworld and datasets are publicly available.[3] We evaluate Westworld with various experiments to measure its precision, completeness, efficiency, and effectiveness in bug finding. Our experiment results show that it is effective and efficient in testing IoT apps and finding bugs (e.g., *division by zero*, *array out of bound*, and *null-pointer dereference*). We made the following contributions.

- Being the first in the literature, we present a system that enables dynamic symbolic execution of IoT apps running on a remote platform, of which the API code is not available to the analyzer and program execution states cannot be cloned.
- *Selective code-segment fuzzing* is proposed to assist dynamic symbolic execution to effectively and precisely handle closed-source proprietary API calls. It captures a unique characteristic of smart apps, whose symbolic variables usually have a small number of discrete values.
- *Boosted generational search* is proposed to save the analysis cost and greatly increases the efficiency of remote DSE.
- We have implemented a prototype named Westworld, and demonstrate its efficiency and effectiveness in bug finding.

[2]Driller [67] adopts symbolic execution assisted fuzzing, while our work uses fuzzing-assisted symbolic execution (see Section 9).
[3]https://github.com/lannan/Westworld

2 BACKGROUND

2.1 SmartThings IoT Platforms

SmartThings is a proprietary platform owned by Samsung. It provides a software stack used to develop applications that monitor and control smart devices. SmartThings includes four main components: *hub*, *smart apps*, *smart devices*, and *cloud*. The hub bridges the communication between connected smart devices and the cloud, although WiFi-based IoT devices do not require a hub. Smart apps are developed in Groovy (a dynamic, object-oriented language) and run in the cloud. While SmartThings has recently started to support other languages, apps in Groovy are still the most popular and sophisticated ones, supported by different versions of SmartThings. SmartThings and third-party developers share their smart apps' code on GitHub [66] and community forum [65], respectively.

2.2 Symbolic Execution and Limitations

Classical Symbolic Execution. Symbolic execution is an analysis approach to program path exploration [40]. Input variables are represented using symbolic values. During path exploration, each path corresponds to a symbolic path condition, which is solved by a constraint solver to generate concrete inputs for the path.

Programs interact with the outside by calling library/system functions, whose code may not be available or contained path constraints cannot be resolved; such functions are called *uninterpreted functions* (see Section 3 of [13]). A key disadvantage of classical symbolic execution is that *it cannot generate accurate inputs when handling them* [13].

Limitations. For uninterpreted functions, classical symbolic execution either sets the return variables as *new* symbolic inputs [54], or applies function modeling [11].

- Smart apps frequently interact with the outside by invoking platform proprietary APIs, whose code is not available to the analyzer. If the return values of platform proprietary API calls are set as *new* symbolic inputs, the generated test cases will become imprecise, since the SMT solver does not know how to handle a constraint like $x == Fun(y)$.
- Function modeling needs precise understanding of the semantics of each API, but detailed documentation of the APIs used by smart apps is not available. Plus, it usually needs tedious manual effort.

Dynamic Symbolic Execution (DSE). There are two main categories [13]. (1) *Concolic testing* executes a program starting with some given inputs and gathers symbolic path constraints at conditional statements along execution. The collected path constraints are negated systematically or heuristically and solved with a constraint solver, yielding new inputs to exercise different paths.

(2) *Execution-generated testing* checks before every operation if the values involved are all concrete [11, 12, 23]. If so, the operation is executed concretely; otherwise, it is performed symbolically. If an API function is met, it solves the current path condition for a satisfying assignment, and uses the generated concrete values to invoke this API function and the following operations.[4]

[4]Function summaries are also frequently used to handle API calls (for speeding up symbolic execution) by encoding test results of an API using input preconditions and output postconditions [32]. This method requires access to API code (see Section 3.1 of [32]) and is thus not applicable in our case.

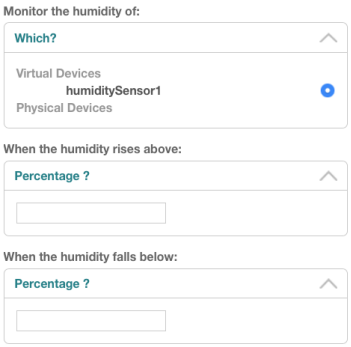


Figure 1: App configuration.

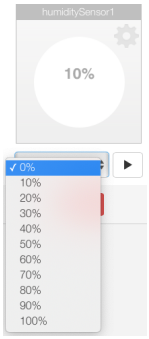


Figure 2: Env. input.

Limitations. DSE uses concrete values to simplify constraints and can generate inputs where classical symbolic execution gets stuck. But this comes with a caveat: *due to simplification, it sacrifices completeness and could miss some execution paths* [13]. While missing some paths might not be a big issue for large-sized programs, such as web applications [3, 5, 55] and Android apps [31, 86], it has a significant impact on small-sized programs like smart apps, where *completeness* becomes critical. Thus, *current DSE cannot be directly applied to smart apps* (see details in Section 3.2).

Our direction. One key observation is that execution-generated testing relies on cloning program execution states (or VM states [9]), which is infeasible on a proprietary platform. To enable remote DSE, we choose *concolic testing*, which does not have the requirement. This is a critical choice for enabling remote DSE of smart apps.

In order to solve the incompleteness issue of DSE (Section 2.2), we propose *selective code-segment fuzzing* to assist DSE for improving path coverage.

3 MOTIVATION

In this section, we first discuss the current method of testing smart apps, and then use a smart app to illustrate the challenges of analyzing smart apps, which motivate us to propose our approach.

3.1 Current Method of Testing Smart Apps

SmartThings provides developers with a *very primitive* web interface to the cloud-based execution environment. In order to test a smart app, a developer needs to first fill *app configurations* and then select *environment inputs* to submit a testing request.

Take Figure 1 and Figure 2 as an example. Figure 1 shows a web interface for filling app configurations, including selecting a humidity sensor and determining the range of the humidity level to turn on/off the humidifier. The app configurations will become variables in the app code and complete the app code's logic. Figure 2 shows a web interface for inputting the environment input—the current humidity level. Once the testing request is submitted, the environment input is converted into an event, which triggers the event handler that subscribes the event to execute. Finally, the developer reads logs to find bugs. It can be seen that it is troublesome to manually test smart apps using such a primitive web interface. Thus, automated app testing is critically needed.

3.2 Limitations of Current Concolic Testing

Figure 3 shows four examples, where the behavior of *sysAPI* is unknown and the code is unavailable. Current concolic testing may be able to achieve full path coverage for the first two, but it is difficult to achieve completeness for the latter two.

Example 1. Suppose that concolic testing starts with the initial input $x = 2$, where x is a symbolic input. In the first execution, the *true* branch of the first *if* statement (Line 2) and the *false* branch of the second *if* statement (Line 4) are taken. The collected symbolic path condition is $(x < 3) \wedge (x \geq 0)$. Concolic testing negates it and solves (i) $(x < 3) \wedge (x < 0)$ to get $x = -1$, and (ii) $x > 3$ to get another input $x = 4$, which executes different paths.

Example 2. Suppose the initial values of the symbolic inputs are $x = 2$ and $y = 4$. In the first execution, the *true* branch of the first *if* statement (Line 2) is taken. Assume the value of the return variable *ret* of `sysAPI` is 10. It then proceeds to explore the *true* branch of the second *if* statement (Line 4) and records the path constraint $y < 10$. To explore the *false* branch of the second *if* statement (Line 4), it keeps the value of x, but changes the value of y to make $y \geq 10$, which results in $x = 2$, and $y = 20$.

Example 3. In Example 2, the return variable of *sysAPI* is compared to a symbolic input, while here the return variable is compared to a constant value. Thus the solution above—i.e., by keeping the value of x and changing the value of y to reverse the result of $(y < ret)$—does not work. The *true* branch of the second *if* may be *missed* due to the difficulty in finding the value of x that satisfies the condition.

Example 4. The return value of *sysAPI* is determined by both x and y (Line 3). Thus, the solution in Example 2 does not work here either: if the value of y is changed, *ret*'s value may also be changed. As a result, a *path divergence* [33] occurs as the path constraint $y < ret$ becomes non-deterministic.

It is difficult for current concolic testing to achieve completeness in Examples 3 and 4. *Current concolic testing, if directly applied, could miss a large portion of execution paths of smart apps*, where completeness is critical for such small-sized programs.

3.3 A Motivating Example

Figure 4 shows a smart app *rise-and-shine*. The app needs the user to select a device to work as *motionSensor* and then fill the app configurations: specifying the values for *timeofDay*, *endTime*, and *timeAgo* (Lines 2-5). It then subscribes to a *motion* event with the value of *active*, and registers the `motionActiveHandler` method (Line 9), which is invoked when a *motion* event with the value of *active* is triggered (Line 12).

`motionActiveHandler` first calculates a time frame determined by *startTime* (influenced by *timeOfDay* and the current time t) and *terminTime* (influenced by *startTime* and *endTime*) (Lines 13-16). If the current time t is within the time frame (Line 18), it checks if the mode has been changed in the past time period determined by *timeAgo* (Lines 19-21). If not and the mode is not "Home", the mode is changed to "Home" (Lines 24-25). Here, the code of the platform APIs, e.g., `timeTodayAfter` and `eventsSince`, is unavailable.

Execution-generated testing (EGT) does not work. It is difficult to perform EGT for smart apps as cloning execution states is not feasible on IoT platforms (Section 2.2).

```
1 void foo (int x) {
2   if (x < 3) {
3     ret = sysAPI(x);
4     if (x < 0) { ... }
5 }}
```

(a) Example 1 ✓

```
1 void foo (int x, int y) {
2   if (x < 3) {
3     ret = sysAPI(x);
4     if (y < ret) { ... }
5 }}
```

(b) Example 2 ✓

```
1 void foo (int x) {
2   if (x < 3) {
3     ret = sysAPI(x);
4     if (ret == 20) { ... }
5 }}
```

(c) Example 3 ×

```
1 void foo (int x, int y) {
2   if (x < 3) {
3     ret = sysAPI(x, y);
4     if (y < ret) { ... }
5 }}
```

(d) Example 4 ×

Figure 3: Four code examples. ✓ indicates the examples for which concolic testing may achieve complete path coverage, while × shows the examples where it is difficult for concolic testing to achieve completeness.

```
1  preferences {
2    input "motionSensor", " capability .motionSensor"
3    input "timeOfDay", "time", title : "Start Time?"
4    input "endTime", "time", title : "End Time?"
5    input "timeAgo", "time", title : "Time Ago?"
6  }
7
8  def installed (){
9    subscribe (motionSensors, "motion. active ", motionActiveHandler)
10 }
11
12 def motionActiveHandler(evt){
13   def t = now()
14   def modeTime = new Date(t)
15   def startTime = timeTodayAfter(modeTime, timeOfDay)
16   def terminTime = timeTodayAfter(startTime , endTime)
17   //change mode during a specific time frame set by the user
18   if (t >= startTime .time && t <= terminTime.time) {
19     def pastTime = new Date(t - (1000 * 60 * timeAgo))
20     def evts = motionSensor.eventsSince (pastTime)
21     def alreadySet = evts .count{ it .value == " active "} > 0
22     def mode = location .mode
23     // let the mode be changed only once in the past time period
24     if (! alreadySet && mode!="Home")
25       setLocationMode("Home")
26 }}
```

Figure 4: An example smart app *rise-and-shine*.

Current concolic testing does not work. (1) Assume in the first execution, the *false* branch of the first *if* statement (Line 18) is taken. To take the *true* branch, one way is to keep the values of *timeOfDay* and *endTime* (such that the values of *startTime* and *terminTime* keep the same) and change the value of *t*. But by changing the value of *t*, a path divergence occurs (like Example 4) as *startTime* is influenced by *t*. (2) In the second *if* statement (Line 24), *alreadySet*, which is influenced by *timeAgo* and the return variable of *eventsSince* (Lines 19-21), is compared to a constant value *true*. It is unknown how to set *timeAgo* to explore both branches—similar to Example 3. Thus it is difficult for current concolic testing to achieve *full path coverage* when analyzing smart apps.

4 OVERVIEW

4.1 Design Goals

We have the following goals about our system WESTWORLD.

- *Precision.* The generated concrete inputs should take the same path as predicated by the system.
- *Completeness.* The system should achieve *high* path coverage, i.e., it can test (almost) all paths of a program.
- *Effectiveness.* The system should be applied to generating test cases and finding bugs in a smart app.
- *Efficiency.* It should not take a long time for the system to finish the symbolic execution of a smart app.

4.2 System Architecture

Figure 5 shows the architecture of WESTWORLD, consisting of three components: *Code Instrumentation*, *Web Interaction*, and *Path Analysis*. Given a smart app, WESTWORLD generates two types of instrumented apps: (1) a *PC-collection app* is the app instrumented with code that collects the symbolic *path condition* (PC) via remote DSE (Section 5); and (2) a *seg-fuzzing app* is the app instrumented by selective code-segment fuzzing to assist remote DSE to effectively handle platform APIs (Section 6).

In the first execution, random values are assigned to symbolic inputs, while, in the subsequent executions, the values of symbolic inputs are generated by the *Path Analysis* component. Specifically, the *Code Instrumentation* component first generates a *PC-collection app* based on the assigned values, and sends it to the *Web Interaction* component (①), which is built upon *Selenium* for automated testing of web application [24]. The *Web Interaction* component submits a testing request to run the app on the cloud. The runtime logs, returned as messages, are analyzed by the *Path Analysis* component, which first identifies the code segment(s) that need to be analyzed by *selective code-segment fuzzing* (②), and sends the code-segment(s) to the *Code Instrumentation* component (③) to generate one or more *seg-fuzzing apps* (④). The *Path Analysis* component then systematically negates the symbolic path condition obtained from the *PC-collection app* (⑤), and combines it with the results from the *seg-fuzzing app*(s) (⑥) to generate new test cases (⑦). The process is iterated until all the paths of the smart app are explored.

5 PATH-CONDITION COLLECTION

This section presents how to collect path conditions by leveraging code instrumentation and the readily available services (i.e., logging and messaging) provided by IoT platforms.

5.1 Identifying Entry Methods

A smart app does not have a *main* method. It declares entry points by subscribing to events. Each subscription includes a device name, a device event, and an event handler (Line 9 in Figure 4). The event handler methods are commonly used to take actions if the corresponding device events are received.

WESTWORLD analyzes all event subscriptions and finds their event handlers, which are considered as entry methods. Each entry method will be tested separately. It is worth noting that smart apps in SmartThings adopt event-driven programming and do not support multi-threading.

5.2 Identifying Symbolic Inputs

Symbolic inputs include user inputs and environment variables.

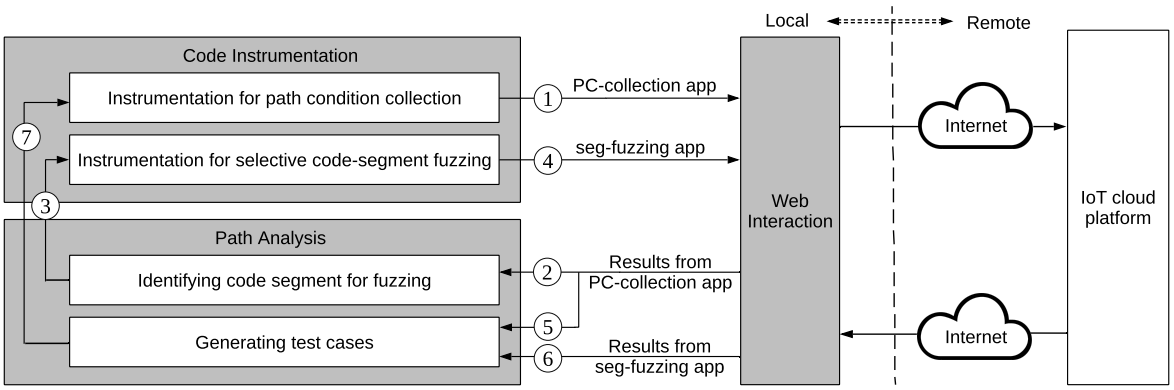


Figure 5: System architecture.

User Inputs. Smart apps often require user inputs (i.e., app configurations) to customize apps. User inputs are often used to form predicates that control device actions, and are set as symbolic. User inputs are declared by the keyword "input"(e.g., Lines 2–5 in Fig. 4).

Environment Variables. Smart apps frequently interact with the outside to retrieve environment data. It would be difficult to vary the environment on the remote platform. We resolve this issue by identifying variables that store environment data, and set them as symbolic inputs. They are categorized as follows.

- **Device state.** Each device object represents a physical device. The platform often defines *interfaces* to access device information (e.g., `getId()` returns the device id), or accesses the fields of a *device* object (e.g., *switch.currentSwitch* returns the state of the *switch* device). The return variables of the interfaces and the fields of a *device* object are set as symbolic.
- **Location.** The location information (such as timezone, longitude, and mode) can be accessed via the interfaces or fields of the *location* object. The fields and return variables of the interfaces are set as symbolic.
- **State.** Smart apps do not store data about their previous executions; instead, they persist and retrieve data across executions via the *state* object (designed as a map). We consider the *state* object and all its elements as symbolic.
- **Event.** Events have different types of information (such as description and value). The *event* fields and the return variables of the *event* interfaces are set as symbolic.

5.3 Instrumentation for PC Collection

5.3.1 Main Idea. We leverage code instrumentation to collect path information and use logging and messaging services to collect the information. The code instrumentor uses the Abstract Syntax Tree (AST) representation of the app code to *find different types of statements* for instrumentation.

First, it creates (i) a program state ϕ, which stores the values of concrete variables and symbolic expressions of symbolic variables, and (ii) a symbolic path condition *PC*, which is a quantifier-free first-order formula over symbolic variables. Then, ϕ is initialized to an empty map and *PC* to true. *Second*, the symbolic inputs are initialized using the values in the test case. *Third*, the code for updating ϕ and *PC* is inserted.

We apply lazy initialization [59] when a field of a symbolic variable is accessed. For a platform API, if at least one parameter is symbolic, we set the return variable as a *temporary symbolic variable (TSV)*, which will be taken care of by selective code-segment fuzzing (Section 6) to identify *its relation with symbolic inputs*.

Definition 1. *(**Temporary Symbolic Variable**) A temporary symbolic variable (TSV) is the return variable of a platform API call whose one or more parameters are symbolic.*

5.3.2 Instrumentation Example. We present how to instrument the app in Figure 4. The instrumented code is in Figure 6.

Identifying and replacing environment variables. We first scan the app code to find all environment variables; each is replaced by a local variable at the beginning of the entry method and each access is replaced by the variable. E.g., *location.mode* in Figure 4 at Line 22 is replaced by a local variable *env1* in Figure 6 at Line 12, and accessed via *env1* at Line 31.

Initializing symbolic inputs. (*a*) First, the `initSymbolicInputs` method is inserted (Line 43), which defines a map containing the concrete values, according to the current test case, of each symbolic input. It accepts a list of symbolic inputs and updates their values. (*b*) Then, it is called at the beginning of the entry method (Line 13).

Declaration expressions. For a declaration expression (e.g., *x=y+1*), the `UpdateProgramState` method is inserted (Line 32). If the right expression is symbolic, the declared variable is updated in ϕ using the symbolic expression; otherwise the concrete value is used.

Function calls. The `HandleFunCall` method is inserted after a function call. (*a*) If the function is user-defined, inter-procedural analysis is applied to instrument the callee function. (*b*) If the function is a platform API and at least one parameter is symbolic, the return variable is set as a *TSV*; otherwise it is directly executed. According to our investigation, the current time is usually compared to a time set by a user (a symbolic input); in this case, we consider the current time as concrete. In Figure 6, *t* and *modeTime* are stored concretely (Lines 14-17). As *timeTodayAfter* is a platform API and *timeOfDay* is symbolic, both *startTime* and *terminTime* (and their fields) are stored as TSVs (Lines 18-21).

If statements. `AddPC` is added for each branch to update *PC*.

for statements. (*a*) If all variables in the conditional statement are concrete, the *loop* is directly executed. (*b*) If at least one variable is

```
1  preferences {
2    input "motionSensor", " capability .motionSensor"
3    input "timeOfDay", "time", title : " Start Time?"
4    input "endTime", "time", title : "End Time?"
5    input "timeAgo", "time", title : "Time Ago?"
6  }
7  def installed () {
8    subscribe (motionSensors, "motion. active ", motionActiveHandler)
9  }
10
11 def motionActiveHandler(evt){
12   def env1, PC, φ
13   initSymbolicInputs ([PC, φ, timeOfDay, endTime, timeAgo, env1])
14   def t = now()
15   HandleFunCall(t, "now")
16   def modeTime = new Date(t)
17   HandleFunCall(modeTime, new Date(t))
18   def startTime = timeTodayAfter(modeTime, timeOfDay)
19   HandleFunCall(startTime, "timeTodayAfter", modeTime, timeOfDay)
20   def terminTime = timeTodayAfter(startTime, endTime)
21   HandleFunCall(terminTime, "timeTodayAfter", startTime, endTime)
22   //change mode during a specific time frame set by the user
23   if (t >= startTime.time && t <= terminTime.time) {
24     AddPC("(t >= startTime.time && t <= terminTime.time) == 1")
25     def pastTime = new Date(t - (1000 * 60 * timeAgo))
26     HandleFunCall(pastTime, "new Date", t, timeAgo)
27     def evts = motionSensor.eventsSince(pastTime)
28     HandleFunCall(evts, "eventsSince", motionSensor, pastTime)
29     def alreadySet = evts.count{it.value == "active"} > 0
30     HandleClosure(alreadySet, "evts.count{it.value=='active'}>0")
31     def mode = env1
32     UpdateProgState(mode, env1)
33     // let the mode be changed only once in the past time period
34     if (!alreadySet && mode!="Home") {
35       AddPC("(!alreadySet && mode!='Home')==1")
36       setLocationMode("Home")
37     } else
38       AddPC("(!alreadySet && mode!='Home')==0")
39   else
40     AddPC("(t >= startTime.time && t <= terminTime.time) == 0")
41 }
42
43 def initialSymbolicInputs (varList) {...}
```

Figure 6: Instrumentation (in *green*) for the app in Figure 4.

symbolic, existing approaches typically unroll the loop a fixed number of times [51, 59]. To achieve completeness, we apply selective code-segment fuzzing (Section 6).

Switch and while statements. Handling *switch* (resp. *while*) is similar to that of handling *if* (resp. *for*) statements.

Closure statements. Closure is a unique feature of Groovy. A closure is a code block that can take arguments, return a value, and be assigned to a variable. (*a*) It can be called as a method. We adopt the way of handling function calls to deal with it. (*b*) It can be used to iterate over all elements in a list, array, or map (e.g., Line 29 in Figure 6). We implement a script to automatically convert a closure to a *for-loop*, and apply the way of handling *for-loop* to handle it.

6 SELECTIVE CODE-SEGMENT FUZZING

We propose selective code-segment fuzzing, which assists remote DSE, to effectively handle platform proprietary API calls.

6.1 Motivation and Main Idea

Consider the example in Figure 7(a). Both x (an environment variable) and u (a user input) are symbolic inputs. `foo` is a user-defined

```
1 def eventHandler(evt) {
2   def x = getEnvVar()
3   def y = foo(x)
4   def O = sysAPI(y)
5   if (u > 3 && O.m == 8) {...}
6 }
```

(a) Original code.

```
1  def eventHandler(evt) {
2    def x = getEnvVar();
3    def A = getEnvVarValues("getEnvVar")
4    for (a in A) {
5      x = a
6      def y = foo(x)
7      def O = sysAPI(y)
8      Log.debug(a, O.m)
9      if (u > 3 && O.m == 8) {...}
10   }
11 }
```

(b) Instrumented code.

Figure 7: An example of selective code-segment fuzzing.

method and the return variable y is an object. As `sysAPI` is a platform API and y is symbolic, the return variable O is a TSV. To explore the *true* branch of Line 5, we need to solve $(u > 3) \wedge (O.m == 8)$; but it is unclear which value of x makes $O.m == 8$.

Our Insight. Most symbolic inputs (user inputs and environment variables, such as *temperature, humidity*, and *home modes*) of smart apps usually have a small to moderate number of possible values. E.g., "*humidity*" has 101 integer values between 0 and 100. This is different from general programs, where their inputs usually have an infinite or huge number of possible values.

Main Idea. Exploiting this uniqueness, we propose *selective code-segment fuzzing* to handle API calls that return TSVs. Note that if a TSV is an object, then its *field*—which is *primitive/string type*—will be involved in path conditions. For each (field of) TSV in the symbolic path condition, we identify the symbolic inputs it relies on, called *influential symbolic inputs (ISIs)*. E.g., in Figure 7(a), O is a TSV, and relies on x, where x is an ISI (highlighted).

Based on the ISIs, we determine a code segment in the app code, and create a *seg-fuzzing* app, where a *for-loop* is inserted surrounding the code segment to iterate over values of the ISIs and learn a *code-segment summary* expressed as the relation between the (field of) TSV and ISIs. For example, through fuzzing we find $((x == 1) \wedge (O.m == 6))||((x == 2) \wedge (O.m == 8))$, which is combined with the path condition $(u > 3) \wedge (O.m == 8)$, and solved together to get the values of the symbolic inputs u and x.

6.2 Learning Code-Segment Summary

To find a code-segment for inserting a *for-loop*, three points need to be determined. (1) A *start point* is the place where an identified ISI is *first accessed* among all; e.g., in Figure 7(a), the start point for $O.m$ is the place right before Line 3. (2) A *logging point* is the place right before the conditional statement where the (field of) TSV is involved; e.g., the place right before Line 5 in Figure 7(a) is the logging point for $O.m$. (3) An *end point* is an *immediate post-dominator* of both the start and logging points. In a control flow graph (CFG), a block m post-dominates a block n if every path from the entry block to n must go through m; and the immediate post-dominator of a block n is the block that does not post-dominate any other post-dominators of n [37]. The end point in Figure 7(a) is right before Line 6. We adopt Lengauer-Tarjan algorithm [42] to find the immediate post-dominator.

Examples. In Fig. 8, each node in the CFG represents a statement. In Fig. 8(a), assuming node 1 is the start point and node 2 the logging point, node 9 is the end point. Fig. 8(b) shows another example.

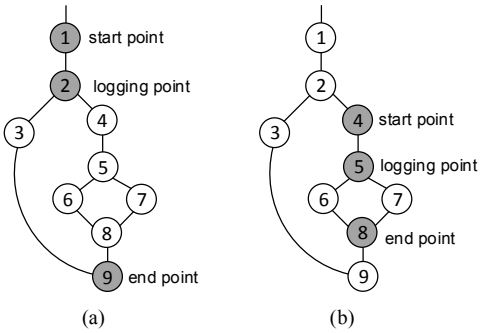


Figure 8: Examples of determining insertion points.

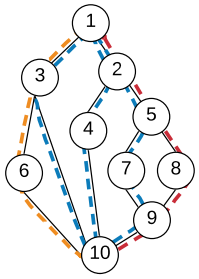


Figure 9: An example of *"generations"*.

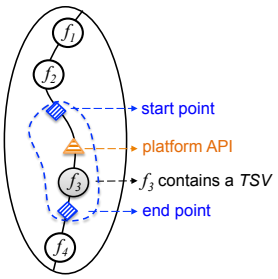


Figure 10: Combining path condition and code-segment summary.

Finally, given the example in Figure 7(a), a *for-loop* is inserted as shown in Figure 7(b). The `getEnvVarValues` method at Line 3 is a function for returning the collection of the values for each symbolic input. These values are obtained from a file (that we create and contains all possible values of each environment variable), and then included in the `getEnvVarValues` method.

Selective code-segment fuzzing can also handle *for* statements. If the conditional statement contains a symbolic input, we iterate over all possible values of the symbolic input. If it contains a (field of) TSV, we first find out the ISIs for the (field of) TSV and iterate over the possible values of these ISIs.

The fuzzing results can be reused. We store the code segments and summaries in a map (each variable name is replaced with an identical name), and check it before generating a *seg-fuzzing* app.

Why not Fuzz Platform APIs? We do not directly fuzz a platform API to learn the relation between the *return variable* and *parameters* for the following two reasons. (1) Given an API parameter, which is not a symbolic input, such as y at Line 3 in Figure 7(a), it is difficult to estimate its value range. (2) If the return variable is an *object*, it is difficult to record the values of all its fields, which are defined by the closed-source platform. E.g., in Figure 7(a), we do not fuzz `sysAPI` to find the relation between O and y (both are objects). Instead, we aim to find *the relation between O.m and x*, where x is a symbolic input that influences $O.m$ (that is, x is the ISI of $O.m$).

6.3 Feasibility Analysis

We analyze the feasibility of selective code-segment fuzzing.

Classification. We divide symbolic inputs into three categories.

(1) *Category-I*: The symbolic input has a few possible values (< 10); e.g., the location mode contains 3 values: *Home*, *Away*, and *Night*. We iterate over all of them.

(2) *Category-II*: The symbolic input contains many but not a large number of values ($\in [10, a]$, where $a \ll 2^{32}$); e.g., the humidity has 101 integer values. We select a subset as follows: i) all the values are divided into n equal parts, where m values are randomly picked from each part (Section 8.3 discusses how to determine n and m); and ii) if the variable is involved in a conditional statement and compared with a constant value, three more values (larger than, smaller than, and equal to the constant value) are selected.

(3) *Category-III*: The symbolic input contains a huge or infinite number of values, such as phone number and location id. Based on our evaluation, none is used as an ISI; e.g., a phone number is usually used as a parameter of *sendMessage*, and a location id is to form a message to describe the app.

Analysis. Assume an app has n symbolic inputs and m TSVs, where each symbolic input I_i has μ_i ($i \in [1, n]$) values and each TSV T_j has v_j ($j \in [1, m]$) ISIs. Then it requires $\sum_{j=1}^{m}(\prod_{i=1}^{v_j} \mu_i^{T_j})$ times for selective code-segment fuzzing to generate the required relations, where $\mu_i^{T_j}$ is the number of values of the i-th ISI for the TSV T_j.

According to our evaluation (Section 8.3), the maximum number of TSVs that an app contains is no more than 3 ($m \leq 3$), and the maximum number of ISIs for a TSV is no more than 3 ($v_j \leq 3$). Moreover, the ISIs have a small or moderate number of possible values ($\mu_i \in [1, a]$, where $a \ll 2^{32}$; belonging to Category-I or -II). Thus, our selective code-segment fuzzing is feasible.

In Figure 4, e.g., we select 10 values for each user input, *timeOfDay*, *endTime*, and *timeAgo*. This app has 3 TSVs. (i) *startTime.time* is influenced by *timeOfDay*, and it takes 10 times of loop iterations to obtain their relation. (ii) *terminTime.time* is influenced by *endTime* and *startTime*, and it takes 100 times. (iii) *alreadySet* is influenced by *timeAgo*, and it takes 10 times. Thus, it totally takes 120 times of loop iterations for the selective code-segment fuzzing.

7 REMOTE PATH EXPLORATION

The *Path Analysis* component is to i) identify TSVs from the symbolic path conditions, and ii) combine symbolic path condition and code-segment summaries to generate new test cases for path exploration. Below we present two path-exploration methods: i) *original* generational search [36] (we call it *vanilla* generational search), which is often used in concolic testing, and ii) *boosted* generational search proposed by us.

7.1 Vanilla Generational Search

We adopt the algorithm in [36] as our *vanilla* generational search. The algorithm systematically negates all path constraints in each symbolic path condition. We use Figure 9 to explain what a "*generation*" means. Assume the first execution exercises the path colored in red ($1 \rightarrow 2 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 10$). By negating each path constraint along this path (at nodes 1, 2 and 5), we generate three *1st-generation* children corresponding to the three paths colored in blue ($1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 9 \rightarrow 10$, $1 \rightarrow 2 \rightarrow 4 \rightarrow 10$, and $1 \rightarrow 3 \rightarrow 10$). By repeating this process, each first generation path can be further expanded to generate (zero or more) *2nd-generation* children, and so on. Here, the *2nd-generation* child corresponds to the path colored in yellow ($1 \rightarrow 3 \rightarrow 6 \rightarrow 10$).

In our work, symbolic path conditions and code-segment summaries are *combined* to generate new test cases. Let $F = \bigwedge_{i=1}^{n} f_i$ denote the symbolic path condition containing n path constraints. Assume m TSVs are involved. Let P_i ($i \in [1, m]$) denote the summary for the i-th TSV. Then F is systematically negated and combined with (a subset of) these P_i to generate test cases.

Special attention needs to be paid when combining F and these P_j: if a path constraint $f_s \in F$ is negated and the conditions involving some TSVs T_j ($j \in [t, m]$) appear *after* f_s in the app code, the summaries P_j ($j \in [t, m]$) are *not* combined. By solving the formula, $\bigwedge_{i=1}^{(s-1)} f_i \bigwedge \neg f_s \bigwedge_{j=1}^{(t-1)} P_j$, a new test case is generated.

An Example. In Figure 10, $F = \bigwedge_{i=1}^{4} f_i$. f_3 contains a TSV returned by a platform API. For the TSV, we identify a code segment (circled by the dashed line), and generate its summary P_1. By applying vanilla generational search, we obtain four formulas: (i) $\bigwedge_{i=1}^{3} f_i \bigwedge \neg f_4 \bigwedge P_1$, (ii) $\bigwedge_{i=1}^{2} f_i \bigwedge \neg f_3 \bigwedge P_1$, (iii) $f_1 \bigwedge \neg f_2$, and (iv) $\neg f_1$. By solving each formula, a new test case is generated.

In vanilla generational search, one *testing request* has to be submitted to the IoT platform to install a smart app (which is instrumented according to a generated test case) and explore a new path. While it is not an issue if the analyzer and execution environment reside together, it imposes large request-handling and communication costs on our analysis. Thus, *reducing the cost is critical.*

7.2 Boosted Generational Search

We propose *boosted generational search* that explores all paths belonging to one generation by submitting *only one* testing request. E.g., in Figure 9, the three first-generation paths in blue need *three testing requests* in vanilla generational search, but only *one* in boosted generational search.

Note that the proposed boosted generational search is not a new search strategy but a speed-up method for tackling the challenge of communication costs due to remote execution. We choose generational search over other search strategies because it maximizes the number of new input tests generated from each symbolic execution [35]. Boosted generational search, which is built upon vanilla generational search, wraps the multiple input tests from one symbolic execution into one remote testing request, such that the number of testing requests can be reduced.

Algorithm. In Algorithm 1, the method `BoostedSearch` first invokes `SymExePCs` to run the app with the initial inputs and collect the symbolic path condition, which is stored in a list δ and inserted into the working queue Q (Lines 2-4). Each element in Q is processed by `NextGenPCs` to generate the symbolic path conditions corresponding to the test cases of the *next generation* (Lines 5-8).

In `NextGenPCs`, it first generates the test cases of the next generation (Lines 11-21) and invokes `SymExePCs` to *execute all the test cases in one testing request* to collect the corresponding symbolic path conditions, which are stored in a list β (Line 23). Specifically, for each path condition *PC* in the input list ψ, it expands every constraint in *PC* (at a position i greater than or equal to a parameter called *PC.bound* initially 0 at Line 3). This is done by checking whether the formula ζ—combined from *PC* and the code-segment summaries—is satisfiable or not (Lines 13-15). If so, a next-generation test case s is found and inserted into `nextGenInputs` (Lines 16-19).

Algorithm 1 Boosted Generational Search.

```
1:  function BoostedSearch(input)
2:      δ = SymExePCs(appCode, input)
3:      δ.get(0).bound = 0          ▷ δ contains one element: the initial path condition
4:      enq(δ, Q)
5:      while Q is not empty do
6:          ψ ← deq(Q)
7:          enq(NextGenPCs(ψ), Q)
8:      end while
9:  end function

10: function NextGenPCs(ψ)
11:     for each PC of ψ do
12:         for i = PC.bound; i < |PC|; ++i do
13:             ρ ← PC[0...(i-1)] and ¬ PC[i]
14:             ζ ← ρ is combined with the fuzzing result
15:             s = ConstraintSolve(ζ)          ▷ check whether ζ is satisfiable
16:             if s is not NULL then
17:                 s.bound = i
18:                 nextGenInputs.insert(s)
19:             end if
20:         end for
21:     end for
22:                         ▷ execute all test cases of one generation in one test request
23:     β = SymExePCs(appCode, nextGenInputs)
24:     for each PC in β and each s in nextGenInputs do
25:         PC.bound = s.bound
26:     end for
27:     return β
28: end function
```

All the test cases in `nextGenInputs` are executed in *one* testing request, and the corresponding symbolic path conditions are collected (Line 23). Each symbolic path condition *PC*'s *bound* is assigned with the value of the corresponding test case s's *bound* (Lines 24-26). Finally, all *PCs* are returned (Line 27).

To execute all test cases of one generation through one testing request, we insert a `TestDriver` method that *wraps* multiple invocations of an entry method. Each invocation initializes the symbolic inputs separately. We register `TestDriver` as an entry method, and then remove the registration of the original one.

8 IMPLEMENTATION AND EVALUATION

8.1 Implementation

Our prototype consists of 4,418 lines of Groovy code and 2,444 lines of Java code. SmartThings provides a primitive web interface to testing apps, e.g., generating virtual events to trigger an entry method. We leverage the web interface to launch remote DSE. We use the constraint solver Z3 [27] to solve symbolic path conditions.

To automate testing, we build the *Web Interaction* component upon *Selenium* [24], which can automatically log in the platform, install an app, and generate events (by simulating human operations). E.g., for the app in Figure 4, the component generates both *motion.active* and *motion.inactive* events to trigger the method `motionActiveHandler`. To demonstrate the benefit of boosted generational search on improving the efficiency of testing smart apps, we implement two versions of Westworld: W-van uses vanilla generational search, and W-boost uses boosted generational search.

8.2 Experimental Settings

We evaluate Westworld in five aspects: *feasibility*, *completeness*, *precision*, *efficiency*, and *effectiveness in bug finding*. Our experiments

```
1  def luxHandler(evt) {
2    def val = evt.value
3    def light = lightSensor.latestValue("illuminance")
4    def motionState = motionSensor.currentState("motion")
5    def elapsed = now() - motionState.rawDateCreated.time
6    def threshold = 1000 * 60 * delayMins - 1000
7    def loc = getLocation()
8    def curMode = loc.getCurrentMode()
9    if (val == "active") {
10     // injected conditions
11     if (light <= tooDark && elapsed >= threshold && lockSensor.latestValue(
         "lock") =="locked" && curMode.name =="Home") {
12       lightSensor.on()
13     }
14     ...
15 }}
```

Figure 11: An example of injected conditions. Code related to the injected conditions is highlighted in green.

were conducted on a machine with an Intel Core i7-7700 CPU @ 3.60GHz with 16GB of RAM.

Datasets. (1) *Dataset-I* includes 136 official (84) and third-party (52) apps randomly collected from the SmartThings GitHub repo [15]. Note that many apps that contain only one path are not selected. (2) As many official and third-party apps contain a small number of paths, we create *Dataset-II* including 64 hand-crafted apps with more paths and more complex conditional statements. E.g., we purposely add more branch conditions to the original official and third-party apps. The injected conditions involve platform APIs, user inputs, and/or environment variables to make the path exploration more challenging. Note that it is not uncommon that users of SmartThings modify existing smart apps for their purposes [62, 63, 70]. Figure 11 shows an example of injected conditions, where part of the injected code is highlighted in green. The original smart app will turn on the light (Line 12) if motion is detected (Line 9). Four conditions are injected to control the light on (Line 11): i) whether the current illuminance is lower than a user input *tooDark*, ii) whether the motion is detected after a period of time *threshold* determine by a user input *delayMins*, iii) whether the status of the lock sensor (an environment variable) is locked, and iv) whether the current mode returned by the platform APIs, `getLocation()` and `getCurrentMode()`, is "Home". (3) *Dataset-III* includes 8 apps with different types of bugs inserted by us, which is used, together with *Dataset-I*, to demonstrate WESTWORLD's bug finding capability.

8.3 Feasibility

To demonstrate the feasibility of selective code-segment fuzzing, we seek to understand the impact of symbolic inputs on the number of fuzzing iterations. We use *Dataset-I*.

The number of symbolic inputs contained in each app. The results show that 1) the average numbers of user inputs and environment variables are 3.24 and 1.73, respectively; 2) the maximum numbers of user inputs and environment variables are 13 and 6, respectively; and 3) all apps have at least one user input and one environment variable. Thus, if environment variables are not set as symbolic but simply use the concrete values, it is unlikely to achieve high path coverage testing of smart apps.

The number of *TSVs*, and the number and categories of *ISIs*. We have the following findings. (a) The maximum number of TSVs that an app contains is 3. (b) The maximum number of ISIs (i.e., symbolic inputs that a TSV relies on) for any given TSV is 3. (c) All the ISIs belong to *Category-I* and *Category-II* (Section 6.3); e.g., the location id, belonging to *Category-III*, is only used to form a message about the app. We can conclude that WESTWORLD equipped with selective code-segment fuzzing is feasible for analyzing smart apps.

Table 1: App statistics of *Dataset I* and *Dataset II*.

# of paths in apps	# of apps in *Dataset-I*	# of apps in *Dataset-II*
≥ 20	16	15
[15, 20)	13	18
[10, 15)	18	13
[5, 10)	26	14
[2, 5)	63	4

The number of ISI values selected for fuzzing. For a symbolic input in *Category-I*, we iterate over all its values to achieve high path coverage. E.g., in Figure 4, if we do not fuzz all the values of *location.mode* (Line 22), some paths may be unexplored (Line 24).

For symbolic inputs in *Category-II*, we set $n = 10$ and $m = 1$, i.e., one value is selected from each of ten equal parts (Section 6.3). The code-segment summary containing ten value-pairs of TSV and ISIs is combined with symbolic path conditions and solved by the constraint solver. If the constraint solver cannot find satisfying assignments given the current summary, WESTWORLD will fuzz using more values from the domain and update the summary until satisfying assignments are found.

We make the choices for n and m with the following empirical considerations: i) the time t_1 used by fuzzing to obtain the code-segment summary, ii) the time t_2 used by the SMT solver to find satisfying assignments, and iii) the maximum size of messages allowed by the Smartthings Platform. Specifically, if more values are selected for fuzzing, both t_1 and t_2 are increased and the code-segment summary contained in the returned message will be too long and truncated by the SmartThings platform (e.g., in our experiments, around 76.3% messages are truncated if 20 values are fuzzed). On the other hand, if less values are selected for fuzzing, the SMT solver may not find satisfying assignments given the current summary, and a new fuzzing request with newly selected values will be needed, which is expensive due to the communication cost (e.g., in our experiments, around 11.5% code-segments need a second round of fuzzing if only 5 values are selected). In short, the more values are selected, the more comprehensive the code-segment summary is; however, on the other hand, it also introduces longer time for obtaining the summary, larger overload for the constraint solving, and higher risks of truncating messages. Thus, the selection of the parameter values is a trade-off between these factors.

8.4 Completeness and Precision

We use *Dataset-I* and *Dataset-II*. The lines of code (LOC) of the apps are in the range of [43, 2058]. They are divided into several parts based on the number of paths, as shown in Table 1. We compare WESTWORLD to three baselines: a grey-box fuzzer, a concolic executor (without fuzzing assisted), and manual testing.

Comparison with Grey-box Fuzzer. For comparison, we developed a grey-box fuzzer. We apply the coverage-guided input generation technique used in American Fuzzy Lop (AFL) [1], and use

Table 2: Completeness result (%) (full path coverage is attained by WESTWORLD after a minor implementation change).

# of paths in apps	WESTWORLD						Fuzzer						Concolic executor					
	Dataset-I			*Dataset-II*			*Dataset-I*			*Dataset-II*			*Dataset-I*			*Dataset-II*		
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg
≥ 20	100	100	100	100	100	100	69.6	22.4	43.3	46.8	14.3	20.0	72.4	28.3	45.7	44.6	12.8	22.3
[15, 20)	100	100	100	100	100	100	68.4	28.6	46.7	58.3	22.6	42.7	73.5	30.6	50.8	50.2	24.1	38.8
[10, 15)	100	100	100	100	100	100	70.4	27.3	43.3	67.2	36.4	51.0	76.4	32.3	48.3	65.5	33.2	40.3
[5, 10)	100	100	100	100	100	100	82.4	30.5	69.3	78.8	31.5	43.4	100	35.5	64.3	67.2	25.5	38.2
[2, 5)	100	100	100	100	100	100	100	37.2	74.9	86.4	42.4	62.5	100	56.2	80.2	73.2	32.0	56.4

Selenium [24] to mount fuzzing. AFL employs evolutionary algorithms, which uses a feedback loop to assess how good an input is, and retain any input that discovers a new path for generating new inputs. For each environment input, we make sure its value is valid; e.g., humidity is an integer between 0 and 100. The process is terminated when the maximum time (4 hours) is reached.

Table 2 shows the results, including the number of paths, the number of apps, and the maximum, minimum, and average path coverage for WESTWORLD and the fuzzer. The fuzzer cannot reach complete path coverage for most apps. There are two main reasons. (1) Values returned by platform API calls are not under the control of the fuzzer, and thus the branches depending on them cannot be all explored. (2) If a variable influenced by a user input is involved in a condition, often it is difficult for the fuzzer to generate an appropriate value to satisfy the condition. E.g., the platform API `getTwcConditions` returns the *weather* data containing the outside temperature, which is not under the control of the fuzzer, causing a branch depending on the outside temperature to be unexplored.

WESTWORLD achieves complete path coverage for all the official and third-party apps *under testing*. There is a special case, the smart app *DoubleTapModeChange*, which contains a condition invoking `isPhysical` to check whether or not an event is from the physical actuation of an IoT device. As we launch the apps from the IDE simulator, two paths (among 16 paths) are not explored, with a path coverage of 87.5% (= 14/16). After we consider the return value of `isPhysical` as environment data, all paths are explored. For the apps in *Dataset-II* (created by us with more paths and more complex conditional statements), WESTWORLD can successfully explore and analyze all the paths.

Comparison with Concolic Executor. We also developed a concolic executor without fuzzing. The purpose is to demonstrate the benefit of selective code-segment fuzzing in improving path converge. The concolic executor considers user inputs and environment variables as symbolic inputs (the same as WESTWORLD). It executes an app with given inputs, and gathers symbolic path constraints along execution. During path exploration, when a platform API is met, it uses the concrete values to execute the platform API. But it does not consider the return variable of the platform API as a TSV or apply fuzzing to analyze it to improve path coverage. As shown in Table 2, the concolic executor cannot achieve full path coverage for most apps. The reason is that it uses concrete values to simplify symbolic path constraints, which sacrifices completeness (Section 2.2). Taking the app in Figure 4 as an example, the *true* branch of the second *if* statement (Line 24) cannot be explored by the concolic executor. The value of *alreadySet* is determined by the return variable of the platform API `eventsSince`, which is influenced by the symbolic input *timeAgo*. However, as `eventsSince` is closed-source, the concolic executor does not know how to set *timeAgo* such that the value of the return variable of `eventsSince` will make *alreadySet* be *false*. In contrast, WESTWORLD applies selective code-segment fuzzing to handle the particular code segment (containing `eventsSince` and influenced by *timeAgo*) and can find the appropriate value to explore both branches.

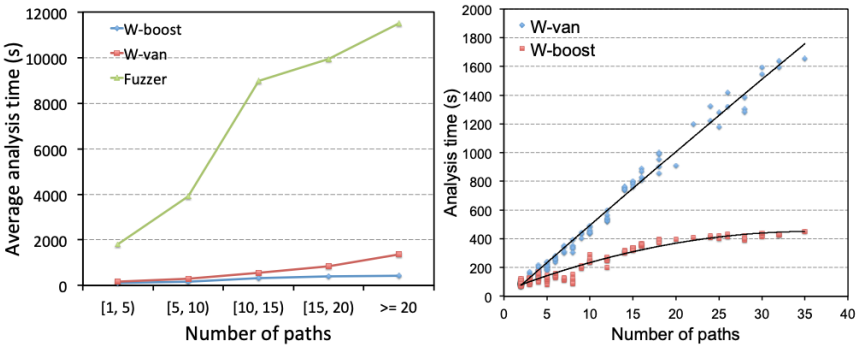
Manual Testing. We randomly selected 10 apps (2 from each category in Table 2) and asked 5 human analysts to test them: one has three years, two have two years, and another two have one year of smart app development experiences. They are skilled in debugging and are familiar with the SmartThings web interface for executing apps. Each analyst tested 2 apps and spent 2 hours on each one. Our studies were approved by IRB at our university.

The results show that 4 apps cannot be fully tested, and 6 apps take more than one hour for testing. We interviewed all the analysts. They mentioned three main problems for testing smart apps. First, it is very troublesome to test smart apps using the SmartThings web interface. When they modified the app configuration to change the values of user inputs (e.g., in Figure 1) or mutated the values of environment variables (e.g., in Figure 2), they needed to wait a long time (more than 30s) until the app is reinstalled on the remote cloud before executing the app. (2) Some environment variables are defined in the source code (e.g., *location.mode* in Figure 4 at Line 22). Through reading the source code, they were not sure which variables are environment variables; more importantly, for some environment variables, they did not know which values can be assigned for testing. E.g., two analysts knew only two values (among three) for the location mode; and one analyst did not know any value for the location mode. This resulted in some paths being unexplored. (3) They could not figure out the appropriate values for user inputs and environment variables in order to trigger unexplored paths, especially when some platform APIs are involved. Thus, a tool that can automatically test smart apps is a critical need.

Precision. To evaluate the precision of our tool, we compare the path condition of the execution path triggered by each test case to the path condition used to generate the corresponding test case; no path divergence occurs. Thus, all generated test cases are precise.

8.5 Efficiency

To evaluate the efficiency, we record the analysis time—the total time used to finish automatic testing of an app—for each app in *Dataset-I*. Figure 12 shows the result. In Figure 12(a), each point represents the average analysis time for the apps in each category.



(a) Our tools *vs.* fuzzer **(b) W-VAN *vs.* W-BOOST**

Figure 12: Efficiency results.

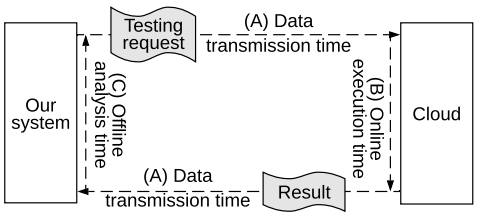


Figure 13: Breakdown of analysis time.

It shows that for the fuzzer, the analysis time grows very quickly when the number of paths increases. W-VAN and W-BOOST are much more efficient than the fuzzer.

In Figure 12(b), the blue diamond represents the analysis time for each app when W-VAN is applied, and the red rectangle shows that when W-BOOST is applied. The two black solid lines are the polynomial trendlines of the analysis time. We can see that the analysis time grows quickly when the number of paths increases for W-VAN, while for W-BOOST, the time grows much slower.

We next seek to understand why W-BOOST performs better than W-VAN. We divide the analysis time into three parts, as shown in Figure 13: (*A*) *data transmission time*, used to send the testing requests to the cloud and get the results back; (*B*) *online execution time*, which includes b_1) logging in the cloud, b_2) locating an app, b_3) updating the app code, b_4) triggering a simulator to install an app, and b_5) running an app on the cloud; and (*C*) *offline analysis time*, used to generate test cases and instrument the app code.

We randomly select 20 apps, run each app ten times, and record the time. (1) The average time for b_1, b_2, b_3 and b_4 is 1.74s, 6.95s, 8.42s, and 34.73s, respectively. The time cost of each is recorded by our *Web Interaction* component. b_1 and b_2 are one-time effort, while b_3 and b_4 are needed for each testing request. (2) The *total* time taken by b_5 and *C* for all testing requests of an app should be approximately the same for W-VAN and W-BOOST. (3) The average time for *A* is around 0.06s, which is very small compared to the cost due to *B*. We can conclude that (i) the time cost due to b_3 and b_4 is large, and (ii) W-BOOST is more efficient than W-VAN as the time taken by b_3, b_4, and *A* is much saved—W-BOOST executes all test cases of one generation through one testing request, while W-VAN executes each test case through one request.

8.6 Bug Finding

Some bugs in IoT apps are hard to find manually; e.g., a division-by-zero bug may be triggered by a particular user input. To demonstrate the effectiveness in bug finding, we apply WESTWORLD to four types of bugs: (1) division by zero, (2) array out of bound, (3) null-pointer dereference, and (4) dead code. The first three types of bugs, once exploited, will make smart apps crash and smart home automation undependable. The last one will increase the app size and analysis cost if static analysis is applied. For a division-by-zero bug, when a division formula (e.g., a/b) is found during path exploration and

```
preferences { input "threshold", "number", required: false }
def contactOpenHandler(evt) {
    def minuteDelay = threshold.toInteger() * 60
    ...
}
```

(a) *GarageDoorOpen-TurnOnLight*.

```
preferences { input "zipCode", "text", required: false }
def presence (evt) {
    def s = getSunriseAndSunset(zipCode)
    def riseTime = s.sunrise
    def setTime = riseTime.time
    ...
}
```

(b) *HoneyImHome* app.

Figure 14: Apps that contain null-pointer dereference bugs.

the denominator is represented as a symbolic expression, the path constraint, $b == 0$, is added to each path condition when it is to be resolved. A division-by-zero bug is found if the path condition is resolvable. This is similar for other bugs with different path constraints added. Due to space limit, we omit it here.

In *Dataset-I*, we found 4 apps with null-pointer dereference bugs, which are caused by the *"enum"* and *"required false"* inputs. If the value for such a variable is not specified, its value will be set to *null* by the platform. When the variable is used to invoke a function, a null pointer exception is triggered.

Figure 14 shows two smart apps containing the bug. (1) In the *GarageDoorOpen-TurnOnLight* app, if a user does not specify a value for the input *threshold*, when the *toInteger* function is called, a null pointer exception is triggered. (2) In the *HoneyImHome* app, the return variable *s* of the platform API `getSunriseAndSunset` is a map. If the value for the input *zipCode* is not specified, the two fields of *s*—*sunrise* and *sunset*—are *null*. As a result, *riseTime* is *null*, and the access to a field of *riseTime* will trigger a null pointer exception.

We further create *Dataset-III* containing eight apps with different bugs inserted, as shown in Table 3: (1) two apps contain division by zero bugs, (2) four are inserted with dead code (due to infeasible paths), (3) one contains an array out of bound bug, and (4) one contains a null-pointer dereference bug. The evaluation shows that WESTWORLD can successfully find all the bugs.

Table 3: Eight inserted bugs.

Bugs	# of apps	Description
Division by zero	1	A user input is converted to another value by a method and then used as a denominator.
	1	A field of the *state* object (storing data of previous executions) is not initialized but used as a denominator.
Dead code	4	Two path conditions are contradicted with each other. No solution can be found by the solver.
Array out of bound	1	The platform API `getChildApps` returns a list of child apps associated with this smart app. An index used to retrieve a child app is larger than the length of the list.
Null-pointer dereference	1	A field of the return variable *weather* of the platform API `getTwcConditions` is accessed. The variable *weather*, which contains the weather data, however, may be null.

9 RELATED WORK

Symbolic Execution. Symbolic execution has been widely applied to test applications like Windows programs [34, 74, 82], Linux programs [4, 10–12, 23, 74], Java programs [5, 8, 51, 59], and firmware [14, 26, 38, 68, 78, 85]. *We propose* remote dynamic symbolic execution *to test smart apps running in a remote execution environment.*

Dynamic Symbolic Execution (DSE). It performs symbolic execution dynamically [11, 12, 23, 34, 60]. Although DSE recovers from imprecision caused by API calls, it sacrifices completeness. To resolve it, we propose *selective code-segment fuzzing* that (1) identifies part of the app code that causes missing execution paths, and (2) fuzzes only this part of code to complement the path coverage.

Fuzzing Combined With Dynamic Symbolic Execution. A set of approaches have been proposed to combine fuzzing and DSE. Most of them are *fuzzing-centric*, in which the path exploration is offloaded to the fuzzer, and DSE is selectively used to assist fuzzing [7, 19, 45, 53, 57, 58, 67, 75, 77, 81, 83, 84]. E.g., Driller [67] uses DSE to make the fuzzer "revive". It aims to find bugs hidden deep, but not complete path exploration. DeepFuzz [7] uses a similar idea. DigFuzz [84] and MDPC [75] design a path prioritization model to quantify each path's difficulty and prioritize them for DSE. QSYM [77] loosens precision of DSE for better performance.

Compared to fuzzing-centric approaches such as Driller, our method is *symbolic execution-centric*. We made this SE-centric choice due to the unique challenge: the communication cost between the remote cloud and local analyzer, and request handling time cannot be omitted. Thus, each testing request is expensive. Given a path like (*temp<75 && temp>68*), Driller cannot avoid generating a lot of testing requests that repetitively take the same path, while symbolic execution is good at this. Plus, our boosted generational search further enhances the vanilla generational search by testing a whole generation of inputs in one request to improve the efficiency.

Analysis of IoT Applications. A lot of researches have been made on analyzing IoT or mobile apps [2, 15–17, 20, 21, 29, 30, 39, 43, 44, 46–48, 61, 73, 76, 79, 80]. FlowFence enforces sensitive data flow control via opacified computation [29]. HAWatcher [30] extracts semantics from IoT apps for anomaly detection. Centaur [51] also needs to handle the difficulty due to the decoupled concrete execution and symbolic execution; specifically, it migrates the heap from an Android system to the symbolic executor. Unlike prior work, WESTWORLD is the first system that enables DSE of IoT apps.

10 DISCUSSION AND LIMITATIONS

To demonstrate bug finding capability, WESTWORLD is applied to four types of bugs that result in crashes. Besides these, it can also be applied to some sophisticated vulnerabilities, e.g., cross-app interference (CAI) bugs [21, 22]. A set of work uses model checking [17, 18, 56], or combines static analysis and NLP techniques [28, 72], to detect CAI bugs. HomeGuard is the first work that leverages classic symbolic execution and SMT solving for finding CAI bugs [22]. To handle platform APIs, it uses manual function modeling and thus causes imprecision. WESTWORLD is the first DSE system for analyzing IoT apps and attains precise analysis.

The proposed remote DSE is enabled by multiple ideas, such as leveraging logging and messaging to collect path conditions, converting environment data to symbolic inputs, boosted generational search, and selective code-segment fuzzing. Extending remote DSE to analyze other types of code running on remote proprietary platforms is an interesting research direction.

Limitations. WESTWORLD uses a constraint solver to generate test cases, which can scale to complex constraints [6] but also has limitations [25, 41]. Our evaluation shows WESTWORLD can achieve completeness for *the apps under testing*, mainly because existing IoT apps usually have a small number of paths. We do not claim WESTWORLD guarantees completeness in general.

Selective code-segment fuzzing works well for analyzing smart apps, as symbolic variables (such as temperature, home mode, and switch state) usually do not have many discrete values. For general programs which often have unbounded possible concrete values for symbolic variables, however, the method does not work.

11 CONCLUSION

We have presented the first system that enables dynamic symbolic execution (DSE) of smart apps. As most IoT platforms are cloud-based proprietary execution environment, various challenges arise. Exploiting the uniqueness of environment inputs, selective code-segment fuzzing was proposed to assist DSE. Boosted generational search was designed to accelerate the analysis. We implemented WESTWORLD, which performs fuzzing-assisted DSE-centric analysis of smart apps. The evaluation shows that WESTWORLD is effective and efficient in path exploration and bug finding.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Dr. Sébastien Bardin, and the anonymous reviewers for their constructive suggestions and comments. This work was supported in part by the US National Science Foundation (NSF) under grants CNS-1815144, CNS-1850278, CNS-1953073, CNS-1856380, CNS-2016415 and CNS-2107093. This work was partially supported by an ASPIRE grant from the Office of the Vice President for Research at the University of South Carolina.

REFERENCES

[1] AFL. 2020. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[2] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. Sok: Security evaluation of home-based iot deployments. In *IEEE S&P*. 208–226.

[3] Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2019. Constraint Programming for Dynamic Symbolic Execution of JavaScript. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 1–19.

[4] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic exploit generation. (2011).

[5] Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, and Gabor Karsai. 2019. Dynamic symbolic execution for the analysis of web server applications in Java. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2178–2185.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[7] Konstantin Böttinger and Claudia Eckert. 2016. Deepfuzz: Triggering vulnerabilities deeply hidden in binaries. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 25–34.

[8] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2016. JBSE: a symbolic executor for Java programs with complex heap inputs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 1018–1022.

[9] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. 2007. *BitScope: Automatically dissecting malicious binaries*. Technical Report. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon ….

[10] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *USENIX Security*.

[11] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[12] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: automatically generating inputs of death. *TISSEC* (2008).

[13] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[14] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference*. 746–759.

[15] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. 2018. Sensitive information tracking in commodity IoT. In *USENIX Security*.

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

[17] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated iot safety and security analysis. In *USENIX Security'18)*. 147–158.

[18] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT.. In *NDSS*.

[19] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.

[20] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. 2021. PFirewall: Semantics-Aware Customizable Data Flow Control for Home Automation Systems. In *NDSS*.

[21] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. *arXiv preprint arXiv:1808.02125* (2018).

[22] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2020. 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, 411–423.

[23] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a Platform for In-Vivo Multi-Path Analysis of Software Systems. In *ASPLOS*.

[24] Code Review Guidelines and Best Practices. [n. d.]. https://www.seleniumhq.org.

[25] J Conway. 1972. Unpredictable iterations. (1972).

[26] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution.. In *USENIX Security Symposium*. 463–478.

[27] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[28] Wenbo Ding and Hongxin Hu. 2018. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 832–846.

[29] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. Flowfence: Practical data protection for emerging iot application frameworks. In *USENIX Security 16*.

[30] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. 2021. Hawatcher: Semantics-aware anomaly detection for appified smart homes. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[31] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 419–429.

[32] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 47–54.

[33] Patrice Godefroid. 2011. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 258–269.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*. ACM.

[35] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. *Queue* 10, 1 (2012), 20–27.

[36] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.

[37] Rajiv Gupta. 1992. Generalized dominators and post-dominators. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 246–257.

[38] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. 2017. Firmusb: Vetting USB device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2245–2262.

[39] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms.. In *NDSS*.

[40] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[41] Jeffrey C Lagarias. 2010. *The ultimate challenge: The 3x+ 1 problem*. American Mathematical Soc.

[42] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1 (1979), 121–141.

[43] Xiaopeng Li, Fengyao Yan, Fei Zuo, Qiang Zeng, and Lannan Luo. 2019. Touch well before use: Intuitive and secure authentication for iot devices. In *The 25th annual international conference on mobile computing and networking*. 1–17.

[44] Xiaopeng Li, Qiang Zeng, Lannan Luo, and Tongbo Luo. 2020. T2pair: Secure and usable pairing for heterogeneous iot devices. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 309–323.

[45] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin IP Rubinstein. 2020. Legion: Best-First Concolic Testing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 54–65.

[46] Xuanyu Liu, Qiang Zeng, Xiaojiang Du, Siva Likitha Valluru, Chenglong Fu, Xiao Fu, and Bin Luo. 2021. SniffMislead: Non-Intrusive Privacy Protection against Wireless Packet Sniffers in Smart Homes. In *RAID*.

[47] Lannan Luo. 2020. Heap memory snapshot assisted program analysis for android permission specification. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 435–446.

[48] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. 2016. Repackage-proofing android apps. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 550–561.

[49] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 389–400.

[50] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177.

[51] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 225–238.

[52] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2019. Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation. *IEEE Transactions on Mobile Computing* 19, 12 (2019), 2946–2964.

[53] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 416–426.

[54] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[55] Malte Mues, Till Schallau, and Falk Howar. 2020. Jaint: A Framework for User-Defined Dynamic Taint-Analyses Based on Dynamic Symbolic Execution of Java Programs. In *International Conference on Integrated Formal Methods*. Springer, 123–140.

[56] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the safety of IoT systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 191–203.

[57] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, 1475–1482.

[58] Brian S Pak. 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University* (2012).

[59] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. [n. d.]. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. In *ASE'13*.

[60] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 263–272.

[61] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. 2015. Network-level security and privacy control for smart-home IoT devices. In *WiMob*. IEEE.

[62] SmartThings. [n. d.]. An Overview of Using Custom Code in SmartThings. https://community.smartthings.com/t/faq-an-overview-of-using-custom-code-in-smartthings-smartthings-classic/16772.

[63] SmartThings. [n. d.]. SmartThings Community. https://community.smartthings.com.

[64] SmartThings. 2018. Code Review Guidelines and Best Practices. https://docs.smartthings.com/en/latest/code-review-guidelines.html.

[65] SmartThings. 2020. SmartThings Community Forum For Third-party Apps. https://community.smartthings.com/.

[66] SmartThings. 2020. SmartThings Official App Repository. https://github.com/SmartThingsCommunity.

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[68] Pramod Subramanyan, Sharad Malik, Hareesh Khattri, Abhranil Maiti, and Jason Fung. 2016. Verifying information flow properties of firmware using symbolic execution. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 337–342.

[69] The Apache Groovy Programming Language. 2019. Closures. https://groovy-lang.org/closures.html.

[70] Things That Are Smart. [n. d.]. Using Custom Code. https://thingsthataresmart.wiki/index.php?title=Using_Custom_Code.

[71] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, Guoqing Harry Xu, and Shan Lu. 2020. Understanding and automatically detecting conflicting interactions between smart home IoT applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1215–1227.

[72] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the attack surface of trigger-action IoT platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1439–1453.

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

[74] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution.. In *NDSS*. Citeseer.

[75] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*. 291–302.

[76] Rixin Xu, Qiang Zeng, Liehuang Zhu, Haotian Chi, Xiaojiang Du, and Mohsen Guizani. 2019. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access* 7 (2019), 63457–63471.

[77] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 745–761.

[78] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.. In *NDSS*.

[79] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 50–61.

[80] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, Zhoujun Li, Chin-Tser Huang, and Csilla Farkas. 2019. Resilient user-side Android application repackaging and tampering detection using cryptographically obfuscated logic bombs. *IEEE Transactions on Dependable and Secure Computing* (2019).

[81] Bin Zhang, Chao Feng, Adrian Herrera, Vitaly Chipounov, George Candea, and Chaojing Tang. 2018. Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing. *Iet Software* 12, 6 (2018), 507–519.

[82] Bin Zhang, Chao Feng, Bo Wu, and Chaojing Tang. 2016. Detecting integer overflow in Windows binary executables based on symbolic execution. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 385–390.

[83] Li Zhang and Vrizlynn LL Thing. 2017. A hybrid symbolic execution assisted fuzzing method. In *TENCON 2017-2017 IEEE Region 10 Conference*. IEEE, 822–825.

[84] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.. In *NDSS*.

[85] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[86] Chaoshun Zuo and Zhiqiang Lin. 2017. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *Proceedings of the 26th International Conference on World Wide Web*. 867–876.