Android Testing via Synthetic Symbolic Execution

Xiang Gao*
National University of Singapore, Singapore
gaoxiang@comp.nus.edu.sg

Shin Hwei Tan†
Southern University of Science and Technology, China
tansh3@sustc.edu.cn

Zhen Dong
National University of Singapore, Singapore
zhen.dong@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Symbolic execution of Android applications is challenging as it involves either building a customized VM for Android or modeling the Android libraries. Since the Android Runtime evolves from one version to another, building a high-fidelity symbolic execution engine involves modeling the effect of the libraries and their evolved versions. Without simulating the behavior of Android libraries, path divergence may occur due to constraint loss when the symbolic values flow into Android framework and these values later affect the subsequent path taken. Previous works such as JPF-Android have relied on the modeling of execution environment such as libraries. In this work, we build a dynamic symbolic execution engine for Android apps, *without* any manual modeling of execution environment. Environment (or library) dependent control flow decisions in the application will trigger an on-demand program synthesis step to *automatically* deduce a representation of the library. This representation is refined on-the-fly by running the corresponding library multiple times. The overarching goal of the refinement is to enhance behavioral coverage and to alleviate the path divergence problem during symbolic execution. Moreover, our library synthesis can be made context-specific. Compared to traditional synthesis approaches which aim to synthesize the complete library code, our context-specific synthesis engine can generate more precise expressions for a given context. The evaluation of our dynamic symbolic execution engine, built on top of JDART, shows that the library models obtained from program synthesis are often more accurate than the semi-manual models in JPF-Android. Furthermore, our symbolic execution engine could reach more branch targets, as compared to using the JPF-Android models.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

*Corresponding Author
†This work was done by the author at National University of Singapore.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238225

KEYWORDS

Android testing, Symbolic execution, Program synthesis

ACM Reference Format:
Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android Testing via Synthetic Symbolic Execution. In *Proceedings of Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238225

1 INTRODUCTION

Symbolic execution is a powerful program analysis technique which can simultaneously explore multiple program paths that a program could take under different inputs. However, it is difficult to apply it to framework-oriented Android apps because Android framework is very complex and it is built with multiple languages. Moreover, Android framework cannot be executed outside Android devices/emulators.

One common solution to this problem is to manually generate a framework *model* that simulates the framework behavior but can be symbolically executed. For example, JPF-Android [4] relies on a semi-manual created model of Android framework that can be symbolically executed on Java virtual machine. However, significant effort needs to be spent on writing models of Android libraries. According to a study on the Android ecosystem, Android is evolving rapidly at an average rate of 115 API updates per month [18]. The rapid evolution of Android API poses additional challenges for applying these approaches in practice. To relieve the burden of writing framework models, PASKET [13] explores the possibility of synthesizing framework models for symbolic execution. It leverages several design patterns (e.g, the Observer patterns) to synthesize models for several classes in Android frameworks and these models are then passed to off-the-shelf symbolic execution engine where the symbolic execution engine simply serves as a separate validation step for the correctness of the synthesized models. Instead of incorporating the constraints gathered during symbolic execution, PASKET requires tutorial programs to exercise the target models.

Another standard solution is to perform concrete execution of the framework code to provide environment models for symbolic execution of a particular code of interest such as app code. For example, S^2E [6] performs symbolic execution on specified components and concretely executes other components. Meanwhile, existing concolic execution techniques on Android apps such as ACTEve [1] and Collider [11] uses instrumentation for constraint tracking. One of the major challenges in such approaches is the

problem of path divergence where missed instrumentations could lead to the divergence between the concrete and symbolic execution paths [1, 10]. Moreover, the path divergence problem may impair the soundness and completeness of the concolic execution. Without symbolic execution of framework code, certain path constraints can be missed when symbolic values flows into the framework.

In this paper, we present *synthetic symbolic execution*, as embodied by our tool SynthesiSE. It is a concolic execution approach for Android apps that aims to alleviate all of the above mentioned problems. Instead of relying on manually written models for Android framework, our approach automatically deduces expression representing Android models dynamically during the execution. When dependent control flow decisions are encountered in subsequent execution, the deduced expression will be refined with the ultimate goal of enhancing branch coverage and to alleviate the path divergence problem. Moreover, instead of generating the entire Android model classes in a single step, our library synthesis is context-specific, which allows it to generate more precise expression for a given context.

Symbolic execution of applications always requires capture of the environment, which includes libraries. The two extreme approaches are the *in-vitro* approach where the effect of the libraries are modeled, and the *in-vivo* approach where the effect of libraries/environment are captured via whole system execution. For C programs, the KLEE tool [5] embodies the in-vitro approach and the S2E tool [6] embodies the in-vivo approach. At a high level, our approach lies in between the in-vivo and in-vitro approaches. We do not model the libraries, and yet we go beyond concrete execution of libraries. Instead, we synthesize expressions to capture the effect of libraries with the goal of enhancing branch coverage in testing. We have implemented a dynamic symbolic execution platform for Android apps. In summary, we make the following contributions.

- **On demand framework synthesis.** We present an approach which synthesizes the relationship between symbolic inputs and outputs for a framework library during concrete execution. Using the deduced relationship, we collect more complete path constraints from the programs to explore paths which are missed by symbolic execution due to concrete execution of framework code. We implement a dynamic symbolic execution platform for Android apps which could be used for enhancing testing of Android apps. As our approach does not require manual modeling, our platform is agnostic to different Android SDK versions. Moreover, this approach could be used for the symbolic execution of any code that uses libraries.
- **Incorporating GUI constraints.** Android apps rely on Graphical User Interface (GUI) to interact with users. GUI constraint imposes restrictions to the layout and application resources defined in an Android app. Our platform automatically extracts these constraints and incorporates them into symbolic execution for exploration of Android apps.
- **Importance of modeling Android libraries** We perform a study of 68 Android apps in the Androtest benchmark [7] to investigate how often the results of invoking Android libraries affect the dependent branch decision in Android apps. Our study shows that 37.1% of branches in Android apps are affected by the results of executing Android libraries.
- **Evaluation.** We evaluate our approach on 14 Android apps by comparing our synthesized models against real implementation and semi-manually created models in JPF-Android. Our evaluation shows that if we treat all the branches affected by an Android library invocation as targets, our synthesized models are able to reach more targets than the models used in JPF-Android.

2 BACKGROUND

Concolic execution. Concolic execution [9] is a program analysis technique combining concrete execution and symbolic execution. It uses a concrete value c to generate a path π_c, and uses symbolic execution along the path to compute a *path condition* pc. Systematic negation of branch conditions in the path condition pc then leads to modified constraints pc', which is solved to generate inputs which trace different paths. The process is repeated to obtain a test-suite with high path coverage. Concolic execution has been used to analyze Android apps for event generation [1], and fault localization [2]. Typically it is performed either on a Java symbolic engine with Android library models, or on Android Runtime using instrumentation techniques to trace the app execution for constraints collection and explore different paths with generated inputs.

Program synthesis. Program synthesis has been formalized to be a second-order constraint solving problem through propositional synthesis encoding, recently in [19], and we use this work in our testing method. Given a set of components, it will construct the set of terms and represent them via a tree. Boolean variables s_i, called selector variables, are assigned to choose a particular term from among a set of terms. Specifically, each leaf of the tree corresponds to components without input and intermediate node has as many subnodes as the maximal number of inputs of a component. For each node i with sub-node $\{i_1,i_2,...,i_k\}$, the output and inputs are represented by out_i and $\{out_{i_1},out_{i_2},...,out_{i_k}\}$, respectively. In addition, s_i^j is the j-th selector of node i, which means j-th component is used in this node, C is the number of components, F_j is the semantics of j-th component. For node i, a set of terms is encoded as $\varphi_i := \varphi_{node} \wedge \varphi_{choice}$, such that

$$\varphi_{node} := \bigwedge_{j=1}^{|C|} s_i^j \Rightarrow out_i = F_j(out_{i_1}, out_{i_2}, ..., out_{i_k}) \quad (1)$$

$$\varphi_{choice} := exactlyOne(s_i^1, s_i^2, ..., s_i^C) \quad (2)$$

φ_{node} describes the semantic relations between the output value of node i and the output value of its subnode, while φ_{choice} restricts that only one component is selected inside each node. Given a set of input-output pairs, this algorithm should return a set of nodes satisfying the input-output restriction. Using the above encoding, the second-order constraint solver can be implemented on top of the first-order solver.

3 THE IMPORTANCE OF MODELING ANDROID LIBRARIES

To study the importance of encapsulating the constraints given by Android libraries, we perform an initial investigation of 68 Android apps in the Androtest benchmark [7], a commonly used benchmark in prior evaluations of Android testing techniques [17, 29]. Our goal

Figure 1: The number of branches affected by Android library calls and the corresponding percentage

is to investigate how often the results of invoking Android libraries affect the dependent branch decision in Android apps. To achieve this goal, we perform taint analysis to determine whether there exists an information flow from Android library calls to subsequent branches in the evaluated Android apps. Specifically, we modify FlowDroid [3], a taint analysis tool, by specifying all Android libraries as "source" and branches as "sink" to count the number of paths from "source" to "sink". We also instrument apps using Soot [34] to compute the total number of branches.

Our investigation aims to answer two questions:

Q1: What is the general structure of Android apps in terms of control flow? What is the number of branches in apps?

Q2: How would the results of invoking Android libraries influence the branch decisions in Android apps?

We study $Q1$ to obtain an overview of the control structure in Android apps. Our main goal in $Q1$ and $Q2$ is to compute the percentage of branch decisions affected by the results of invoking Android libraries.

Figure 1 shows the results of our study. The x-axis in Figure 1 denotes the names of the evaluated apps, whereas the y-axis on the left and the bar represent the number of branches in each app that are affected by the results of invoking Android libraries. Meanwhile, the dashed line and the y-axis on the right of Figure 1 denote the percentage of branches affected by Android library calls, computed using the equation in the following.

$$\%branches\ affected = \frac{\#\ of\ branches\ affected}{total\ \#of\ \ branches}$$

According to Figure 1, the branch decisions of most apps are heavily influenced by Android libraries. On average, 37.1% of branches in the evaluated Android apps are affected by the results of invoking Android libraries. Figure 1 also shows that in one extreme case, although the "gestures" app only contains six branches in total, all of its branches are affected by Android library calls. Overall, our study demonstrates a high percentage of branches in Android apps are affected by Android library calls. Hence, we derive that if Android libraries are not properly modeled by a symbolic execution engine, the constraints imposed by Android libraries in Android apps may be lost. Subsequently, the inappropriate modeling of Android libraries may also hinder the ability of a symbolic execution engine to explore more program branches.

```
1  public boolean onTouchEvent(MotionEvent event) {
2    float x = event.getX();
3    float y = event.getY();
4    int offset = getOffsetForPosition(x, y);
5    putOffsetInRange(offset);
6  ...}
7  private int putOffsetInRange(int o) {
8    int offset = o;
9    Editable text = getText();
10   int realLength = text.length();
11   if (offset >= realLength)
12     return offset;
13   Editable editable = getText();
14   while (offset >= 0 && findText(editable, offset)
15            == -1 && findChip(offset) == null) {
16     offset--;
17   }
18   return offset;
19 }
```

Listing 1: The onTouchEvent event handler of TagActivity

4 OVERVIEW

We give a high-level overview of synthetic symbolic execution and SynthesiSE by presenting an example Android application, *TagActivity*. *TagActivity* is an Android open-source text tag editor that generates tags using user text input. Listing 1 shows the simplified source codes of one event handler, *onTouchEvent*, which will be invoked when user touches the screen. The execution of this event handler proceeds as follows: (1) extracting the touch coordinates (x, y) (2) invoking Android library method *getOffsetForPosition* with touch coordinates as parameters (3) triggering a local method *putOffsetInRange* which contains at least four path conditions (including path conditions inside *findText* and *findChip*) that are related to the return variable of *getOffsetForPosition*.

Existing symbolic execution approaches may face several challenges when analyzing this simple Android app. First, path explosion may occur when these techniques symbolically execute the whole Android app and framework. Second, some Android libraries, such as *getX*, cannot be symbolically executed, since it will invoke a native call which is implemented in the C language. Although existing symbolic execution techniques could leverage pre-built models for Android libraries, building models is a time-consuming task and the manually/automatically designed model may be imprecise. For example, in the model of JPF-Android, the method *getOffsetForPosition* in Listing 1 simply returns a constant global variable TOP_INT. Another option is to concretely execute Android libraries while performing symbolic execution of app code. However, the symbolic relationship between the input *x*,*y*, and the output *offset* will be lost if the library code is concretely executed. Therefore, existing

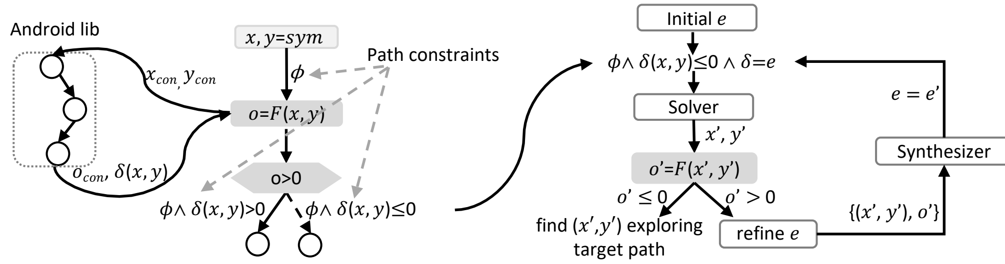

Figure 2: Workflow of on-demand synthesis for symbolic execution

symbolic execution engines cannot generate the correct inputs (x,y) to explore program branches that are affected by the output (*offset*).

To solve the above mentioned problems, SynthesiSE is designed to efficiently explore behaviors that are constrained by the return values of concretely executed library. For example, if we concretely execute the method *getOffsetForPosition*, the execution of line 16 (Listing 1) is constrained by the concrete execution result. To continue the symbolic exploration after concrete execution, symbolic relation between its input and output should be recovered.

The key idea of SynthesiSE lies on the Symbolic-Concrete transition across the symbolic/concrete execution boundary. The interleaving between symbolic value and concrete value must be handled carefully, to preserve the consistency and completeness. We now describe how we handle the transition.

Symbolic → Concrete transition When a library that should be concretely executed is invoked and its parameters are marked symbolic, the concrete values for the arguments should be generated. The concretization of symbolic variables can affect subsequent path exploration by (incorrectly) ruling out certain paths. S^2E uses a back-and-forth mechanism for switching between symbolic and concrete execution to ensure the execution consistency [6]. Different from S^2E, SynthesiSE does not suffer from the path missing problem due to concretization, as it performs concolic execution from the beginning of the execution. SynthesiSE retains concrete value as well as symbolic value for each variable at each program point. Therefore, it can directly invoke a concrete execution with the concrete values of its parameters, without affecting the subsequent symbolic exploration.

Concrete → Symbolic transition We consider two situations where concrete value is transferred to symbolic:

- the return value of library methods that obtain user inputs (like *getX()* in Listing 1)
- the return value of library method, if its return value is dependent on its symbolic arguments.

For the first scenario, a new symbolic variable will be created. For the second scenario, we will illustrate our workflow in Figure 2.

Assume that the library code *getOffsetForPosition* (represented by F in Figure 2) should be concretely executed and x, y are symbolic variables. This method is invoked with the concrete value of its parameters. Due to the concrete execution, the symbolic information of parameters cannot flow into the output (*offset*). Therefore, a function $\delta(x, y)$ is introduced to represent the Android library call (F). Meanwhile, the symbolic value of method output (*offset*) will be regarded as $\delta(x, y)$. In the subsequent execution, there is path condition $\delta(x, y) > 0$ related to variable o. Assume that in the first iteration of the execution, we explore the left path, then we can generate a path constraint ($\phi \wedge \delta(x, y) > 0$) along with the concrete execution. To explore a different path, we negate the corresponding path condition and provide the generated path constraints $\phi \wedge \delta(x, y) \leq 0$ to the solver. As function $\delta(x, y)$ is unknown, a synthesis process will be triggered to deduce the relation between inputs (x, y) and output (*offset*). With the initialized expression e, a new value (x', y') will be generated by solving $\phi \wedge \delta(x, y) \leq 0 \wedge \delta = e$. Ideally, the solution (x', y') should allow us to explore the target path. However, since we cannot ensure that the synthesized expression $\delta(x, y)$ represents F correctly, this solution may fail to follow the negated path. If the input (x', y') could trigger the execution of the target path, then we derive a new input. Otherwise, the expression e will be refined based on the newly generated input (x', y') and its corresponding output o'.

Furthermore, application configurations or UI hierarchy may impose additional constraints on the variables. In this example, touch coordinates (x, y) must be within the scope of the screen. We define such constraints as *GUI constraint*, which can be missed by concretely executing library code. Therefore, we introduce a strategy to collect *GUI constraint* by parsing app configuration (XML) and monitoring UI hierarchy.

5 APPROACH

In this section, we first introduce the notations that we use, and then we present the on-demand program synthesis and the extension of traditional concolic execution technique. Throughout this section, P represents the program under test, X to denote the set of symbolic variables, and δ to denote the library function to be synthesized.

5.1 On-demand analysis

Algorithm 1 shows the key steps in our on-demand analysis built on top of existing concolic execution engine. The algorithm operates on an Android app that consists of Java source code with function calls to Android libraries. The *pathExploration* procedure is similar to traditional concolic execution approaches [9] where symbolic execution is run simultaneously with concrete execution and path constraints *pc* are collected alongside with the concrete execution. The *pathExploration* procedure proceeds by invoking the *executeConcolic* procedure that iterates through each program statement. For each program statement *stmt*, *executeConcolic* procedure distinguishes between two kinds of statements: (1) statements that involve invocations of Android libraries and (2) other statements that can be treated as regular Java statements. The algorithm will process the second kind of statements by invoking

Algorithm 1: Synthetic Symbolic Execution; the main procedure is `pathExploration`

```
1  Procedure executeConcolic(program P, symbolic X)
2      symbolic memory S=[x ↦ expression | x ∈ X];
3      concrete memory C=[x ↦ value | x ∈ X];
4      path condition pc := ∅;
5      library functions Δ := [];
6      stmt := stmt_0;// Initial program statement
7      while stmt ∉ Exits do
8          switch stmt do
9              case y := libraryInvocation(v_0, v_1,...v_n) do
10                 c := executeConcrete (stmt);
11                 C := C[y ↦ c];
12                 D := {v|v ∈{v_0, v_1,...v_n}∧S[v] ≠ null};
13                 if !isEmpty(D) then
14                     s ← δ({S[d] | d ∈ D});
15                     S := S[y ↦ s];
16                     Δ := Δ ∪ δ;
17                 end
18             otherwise do
19                 executeJavaConcolic (stmt, pc);
20             end
21         end
22         stmt := P.getNextStmt ();
23     end
24     return ⟨pc, Δ⟩ ;
25
26 Procedure pathExploration(program P)
27     X := init ();
28     while X ≠ null do
29         ⟨pc, Δ⟩ := executeConcolic (P, X);
           // select one condition to negate
30         pc_i := select(pc);
31         pc′ = ¬pc_i ∧ (pc \ pc_i);
32         Θ := {δ|δ ∈ Δ ∧ contains(pc′, δ)};
33         if !isEmpty(Θ) then
34             X := synthesize(pc′, Θ);
35         else
36             X = solve (pc′);
37         end
38     end
```

a concolic execution engine for Java programs (lines 18-20). For the statements that contain invocations of Android libraries, we assume that they are of the form y:= $libraryInvocation(v_0, v_1,...v_n)$ where the effect of calling an Android library function will be captured through its return values[1]. We execute the statements that involve Android library function calls and analyze the statement to check if it satisfies two criteria (1) at least one of the function arguments is symbolic variable (line 12-13), and (2) its return value is accessed in subsequent branch conditions in the program (line 32). As our synthesis engine is triggered when an Android library invocation satisfies these two criteria, our synthesis engine is invoked *on-demand*. In line 14, the to-be-synthesized function δ collects all arguments of the function $libraryInvocation$ that contain symbolic values. After that, our algorithm maps the return variable y to the to-be-synthesized function in our symbolic memory S (line 15).

[1] A function where its return value is not stored is rewritten with new temporary variable that store the return value.

Algorithm 2: Iterative Synthesis

```
1  Procedure synthesize(program P, pathCond pc, func δ)
       // < in_0, out_0 > from initial execution
2      e = out_0;
3      R = {⟨in_0, out_0⟩};
4      synthesisIteration = 0;
5      while synthesisIteration < L do
6          x′ := solve(pc ∧ δ = e);
7          if UNSAT or UNKNOWN then
8              x′ := solve(pc ∧ δ ≠ e);
9              if UNSAT or UNKNOWN then
10                 return null;// unsatisfiable pc
11             end
12         end
13         ⟨π, r_io⟩ := concreteExecution(P, x′);
14         R = R ∪ {r_io};
15         if π == target then
16             return x′;
17         else
18             e := componentBasedSynthesis(R);
19         end
20         synthesisIteration + +;
21     end
```

After processing the program statements on line 29, the concolic execution engine proceeds by picking a condition to negate in order to visit a new execution path (line 30-31). In lines 32-34, if the modified path constraint pc' contains the to-be-synthesized function, our algorithm will trigger the *synthesis* procedure to generate new input to explore a new execution path. Otherwise, our engine will solve the modified path condition like regular concolic execution.

5.2 Synthesis

One of the main challenges in concolic execution is the problem where concolic testing may get stuck in exploring a huge number of program paths before reaching the target state[16]. Our synthesis engine aims to solve this problem by synthesizing a representation for the Android library invocation. Specifically, we consider all the subsequent branch decisions that are dependent on the results of the Android library invocation as the target states $target$. During the path exploration, a generated input may not be able to reach the target states, or no input satisfies the condition of the target states due to the inaccurate library synthesis. For such cases, we concretely execute the program and iteratively refine the synthesized function until generated inputs reach the target states.

5.2.1 Iterative refinement. Algorithm 2 presents our iterative refinement steps. In practice, path conditions collected by concolic engine might be computed with symbolic values from multiple to-be-synthesized functions. Our algorithm refines each of the involved functions independently using the same process. For the sake of simplicity, we demonstrate iterative refinement with a single synthesized function in Algorithm 2. Given the input-output

Table 1: The categorization of synthesis components

Level	Type	Elements
1	Constant	constants
2	BitWise	<<, >>, &, |
3	Arithmetic	+, -, *, /
4	Flow-control	Ite (If-then-else)
5	Array-access	array

pair $\langle in_0, out_0\rangle$ from the initial concrete execution of an Android library function, the *synthesize* procedure starts by initializing the synthesized expression e to the output out_0 (constant expression) of the Android library function (line 2). Then, our synthesis engine conjoins the constraint $\delta = e$ represented by the synthesized expression with the path constraint pc and pass this new constraint to the SMT solver (line 6). If the solver returns UNSAT or UNKNOWN, this unsatisfiability may be caused by pc or the introduced constraint $\delta = e$. Therefore, we will try to solve $pc \wedge \delta \neq e$. If the solver still returns UNSAT or UNKNOWN, that means pc is unsatisfiable and we directly return null. Otherwise, we generate a new input x', and perform concrete execution of program P using the updated input to obtain additional input-output pair r_{io} (line 13). When our algorithm has successfully synthesized a library representation to reach the target state, we return the new generated input x' to the concolic execution engine for its path exploration (Section 5.1). Otherwise, the synthesis engine adds a new input-output pair r_{io} to the set R and continue generating new expressions until we reach the target state or the synthesis iteration exceeds the threshold L.

Though this algorithm requires iterative refinement, the program will not be concolically executed multiple times. To gather more input-output pairs, multiple concrete executions are needed, while the symbolic execution is performed once. Therefore, compared with traditional concolic execution, this algorithm will not induce too much overhead.

5.2.2 Core program synthesis. Different from traditional synthesis approaches which aim to simulate behaviors of a library, our synthesis engine is designed to synthesize an expression that can guide path exploration of symbolic execution. We use the recent work of [19] to generate expressions by incrementally feeding input-output pairs.

As mentioned in Algorithm 2, we provide our synthesizer with one input-output pair initially, then iteratively feed more pairs only if the synthesized expression does not help in generating new inputs to explore the target state. When the generated input x' by solving $(pc \wedge (\delta = e))$ fails to make real execution follow the target path, the synthesized expression e must be incorrect. In this situation, refinement process will be triggered to refine expression e. This strategy does not need pre-generated input-output pairs, instead, it will generate new pairs according to the feedback of concrete execution. Therefore, this strategy needs less input-output pairs to synthesize an expression to cover more program behaviors.

Moreover, we provide different components to the synthesizer based on an incremental strategy. To reduce the complexity of synthesis, we first categorize common components according to their complexity [24]. Table 1 shows the categorized components. The first level is "Constant", which means that our synthesizer will first use a constant value to represent the library. If this level fails to generate expressions that satisfies the input-output relations, then it will try higher levels with more input-output pairs. This process will terminate when a certain iteration limit is reached or when it generates new inputs that explore the target path (algorithm will reach the iteration limit if the target path is infeasible).

5.2.3 Context-specific. Our synthesis process is context-specific, which means that we will treat the same method invoked in different program points as different. Synthesizing a library method under a certain context will simplify the synthesis process, because the library method may only show part of its behaviors under this context (e.g. some execution paths are constrained by global states). Under a certain context, more precise models can be synthesized. Context-specific synthesis is used because the goal of our synthesis is not simulating complete library behaviors, instead, synthesizing a model to help symbolic execution reach more branches.

We consider the *context* of an Android app to be the global states of the app. In the refinement process, we have to make sure the function to-be-synthesized is invoked under the same context, so that the input-output pairs are generated based on same context. We will illustrate this process using the example in Figure 2. Consider a library call *F(x,y)* invoked with x and y as argument. Let x_1 and x_2 be two concrete values of x (same operation for variable y). In Figure 2, ϕ is the path constraint before the invocation of *F(x,y)*, and x_1 and x_2 is obtained by solving $\phi \wedge \delta(x, y) \leq 0 \wedge \delta = e$. Since x_1 and x_2 satisfy the path constraint ϕ, the program will reach the method invocation F following the same path when it is given either x_1 or x_2 as input. Hence either the library invocation is reached with the same global state, or the difference in global states at the time of the library invocation *F(x)* is accounted for by the difference in the value of x. Though our synthesized expressions do not model the effect of global states, they adequately capture this situation by synthesizing expressions on the argument x.

5.3 GUI constraints

For Android apps, developers often specify certain constraints on inputs by defining constraints for UI elements. These constraints are validated by Android libraries at runtime. For instance, in app *TagActivity*, the touch coordinates (*x*,*y*) are extracted at runtime. Based on our experimental devices, the bound of screen is [0,0][1440,2879]. This means that x is limited to (0, 1440) while y is limited to (0, 2879). We call this type of constraint *GUI constraint*. GUI constraint collection can be added after line 17 in Algorithm 1 and then incorporated to path condition *pc*. Adding GUI constraints to the path condition can help the concolic execution engine to generate valid inputs (*i.e.*, inputs that are accepted by the app). Generating valid inputs is important in ensuring that the errors found by our concolic execution engine correspond to real errors that could be replicated in Android devices.

Our concolic execution incorporates GUI constraints in two steps. First, we collect the GUI constraints from the Android execution at runtime. Secondly, we provide a set of templates to translate GUI constraints to constraints that are supported by the SMT solver. Defining a template is required since GUI constraints are typically defined in high-level semantics and cannot be directly used by SMT solvers. Currently, we support seven common GUI constraints

Table 2: The templates for translating GUI constraints, where m and n represent attribute values.

Source	GUI constraints	Translation
Layout	android:maxLength=n; android:digit=true	$0 \leq i$ && $i < 10^n$
	android:alpha = m	$0 \leq m$ && $m \leq 1.0$
	android:progress = m; android:max = n	$0 \leq m$ && $m \leq n$
	android:maximumAngle = m	$0 \leq m$ && $m < 360$
UI hierarchy	comboBox:numItems = n	$0 \leq i$ && $i < n$
	list:numItems = n	$0 \leq i$ && $i < n$
	bound=[0,0][m,n]	$0 \leq x < m$&&$0 \leq y < n$

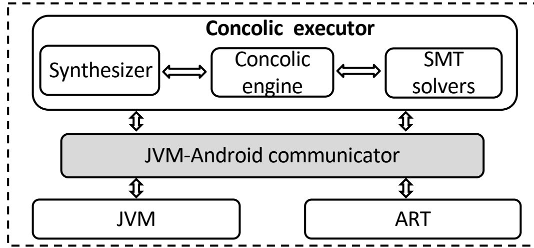

Figure 3: Architecture of our implementation SynthesiSE.

shown in Table 2. This list of supported GUI constraints can be easily extended.

5.4 Handling library functions with side effects

As capturing the effect of an Android library invocation via its return value may not be sufficient for library functions with side effects, we handle the potential imprecision caused by methods with side effects by mitigating this issue for certain Android library functions. Specifically, we analyze Android libraries that are *accessor methods* (getters) if it has a corresponding *mutator methods* (setters) that are invoked before the execution of the mutator method within the same class. For each mutator method (e.g., `setX(val)`) that changes the value of a field, we store the value in a map and we use this value for representing the subsequent accessor method (e.g., `val=getX()`). Similar strategy can also be used for library functions related to inter-process communication (Intent) and database access.

6 IMPLEMENTATION

To perform concolic execution of Android app, we reuse the concolic execution engine designed for Java programs. During concolic execution, when encountering a call to Android API (from Android framework), it communicates with Android device/emulator to obtain the runtime value for the result of the API call. If the Android library invocation satisfies the criteria defined in Section 5.1, it will trigger a synthesis process.

Figure 3 shows the architecture of our implementation SynthesiSE (implemented in Java), which comprises of three layers: concolic executor, JVM-Android communicator, and Java and Android execution environments.

Concolic executor. We implement our synthesis engine based on the recent work of [19] which embodies program synthesis via second-order constraint solving; this is partly because [19] has been successfully used for library modeling. We choose JDART [15] (GitHub commit id 6584bd0) as the concolic engine of SynthesiSE. JDART, which has been used to test industry programs, is developed as an extension to Java Pathfinder (JPF) [36]. The concolic engine can be easily replaced by other engines since our approach does not require modifications of the concolic execution engine, In SynthesiSE, we use Microsoft Z3 [8] as the SMT solver since it supports constraints containing complex arithmetic operations.

JVM-Android communicator. The Android-related statements will be delegated to the Android devices/emulators so that the Java concolic engine does not need to execute them. We implement Android execution delegator on top of the Model Java Interface (MJI[2]) component supported by JPF and JDART, which allows delegating the execution of the specified methods to the host-VM from the concolic engine. Furthermore, we leverage the Android Debug Bridge (ADB) to communicate with a device/emulator from one desktop.

Execution environments. Our concolic execution runs in the Java environment. The concrete execution of Android apps can run on real devices/emulators so that our concolic execution can analyze Android apps in a real Android environment.

7 EVALUATION

We perform evaluation on the effectiveness of SynthesiSE in synthesizing Android library methods, and its ability to enhance code coverage of symbolic execution. Our evaluation aims to address the following research questions:

RQ1 If we treat all the branches affected by an Android library invocation as targets, how many targets can we reach with our synthesized model?

RQ2 What is the quality of the synthesized library code? How many iterations are needed for our synthesis engine?

7.1 Subject selection

To compare our synthesized library models with other existing models, we evaluate our generated models against the models in JPF-Android [4]. We choose to evaluate against the models in JPF-Android[3] because (1) it contains a large number of models (a set of models developed over a course of several years); and (2) all of these models are publicly available. Some of the models in JPF-Android are manually crafted, whereas others are automatically generated using OCSEGen [33] (these automatically generated model methods will return default values). The details of our selection process are described below:

(1) As it is difficult to distinguish between manually crafted models and automatically generated models, we first obtain the set of Android library methods that have been modeled by JPF-Android. Among the supported library models, we only consider the library methods with at least one input of primitive type and output of primitive type.
(2) We randomly select 20 methods for our evaluation because we need to manually assess the quality of each synthesized library methods.
(3) For each of these methods, we search through GitHub for Android apps that invoke these methods.
(4) From the GitHub search results, we select the first app where at least one of its branches are affected by the output of the

[2]https://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/mji
[3]https://bitbucket.org/heila/jpf-android/src

Table 3: Statistics for the selected Android library calls

Method	Class	Description	App	LOC
calculateSignalLevel	WifiManager	Calculates the level of the signal.	WifiScanner	875
checkSignatures	PackageManager	Compare the signatures of two packages	CatLogDonate	3.5K
compareSignalLevel	WifiManager	Compare Signal level	GoProClose	3.4K
getAttributeResValue	AttributeSetImpl	Return the value of 'attribute' as a resource identifier.	MusicXiaMi	3.4K
getColor	ContextCompat	Retrieve the color value for the attribute at index	MyLeafPic	16K
getDefaultSize	View	Utility to return a default size constrained by input	AudioVideoRecord	12K
getDimensionPixelOff	TypedArray	Retrieve a dimensional offset in raw pixels	Marketing	738
getIndex	TypedArray	Returns an index in the array that has data.	RobolectricIssue	100
getIntExtra	Intent	Retrieve extended data from the intent.	twowayActivity	156
getLayoutDimension	TypedArray	Retrieving ViewGroup's layout_width and layout_height attributes	PrecentAdaptation	579
getOffsetForPosition	TextView	Get the character offset closest to the specified absolute position.	TagActivity	1.5K
nextSpanTransition	SpannableStringBuilder	Return the first span offset that is greater than the first parameter, or parameter itself.	SecHandTreak	3.6K
resolveSize	View	Reconcile a desired size and state, and return only masked bits	Surrounding-scanner	425
resolveSizeAndState	View	Reconcile a desired size and state constrained by input	Paper-scissors	719

corresponding library methods. We exclude six methods where the outputs of these methods do not affect any branch decision in the evaluated Android apps.

Overall, we select 14 Android apps using the process mentioned above. Table 3 shows the selected subjects as well as their statistics, where the *Class* column represents the class in which the corresponding method is defined. Meanwhile, the *Description* column gives a brief description of each method according to the Google API document. The last two columns show the open-source Android apps that use the corresponding method, and the line of code (*LOC*) of these apps.

7.2 Experimental Setup

We conduct two experiments to answer our research questions. To answer *RQ1*, we regard all the branches affected by the invocation of the given library method as *targets*. We manually generate an event sequence that can reach the method (in app code) where the corresponding library method is invoked. Based on the generated event sequence, concolic execution will generate data inputs, environment inputs, etc, to reach the *targets*. We evaluate how many *targets* can be reached with three variants of concolic execution:

Concrete Concolic execution with concretely executing Android library method (without synthesis).

SynthesiSE Concolic execution with concretely executing Android library method and on-demand synthesis.

JA Model Concolic execution with JPF-Android models.

The comparison of the first two approaches aims to investigate whether our synthesis could help concolic execution explore more affected branches. Meanwhile, the comparison with existing model evaluates existing manual/automated modeling strategy. We choose to use the JPF-Android models instead of the tool itself because (1) JPF-Android requires users to manually write the input sequence and application-specific models which would require almost one day for each subject app [4]; and (2) we need to ensure that the concolic execution engine used in all variants is the same to facilitate a fair comparison between the synthesized library methods and the JPF-Android models.

Our synthesis terminates when we find new inputs that can cover the target branch or when the number of iterations exceeds a certain limit (we use `synthesisIteration`=20 for our experiment).

To answer *RQ2*, we evaluate SynthesiSE by manually analyzing the synthesized expressions obtained from *RQ1*. Our analysis evaluates the correctness of our synthesized expressions compared to the real library and JPF-Android models.

We conduct all the experiments on a real Android device (LG G6, Android OS v7.0, API 24). For the concolic execution engine, we configure it to use Z3 for constraint solving (symbolic.dp=z3) and reuse other default configurations (symbolic.dp.z3.bitvectors=true) [15]. We run our concolic execution engine on a desktop (Ubuntu 16.04, Intel Core i7-2600 3.40GHz processor, 8GB Memory).

7.3 Results

Target reachability. As our synthesized models will only induce differences in code coverage for branches that are dependent on the results of Android library invocations, we regard those branches as *target*. Table 5 shows the number of reached *targets* for each subject by each variant. The *#Targets* column represents the total number of *targets*, whereas the *SynthesiSE*, *JA Model* and *Concrete* columns indicate the number of reached *targets* using synthesis, JPF-Android model and concrete values, respectively. If we compare the reached *targets* by SynthesiSE and concolic execution using JPF-Android model, SynthesiSE outperforms the JPF-Android models in its ability to reach more *targets* for most of the subjects. Specifically, SynthesiSE reach more *targets* in nine subjects. For the subjects marked with *NA*, SynthesiSE could not synthesize expressions to reach more *targets* as it reaches its limit during the synthesis iterations. Specifically, SynthesiSE reach one more *target* than the JPF-Android model for the *checkSignatures* method because the JPF-Android model throws an exception. For the *getIntExtra* and *compareSignalLevel* methods, both synthesis and JPF-Android provide correct models. Compared to concolic execution with concrete values, SynthesiSE reach more *targets* except for: (1) two subjects that SynthesiSE fail to synthesize expressions; (2) one subject *getDimensionPixelOff*, where some *targets* are unreachable in the single test input used for concolic execution.

Table 4: Synthesized results

Method	#synthesis	Time(s)	Synthesis Type	Synthesized	Correctness	JA model
calculateSignalLevel	3	29	Arithmetic	0.18*x1+18	C2	Constant(0)
checkSignatures	Reach Limit	336	No	No	-	Exception
compareSignalLevel	2	15	Arithmetic	x1 - x2	C1	x1 - x2
getAttributeResValue	Reach Limit	438	No	No	-	Constant(0)
getColor	13	156	Array-access	array(x1->y)	C2	x2
getDefaultSize	4	28	BitWise	x2 & 0xbffffff	C2,C3	TOP_INT
getDimensionPixelOff	15	192	Array-access	array(x1->y)	C2	x2
getIndex	14	204	Array-access	array(x1->y)	C2	Canstant(0)
getIntExtra	1	10	Set-get	put ->get	C1	Map
getLayoutDimension	12	120	Array-access	array(x1->y)	C2	x2
getOffsetForPosition	4	35	Arithmetic	x1/30-0.13	C2	TOP_INT
nextSpanTransition	2	28	Arithmetic	x2	C3	Constant(0)
resolveSize	4	42	BitWise	x2 & 0xbffffff	C2,C3	TOP_INT
resolveSizeAndState	5	54	BitWise & ITE	(x2==0)? 0:x2&0xbffffff	C2,C3	TOP_INT

Table 5: Affected branches reached with synthesized models, JPF-Android models and Concrete values

Method	#Targets	#reached targets		
		SynthesiSE	JA Model	Concrete
calculateSignalLevel	2	2	1	1
checkSignatures	2	(NA)1	(Exception)0	1
compareSignalLevel	2	2	2	1
getAttributeResValue	2	(NA)1	1	1
getColor	18	18	1	1
getDefaultSize	4	4	2	2
getDimensionPixelOff	8	4	4	4
getIndex	3	3	1	1
getIntExtra	5	5	5	1
getLayoutDimension	8	6	4	4
getOffsetForPosition	8	6	3	3
nextSpanTransition	2	2	1	1
resolveSize	4	4	1	1
resolveSizeAndState	2	2	1	1

SynthesiSE is able to reach more *targets* that are affected by library output.

Synthesis result. Table 4 shows the synthesis results, where *#synthesis* is the number of synthesis iterations and *Synthesis Type* indicate the type of synthesized expression. The *Time* column represents the time taken to infer each of the stubs, which is correlated with the number of synthesis iterations. On average, it takes 103s to synthesize the final expression. The *Synthesized* column shows the simplified expression, where x1, x2 represent the first and second parameter respectively and y represents the output. The supported synthesis type (component) includes "Constant", "Arithmetic", "BitWise", "ITE (if-then-else)" and "Array-access". We also include "Set-get" (explained in Section 5.4) as a synthesis type. Among the 14 subjects, four of the synthesized expressions are arithmetic, three of them are bitwise operation, four expressions are synthesized by Array-access, one expression synthesized using ITE, and one is handled by Set-get. Table 4 shows that the number of synthesis iterations for the "Arithmetic/Bitwise" expression is less than the "Array-access" expression because we give higher priority to Arithmetic/Bitwise component (Table 1). Meanwhile, library methods that require access to Android resources are usually synthesized using Array-access. For example, the method *getColor* which extracts color values based on resource id, is synthesized using Array-access.

SynthesiSE generates expressions for 12 out of 14 evaluated method libraries within the iteration limit.

Correctness We manually compare each synthesized expression with corresponding method in Android framework and JPF-android model. If the semantic of the synthesized expression is different from the real library, we further investigate the context-specific and conditional correctness. Given a synthesized expression lib_{syn}, and its corresponding implementation in Android framework lib_{real}, we measure the correctness of lib_{syn} using three criteria:

(C1) Correct: lib_{syn} is correct if and only if lib_{syn} and lib_{real} always produce the same semantic behavior.

(C2) Conditionally Correct: lib_{syn} is conditionally correct if lib_{syn} and lib_{real} show the same behavior for a range of inputs.

(C3) Context-specific Correct: lib_{syn} is context-specific correct if lib_{syn} and lib_{real} show the same behavior under certain context (e.g. global variables).

In the *Correctness* column, we mark two cases with "-" because our synthesis engine reaches its iteration limit for these cases. Specifically, the correct expression of *checkSignatures* and *getAttributeResValue* should involve object comparison, which our synthesis does not support. These two cases do not synthesize code which can be classified as C1, C2 or C3.

The *Correctness* column denotes the correctness category for each synthesized expression. Overall, our synthesis engine generates correct expressions for two library methods. Meanwhile, nine of the synthesized expressions are considered conditionally-correct expressions. Among these conditionally-correct expressions, three expressions are context-specific at the same time. Our synthesis engine may generate conditionally-correct expression because (1) the synthesis process terminates once new inputs exploring target path are generated, and (2) the input-output pair may not be able to cover all behaviors. If more input-output pairs are given, then our synthesis engine will be able to generate more correct expressions.

For example, the synthesized expression for *calculateSignalLevel* is (0.18*x1+18). This synthesized expression is correct for all inputs, except for cases where the real library will return 0 if x1 is less than a constant variable MIN_RSSI. Given more input-output pairs (e.g. <MIN_RSSI-1, 0>), SynthesiSE will be able to generate a more precise expression. Meanwhile, we synthesize four context-specific correct expressions. These expressions simulate the (simplified) behaviors of the library only under certain context (global states).

All of the expressions synthesized within the iteration limit are correct, conditionally-correct or context-specific correct.

7.4 Threat to validity

Internal validity: There are several internal threats in the experimental methodology that may affect our results. While there are several symbolic execution engines that rely on library models, we only compare our synthesized models against the JPF-Android models because they are publicly available and are crafted specifically for Android libraries. Our synthesis engine terminates after 20 refinement iterations, we did not investigate the effect of different iteration limits on the synthesis results. Nevertheless, for most subjects in our evaluation, a small number of synthesis iterations is sufficient to synthesize the desired expression. Moreover, we manually evaluate the correctness of the synthesized expression. As the implementation of Android framework is open-source, the analysis is relatively straightforward.

External validity: Our study is limited to the evaluated Android apps and our results may not generalize beyond the evaluated apps.

8 RELATED WORK

Symbolic/concolic execution. There exists several symbolic/concolic execution frameworks. S^2E [6] embodies the in-vivo approach to perform exploration of programs inside complex systems. Similar to our work, S^2E does not require modeling of libraries but it differs from SynthesiSE in two aspects. First, S^2E performs symbolic execution while SynthesiSE conducts concolic execution. After symbolic variables are concretized (for concrete execution), S^2E stops exploring paths constrained by these variables and uses a backtracking mechanism to alleviate this issue. SynthesiSE solves this issue by maintaining both symbolic values and concrete values. Second, SynthesiSE leverages program synthesis for deducing the relations between inputs and outputs, whereas S^2E may obtain incomplete constraints when symbolic information flows into libraries. KLEE [5] embodies the in-vitro approach which relies on manually-written POSIX libraries. Symbolic Pathfinder [25, 26] (SPF) is an extension of JPF, which uses Java library models. Different from these approaches, SynthesiSE do not model libraries, instead, we synthesize expressions to capture the effect of libraries.

Program synthesis for symbolic execution. We use program synthesis [19] which embodies program synthesis via second-order constraint solving, to synthesize a representation for Android library. To increase the scalability of program synthesis, our synthesis engine adapts the categorization of common components in prior work that synthesizes program expression to patch software errors [24]. Although the models synthesized by PASKET [13] are also used in a symbolic execution engine, its synthesis algorithm is driven by several commonly used design patterns. Meanwhile, SynthesiSE synthesizes libraries to drive the program execution to unexplored program branches and to relief the program divergence problem. Qi et al. [27] synthesize library models by sampling behaviors of the original implementation of a function. SynthesiSE synthesizes library model using input-output specifications and the feedback about branch reachability.

Android testing via symbolic/concolic execution. Extensions of Symbolic Pathfinder (e.g., JPF-Android [35], PathDroid [23] and Mirzaei's work [22]) conduct symbolic execution on Android apps in JVM by modeling Android libraries which require significant manual effort to adapt to the rapid evolution of Android versions, while SynthesiSE is designed to solve this problem. SymDroid [12] is a symbolic executor for Dalvik bytecode, while jpf-mobile [14] attempts to run JPF on Android systems. Both approaches have been evaluated only on small demo apps. Applying symbolic/concolic execution to GUI testing has been explored in several works: ACTEve [1], Collider [11], AppIntent [37], ConDroid [28], and SIGDroid [21]. Those tools uses either instrumentation or simplified models. Instrumentation based approaches (including ACTEve [1], [11], etc) may suffer from path divergence problem as missed instrumentations could lead to the divergence between the concrete and symbolic execution paths. The refinement process in SynthesiSE is able to address this problem. Approaches based on library stubs, including ConDroid [28] and SIGDroid [21] (does not consider Android framework in symbolic execution) may miss many path constraints. In constrast, SynthesiSE leverages on-demand synthesis to deduce a model to simulate Android libraries. Meanwhile, crashes found by our approach could be used as inputs for existing repair approaches for automatically fixing these crashes [20, 30–32].

9 CONCLUSION

We present *synthetic symbolic execution*, and its realization via a concolic execution engine that leverages a novel on-demand program synthesis for testing Android apps. Program synthesis is iteratively invoked to generate library code which can help achieve greater branch coverage in testing of Android apps. We believe our approach shows promise in terms of solving the rather hard problem of environment behavior capture in symbolic execution, since libraries are a form of environment. Our work shows the promise of symbolic execution techniques which neither manually model the environment, nor depend on whole system executions to capture the environment via under-approximations.

ACKNOWLEDGEMENT

This research is partially supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory at University Scheme, National University of Singapore, and Singapore Telecommunications Ltd.

REFERENCES

[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)*. ACM, Article 59, 11 pages.

[2] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA 2010)*. ACM, 49–60.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*. ACM, 259–269.

[4] Heila Botha, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. [n. d.]. Addressing Challenges in Obtaining High Coverage when Model Checking Android Applications. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN 2017)*. ACM, 31–40.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*. USENIX Association, 209–224.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems (ASPLOS 2011)*, Vol. 46. ACM, 265–278.

[7] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. IEEE, 429–440.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008/ETAPS 2008)*. Springer, 337–340.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005)*, Vol. 40. ACM, 213–223.

[10] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing. In *The Network and Distributed System Security Symposium (NDSS 2010)*, Vol. 8. 151–166.

[11] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated Testing with Targeted Event Sequence Generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, 67–77.

[12] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster. 2012. SymDroid: Symbolic execution for Dalvik bytecode.

[13] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing Framework Models for Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 156–167.

[14] Alexander Kohan, Mitsuharu Yamamoto, Cyrille Artho, Yoriyuki Yamagata, Lei Ma, Masami Hagiya, and Yoshinori Tanabe. 2017. Java Pathfinder on Android Devices. *SIGSOFT Software Engineering Notes* 41, 6 (Jan. 2017), 1–5.

[15] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*. Springer, 442–459.

[16] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *Proceedings of the 29th international conference on Software Engineering (ICSE 2007)*. IEEE, 416–426.

[17] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, 94–105.

[18] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *29th IEEE International Conference on Software Maintenance (ICSM 2013)*. IEEE, 70–79.

[19] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic Execution with Existential Second-Order Constraints. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM.

[20] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE 2016)*. IEEE, 691–701.

[21] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. SIG-Droid: Automated system input generation for Android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE 2015)*. IEEE, 461–471.

[22] 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.

[23] NASA. 2013. PathDroid. https://ti.arc.nasa.gov/opensource/projects/pathdroid/

[24] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE 2013)*. IEEE, 772–781.

[25] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE 2010)*. ACM, 179–180.

[26] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (2013), 391–425.

[27] Dawei Qi, William N Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and Abhik Roychoudhury. 2012. Modeling software execution environment. In *19th Working Conference on Reverse Engineering (WCRE 2012)*. IEEE, 415–424.

[28] Julian Schütte, Rafael Fedler, and Dennis Titze. 2015. ConDroid: Targeted Dynamic Analysis of Android Applications. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. 571–578.

[29] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 245–256.

[30] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. IEEE, 187–198.

[31] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. IEEE, 471–482.

[32] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 727–738.

[33] Oksana Tkachuk. 2013. OCSEGen: Open components and systems environment generator. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis*. ACM, 9–12.

[34] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[35] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2014. Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes* 39, 1 (2014), 1–5.

[36] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model checking programs. *Automated software engineering* 10, 2 (2003), 203–232.

[37] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS 2013)*. ACM, 1043–1054.