Android Testing via Synthetic Symbolic Execution

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

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

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

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;

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, S²E [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 \( \pi_c \) 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, \( s_j^i \) 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

\[
\varphi_i := \varphi_{node} \land \varphi_{choice}, \text{ such that } \\
\varphi_{node} := \bigwedge_{j=1}^{\lfloor C \rfloor} s_j^i \Rightarrow out_i = F_j(out_{i_1},out_{i_2},...,out_{i_k}) \tag{1}
\]

\[
\varphi_{choice} := \text{exactlyOne}(s_1^i,s_2^i,...,s_C^i) \tag{2}
\]

\( \varphi_{node} \) describes the semantic relations between the output value of node \( i \) and the output value of its subnode, while \( \varphi_{choice} \) 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
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.

\[
\% \text{branches affected} = \frac{\# \text{of branches affected}}{\text{total \# of branches}}
\]

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.

Listing 1: The onTouchEvent event handler of TagActivity

```java
public boolean onTouchEvent(MotionEvent event) {
    float x = event.getX();
    float y = event.getY();
    int offset = getOffsetForPosition(x, y);
    putOffsetInRange(offset);
    ...

    private int putOffsetInRange(int o) {
        int offset = 0;
        Editable text = getText();
        int realLength = text.length();
        if (offset == realLength)
            return offset;
        Editable editable = getText();
        while (offset >= 0 && findText(editable, offset) == -1 && findChip(offset) == null) {
            offset--;
        }
        return offset;
    }
}
```

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

To solve the above mentioned problems, SynthesisSE 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 SynthesisSE 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 \(\rightarrow\) 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 \(S^2E\), SynthesisSE does not suffer from the path missing problem due to concretization, as it performs concolic execution from the beginning of the execution. SynthesisSE 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 \(\rightarrow\) 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 \((\text{offset})\). Therefore, a function \(\delta(x, y)\) is introduced to represent the Android library call \((F)\). Meanwhile, the symbolic value of method output \((\text{offset})\) 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 \((\phi \land \delta(x, y) > 0)\) along with the concrete execution. To explore a different path, we negate the corresponding path condition and provide the generated path constraints \(\phi \land \delta(x, y) \leq 0\) 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 \((\text{offset})\). With the initialized expression \(e\), a new value \((x', y')\) will be generated by solving \(\phi \land \delta(x, y) \leq 0\land \delta = e\). 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 \(\text{GUI constraint}\), which can be missed by concretely executing library code. Therefore, we introduce a strategy to collect \(\text{GUI constraint}\) 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 \(\delta\) 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...
where the effect of calling an Android library function will be captured when our synthesis engine is triggered. To satisfy these two criteria, our synthesis engine is invoked when an Android library invocation as the target states. 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 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. 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. 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.

---

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

```plaintext
Procedure executeConcolic(program P, symbolic X)
| symbolic memory S: [x → expression | x ∈ X];
| concrete memory C: [x → value | x ∈ X];
| path condition pc: ∅;
| library functions Λ = [ ];
stmt := stmt0; // Initial program statement
while stmt ≠ Exits do
| switch stmt do
  | case y := libraryInvocation(v0, v1, ..., vn) do
    | C := executeConcrete(stmt);
    | D := {v | v ∈ {v0, v1, ..., vn} ∧ S[v] ≠ null};
    | if isEmpty(D) then
      | s = δ((S[d] | d ∈ D));
      | S = S[y → s];
      | Λ = Λ ∪ δ;
    | else
      | executeJavaConcolic(stmt, pc);
    | end
  | end
stmt := P getNextstmt();
end
return (pc, Λ);
```

**Procedure pathExploration(program P)**

X := init();
while X ≠ null do
  (pc, Λ) := executeConcolic(P, X);
  // select a condition to negate
  pc′ := select(pc);
  pc′ := ¬pc′ ∧ pc ∨ pc′;
  Θ = {δ | δ ∈ Λ ∧ contains(pc′, δ)};
  if isEmpty(Θ) then
    X := synthesize(pc′, Θ);
  else
    X = solve(pc′);
end

---

**Algorithm 2: Iterative Synthesis**

```plaintext
Procedure synthesize(program P, pathCond pc, func δ)
| // < in0, out0 > from initial execution
| e = out0;
| R = {⟨in0, out0⟩};
synthesisIteration = 0;
while synthesisIteration < L do
|x′ := solve(pc ∧ δ = e);
if UNSAT or UNKNOWN then
  x′ := solve(pc ∧ δ ≠ e);
if UNSAT or UNKNOWN then
  return null // unsatisfiable pc
end
⟨π, r0⟩ := concreteExecution(P, x′);
R = R ∪ {r0};
if π == target then
  return x′;
else
  e := componentBasedSynthesis(R);
synthesisIteration + +;
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(v0, v1, ..., vn) where the effect of calling an Android library function will be captured through its return values1. 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).

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.

---

1 A function where its return value is not stored is rewritten with new temporary variable that store the return value.
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, \( \delta \) is the path constraint before the invocation of \( F(x, y) \), and \( x_1 \) and \( x_2 \) is obtained by solving \( \phi \land \delta(x, y) \leq 0 \land \delta = e \). Since \( x_1 \) and \( x_2 \) satisfy the path constraint \( \phi \), 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 1: The categorization of synthesis components

<table>
<thead>
<tr>
<th>Level</th>
<th>Type</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>constants</td>
</tr>
<tr>
<td>2</td>
<td>BitWise</td>
<td>(&lt;, &gt;, &amp;),</td>
</tr>
<tr>
<td>3</td>
<td>Arithmetic</td>
<td>+, -, &quot;, /</td>
</tr>
<tr>
<td>4</td>
<td>Flow-control</td>
<td>Ite (If-then-else)</td>
</tr>
<tr>
<td>5</td>
<td>Array-access</td>
<td>array</td>
</tr>
</tbody>
</table>

pair \((ino, outo)\) from the initial concrete execution of an Android library function, the \textit{synthesize} procedure starts by initializing the synthesized expression \( e \) to the output \( ou_0 \) (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 \( pc \land \delta \neq e \). 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 \( r_{ia} \) (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 \( r_{ia} \) 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 \((pc \land (\delta = e))\) 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

Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury
Table 2: The templates for translating GUI constraints, where \( m \) and \( n \) represent attribute values.

<table>
<thead>
<tr>
<th>Source</th>
<th>GUI constraints</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout</td>
<td>android:maxLength=n, android:digit=true</td>
<td>( 0 \leq m \leq n )</td>
</tr>
<tr>
<td></td>
<td>android:alpha = m</td>
<td>( 0 \leq m \leq 1 )</td>
</tr>
<tr>
<td></td>
<td>android:progress = m, android:max = n</td>
<td>( 0 \leq m \leq m \leq n )</td>
</tr>
<tr>
<td></td>
<td>android:maximumAngle = m</td>
<td>( 0 \leq m \leq m \leq 360 )</td>
</tr>
<tr>
<td>UI hierarchy</td>
<td>bound=[0,0][m,n]</td>
<td>( 0 \leq x \leq m, 0 \leq y \leq n )</td>
</tr>
</tbody>
</table>

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., set\( x \)(\( val \))), 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., get\( x \)(\( val \))). 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 code does not need to execute them. We implement Android execution delegator on top of the Model Java Interface (MJI) 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-Android2 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. 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

---

2https://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/mji
3https://bitbucket.org/heila/jpf-android/src
We conduct two experiments to answer our research questions. To whether our synthesis could help concolic execution explore more

class API document. The last two columns show the open-source An-

ASE ’18, September 3–7, 2018, Montpellier, France Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury

Table 3: Statistics for the selected Android library calls

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

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.

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.

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.
Among the 14 subjects, four of the synthesized expressions are arithmetic, three of them are bitwise operation, four expressions are synthesized using 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.

### 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 always produce the same semantic 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 \texttt{calculateSignalLevel} is \((0.18 \times x1 + 18)\). 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 \texttt{MIN_RSSI}. Given more input-output pairs (e.g., \(<\texttt{MIN_RSSI}=1, 0>\)), 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).

<table>
<thead>
<tr>
<th>All of the expressions synthesized within the iteration limit are correct, conditionally-correct or context-specific correct.</th>
</tr>
</thead>
</table>

### 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.

### 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


Android Testing via Synthetic Symbolic Execution


