Automated Concolic Testing of Smartphone Apps

Saswat Anand[1] Mayur Naik[1] Hongseok Yang[2] Mary Jean Harrold[1]

Georgia Institute of Technology[1]
{saswat,naik,harrold}@cc.gatech.edu

University of Oxford[2]
hongseok.yang@cs.ox.ac.uk

ABSTRACT

We present an algorithm and a system for generating input events to exercise smartphone apps. Our approach is based on concolic testing and generates sequences of events automatically and systematically. It alleviates the path-explosion problem by checking a condition on program executions that identifies subsumption between different event sequences. We also describe our implementation of the approach for Android, the most popular smartphone app platform, and the results of an evaluation that demonstrates its effectiveness on five Android apps.

1. INTRODUCTION

Mobile devices with advanced computing ability and connectivity, such as smartphones and tablets, are becoming increasingly prevalent. At the same time there has been a surge in the development and adoption of specialized programs, called *apps*, that run on such devices. Apps pervade virtually all activities ranging from leisurely to mission-critical. Thus, there is a growing need for software-quality tools in all stages of an app's life-cycle, including development, testing, auditing, and deployment.

Apps have many features that make static analysis challenging: a vast SDK (Software Development Kit), asynchrony, inter-process communication, databases, and graphical user interfaces (GUIs). As a result, many proposed approaches for analyzing apps are based on dynamic analysis (e.g., [6, 8, 9]).

A question central to the effectiveness of any dynamic analysis is how to obtain relevant program inputs. The most indivisible and routine kind of inputs to an app are *events*. A tap on the device's touch screen, a key press on the device's keyboard, and an SMS message are all instances of events. This paper presents an algorithm and a system for generating input events to exercise apps. Apps can—and in practice often do—possess inputs besides events, such as files on disk and secure web content. Our work is orthogonal and complementary to approaches that provide such inputs.

Apps are instances of a class of programs we call *event-driven programs*: programs embodying computation that is architected to react to a possibly unbounded sequence of events. Event-driven programs are ubiquitous and, besides apps, include stream-processing programs, web servers, GUIs, and embedded systems. Formally, we address the following problem in the setting of event-driven programs in general, and apps in particular.

> **The Problem:** Given a constant bound $k \geq 1$, efficiently compute a set of event sequences that covers each program statement reachable on some event sequence of length up to k.

The above problem poses two separate challenges: (1) how to generate single events and (2) how to extend them to sequences of events. We next look at each of these in turn.

Generating Single Events. Existing approaches for generating all events of a particular kind use either *capture-replay* techniques to automatically infer a model of the app's GUI [22, 30] or *model-based* techniques that require users to provide the model [33, 35]. These approaches have limitations. Capture-replay approaches are tailored to a particular platform's event-dispatching mechanism but many apps use a combination of the platform's logic and their own custom logic for event dispatching. For example, where the platform sees a single physical widget, an app might interpret events dispatched to different logical parts of that widget differently. In contrast, model-based approaches are general-purpose, but they can require considerable manual effort.

We developed a general, fully-automatic solution to this problem, that builds on a systematic test-generation technique, *concolic testing*, which has made significant strides in recent years [5, 14, 29]. Our concolic-testing approach symbolically tracks events from the point where they originate to the point where they are ultimately handled. Our solution is thus, oblivious to where and how events are dispatched.

Generating Event Sequences. Our concolic-testing approach for generating single events can be extended naturally to iteratively compute sets of increasingly longer sequences of events. But the classic concolic testing approach [14, 29], hereafter called Classic, causes the computed sets to grow rapidly. For instance, for a music player app, Classic produces 11 one-event sequences, 128 two-event sequences, 1,590 three-event sequences, and 21K four-event sequences. The reason is that it tries to generate a set of inputs P such that each program path is executed by a different input in P. Each reachable program statement is covered by some input in P but the number of paths is typically exponential in program size. Thus, Classic suffers from the *path-explosion problem*, and in practice it is capable of exploring only a small subset of paths.

Significant advances have been made in recent years to tackle path explosion (e.g., [1, 3, 11–13, 16, 18, 20, 28]). To be effective in practice, however, a technique can exploit characteristics of its target class of programs. For instance, concolic testing of a program with complex structured inputs (e.g., a compiler) requires a grammar-based specification of valid inputs, to go beyond shallow exploration of the initial parsing stages of the program.

We seek to bring the benefits of concolic testing to event-driven programs—an increasingly prevalent and significant class of programs that includes apps—by proposing a novel way to alleviate the path-explosion problem tailored to this class of programs. Our key insight is a novel notion of sub-

sumption between different event sequences. If an event sequence π is *subsumed* by another event sequence π', then pruning π in a particular iteration of our algorithm prevents extensions of π from being considered in any future iteration, thereby providing compounded savings, while still ensuring completeness with respect to CLASSIC due to the presence of π'.[1] Specifically, we prove that our algorithm reaches a program statement in k iterations if and only if CLASSIC reaches it in k iterations.[2]

Our subsumption condition involves checking simple data- and control-flow facts about program executions. Being fully dynamic, it is applicable to real programs that often contain parts that are beyond the scope of static analysis. Moreover, the condition is inexpensive to check and occurs frequently in real apps, causing the savings to greatly offset the cost of checking it. For example, for the previously-mentioned music player app, using $k = 4$, our algorithm explores only 16% of the inputs (3,445 out of 21,117) that CLASSIC explores.

We have implemented our algorithm in a system for Android, the dominant smartphone app platform. Our system instruments and exercises the given app in a mobile device emulator that runs an instrumented Android SDK. This enables our system to be portable across mobile devices, leverage Android's extensive tools (e.g., to automatically run the app on generated inputs), and exploit stock hardware; in particular, our system uses any available parallelism, and can run multiple emulators on a machine or a cluster of machines. Our system also builds upon recent advances in concolic testing such as function summarization [11], generational search [15], and SMT solving [7]. We demonstrate the effectiveness of our system on five Android apps.

We summarize the primary contributions of our work.

1. A novel way to systematically generate events to exercise apps. Our approach, based on concolic testing, is fully automatic and general, in contrast to existing model-based or capture-replay-based approaches.
2. A concolic-testing algorithm to efficiently generate sequences of events. Our key insight is a subsumption condition between event sequences. Checking the condition enables our algorithm to prune redundant event sequences, and thereby alleviate path explosion, while being complete with respect to CLASSIC.
3. An implementation and evaluation of our algorithm in a system for Android, the dominant smartphone app platform. Our system is portable, exploits available parallelism, and leverages recent advances in concolic testing. We demonstrate the effectiveness of our system on five Android apps.

2. OVERVIEW OF OUR APPROACH

In this section, we illustrate our approach using an example music player app from the Android distribution. We first describe the app by discussing its source code shown in Figure 1 (Section 2.1). We then describe how we generate events to exercise this app (Section 2.2) and how we extend them to sequences of events (Section 2.3).

[1] More precisely, instead of concrete event sequences, our algorithm works on symbolic counterparts.

[2] Our algorithm can also be used for bounded verification of safety properties, by converting a given `assert(e)` statement to `if (e) assert(false)` and checking reachability of the `assert(false)` statement.

```
public class MainActivity extends Activity {
    Button mRewindButton, mPlayButton, mEjectButton, ...;
    public void onCreate(...) {
        setContentView(R.layout.main);
        mPlayButton = findViewById(R.id.playbutton);
        mPlayButton.setOnClickListener(this);
        ... // similar for other buttons
    }
    public void onClick(View target) {
        if (target == mRewindButton)
            startService(new Intent(ACTION_REWIND));
        else if (target == mPlayButton)
            startService(new Intent(ACTION_PLAY));
        ... // similar for other buttons
        else if (target == mEjectButton)
            showUrlDialog();
    }
}
public class MusicService extends Service {
    MediaPlayer mPlayer;
    enum State { Retrieving, Playing, Paused, Stopped, ... };
    State mState = State.Retrieving;
    public void onStartCommand(Intent i, ...) {
        String a = i.getAction();
        if (a.equals(ACTION_REWIND)) processRewind();
        else if (a.equals(ACTION_PLAY)) processPlay();
        ... // similar for other buttons
    }
    void processRewind() {
        if (mState == State.Playing || mState == State.Paused)
            mPlayer.seekTo(0);
    }
}
```

Figure 1: Source code snippet of music player app.

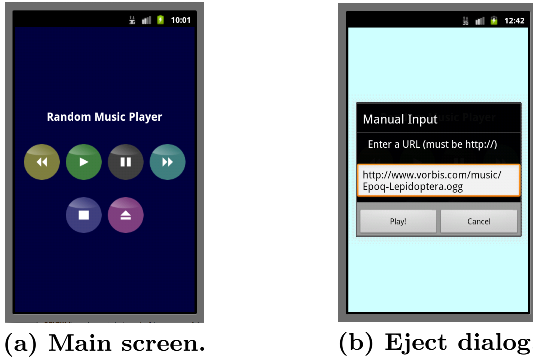
2.1 The Music Player App

Android apps are incomplete programs: they implement parts of the Android SDK's API and lack a main method. When the music player app is started, the SDK creates an instance of the app's main activity `MainActivity`, and calls its `onCreate()` method. This method displays the main screen, depicted in Figure 2(a), which contains six buttons: rewind, play, pause, skip, stop, and eject. The method also sets the main activity as the handler of clicks to each of these buttons. The app waits for events once `onCreate()` finishes.

When any of the six buttons is clicked, the SDK calls the main activity's `onClick()` method, since the main activity was set to handle these clicks. If the eject button is clicked, this method displays a dialog, depicted in Figure 2(b), that prompts for a music file URL. If any of the other buttons is clicked, the `onClick()` method starts a service `MusicService` with an argument that identifies the button. The service processes clicks to each of the five buttons. For brevity, we only show how clicks to the rewind button are processed, in the `processRewind()` method. The service maintains the current state of the music player in `mState`. Upon startup, the service searches for music files stored in the device, and hence the state is initialized to `Retrieving`. Upon completion of the search, the state is set to `Stopped`. Clicks to each button have effect only in certain states; for instance, clicking the rewind button has no effect unless the state is `Playing` or `Paused`, in which case `processRewind()` rewinds the player to the start of the current music file.

2.2 Generating Single Events

Our first goal is to systematically generate single input events to a given app in a given state. For concreteness, we focus on tap events, which are taps on the device's touch screen, but our observations also hold for other kinds of



(a) Main screen. (b) Eject dialog.

Figure 2: Screen shots of music player app.

events, such as key presses on the device's keyboard and incoming phone calls.

As for many apps, the primary form of input to the music player app is button taps. Our goal is to generate tap events such that each widget on the displayed screen of the app is clicked once. In Android, the widgets on any screen of an app are organized in a tree called the *view hierarchy*,[3] where each node denotes the rectangular bounding box of a different widget, and the node's parent denotes its containing widget. Figure 3 shows the view hierarchy for the main screen of the music player app depicted in Figure 2(a). Given this hierarchy, we can achieve our goal of clicking each widget once, by generating Cartesian coordinates inside each rectangle in the hierarchy and outside its sub-rectangles.

As we discussed in the Introduction, existing approaches either infer the hierarchy automatically (capture-replay [22, 30]) or require users to provide it (model-based approaches [33, 35]); both have limitations. Capture-replay approaches are ad hoc: although the music player app uses only SDK-provided widgets, many apps also use custom compound widgets and interpret clicks to different components within such widgets differently. The view hierarchy conflates such logically distinct widgets into a single physical widget, and stymies our goal of clicking all widgets. Model-based approaches allow faithful modeling of a GUI's logical components but require substantial manual effort for each app.

We propose a radically different approach that is general and fully automatic. Our approach is based on concolic testing. It symbolically tracks events from the point where they originate to the point where they are handled. For this purpose, our approach instruments the Android SDK and the app under test. In the case of tap events, whenever a concrete tap event is input, this instrumentation creates a fresh symbolic tap event and propagates it alongside the concrete event. As the concrete event flows through the SDK and the app, the instrumentation tracks a constraint on the corresponding symbolic event which effectively identifies all concrete events that are handled in the same manner. This not only lets our approach avoid generating spurious events but also enables it to exhaustively generate orthogonal events. For the main screen of our music player app, for instance, our approach generates exactly 11 tap events, one in each of the rectangles (and outside sub-rectangles) in the screen's view hierarchy depicted in Figure 3. Section 3 describes how our approach generates these events in further detail.

[3] Other platforms, e.g., IPhone, have an analogous concept.

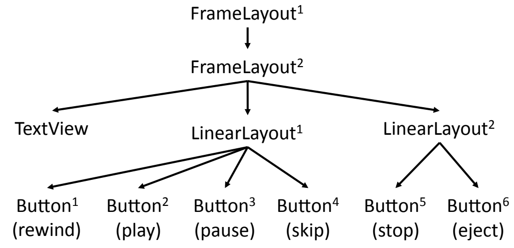


Figure 3: View hierarchy of main screen.

2.3 Generating Event Sequences

Our concolic testing approach for generating single events can be extended naturally to iteratively compute sets of increasingly longer sequences of events. However, as we discussed in the Introduction, the Classic concolic testing approach [14, 29] causes the computed sets to grow rapidly—for the above music player app, Classic produces 11 one-event sequences, 128 two-event sequences, 1,590 three-event sequences, and 21K four-event sequences.

We studied the event sequences produced by Classic for several Android apps and observed a significant source of redundancy. Conceptually, each app may be viewed as an automaton: the app is in a particular state at any instant, and each event that is dispatched to the app (e.g., by our above-described approach for generating single events) may be viewed as a state transition. We observed empirically that, for many states, most events have no effect in that the state remains unchanged (or, equivalently, the state transitions to itself upon those events). We call such events *read-only* because we identify them by checking that no memory location is written when they are dispatched and handled. Upon closer inspection, we found diverse reasons for the prevalence of read-only events, which we describe below.

First, many widgets on any given screen of any app *never* react to any clicks. As shown in the view hierarchy of the main screen of the music player app (Figure 3), these include boilerplate widgets like `FrameLayout` and `LinearLayout`, which merely serve to layout other actionable widgets, such as `Button`, and informational widgets, such as `TextView`, that display uneditable text. Thus, only 6 of 11 widgets on the main screen of the music player app (namely, the six buttons) are actionable, and clicks to the remaining 5 widgets constitute read-only events.

Second, many widgets that are actionable might be disabled in certain states of the app. This situation often occurs when apps wish to guide users to provide events in a certain order or when they wish to prevent users from providing undefined combinations of events.

Third, GUI-based programs are conservative in updating state. In particular, they avoid unnecessary updates, to suppress needlessly re-drawing widgets and notifying listeners. For instance, if an event wishes to set a widget to a state γ, then the event handler for that widget reads the current state of the widget, and does nothing if the current state is already γ, effectively treating the event as read-only.

Finally, many apps simply do not react to the vast majority of event types. The Android platform defines hundreds of different types of events to which apps can choose to react, such as SMS received event and power connected event. It is tedious at best and infeasible at worst to infer upfront the set of all event types to which a given app may react (e.g., it can

require static analysis of the SDK and the app). Instead, it is convenient to simply send each type of event to the given app in each state, and observe dynamically whether the app reacts to it (namely, whether any memory location is written). If the app does not react—the common case—then our approach classifies the event as a read-only event.

Based on the above observations, we propose a novel way to alleviate the path-explosion problem tailored to apps: our approach does not extend event sequences that end in a read-only event. Pruning such sequences in a particular iteration of our approach prevents extensions of those sequences from being considered in future iterations, thereby providing compounded savings, while still ensuring completeness with respect to the Classic concolic testing approach.

We go even further and show that the read-only pattern is an instance of a more general notion of *subsumption* between event sequences. Specifically, we show that an event sequence that ends in a read-only event is subsumed by the same event sequence without that final event.

For our example music player app, using read-only subsumption, our approach explores 3,445 four-event sequences, compared to 21,117 by the Classic approach, which does not check for subsumption but is identical in all other respects. Besides the fact that clicks to many passive widgets in this app (e.g., `LinearLayout`) are read-only, another key reason for the savings is that even clicks to actionable widgets (e.g., `Button`) are read-only in many states of the app. For instance, consider the two-event sequence [stop, rewind]. The first event (stop) writes to many memory locations (e.g., fields `mPlayer` and `mState` in class `MusicService` shown in Figure 1). However, the second event (rewind) does not write to any location, because the `processRewind()` method of class `MusicService` that handles this event only writes if the state of the music player is `Playing` or `Paused`, whereas after the first stop event, its state is `Stopped`. Thus, our approach identifies the rewind event in sequence [stop, rewind] as read-only, and prunes all sequences explored by Classic that have this sequence as a proper prefix.

Section 4 presents a formal description of subsumption, the read-only instantiation of subsumption, and the formal guarantees it provides.

3. GENERATING SINGLE EVENTS

In this section, we describe how our approach systematically generates single events. We use tap events in Android as a proof-of-concept. Tap events are challenging to generate because they are continuous and have more variability than discrete events such as incoming phone calls, SMS messages, and battery charging events. Moreover, tap events are often the primary drivers of an app's functionality, and thus, control significantly more code of the app than other kinds of events. The principles underlying our approach, however, are not specific to tap events or to Android.

We begin by describing how Android handles tap events. Figure 4 shows the simplified code of the `dispatchEvent()` method of SDK class `android.view.ViewGroup`. When a tap event is input, this method is called recursively on widgets in the current screen's view hierarchy, to find the innermost widget to which to dispatch the event, starting with the root widget. If the event's coordinates lie within a widget's rectangle, as determined by the `contains()` method of SDK class `android.graphics.Rect`, then `dispatchEvent()` is called on children of that widget, from right to left, to determine whether the current widget is indeed the innermost one containing the event or if it has a descendant containing the event. For instance, any tap event that clicks on the pause button on the main screen of our example music player app results in testing for the event's containment in the following widgets in order, as the event is dispatched in the view hierarchy depicted in Figure 3. We also indicate whether or not each test passes: `FrameLayout`^{1} (yes) $\rightarrow$ `FrameLayout`^{2} (yes) $\rightarrow$ `LinearLayout`^{2} (no) $\rightarrow$ `LinearLayout`^{1} (yes) $\rightarrow$ `Button`^{4} (no) $\rightarrow$ `Button`^{3} (yes).

```
public class android.view.ViewGroup {
    public boolean dispatchEvent(Event e) {
        float x = e.getX(), y = e.getY();
        for (int i = children.length - 1; i >= 0; i--) {
            View child = children[i];
            if (child.contains(x, y))
                if (child.dispatchEvent(e))
                    return true;
        }
        return false;
    }
}
public class android.graphics.Rect {
    public boolean contains(float x, float y) {
        return x >= this.left && x < this.right &&
               y >= this.top && y < this.bottom;
    }
}
```

Figure 4: Source code snippet of Android SDK.

Our approach uses concolic testing to generate a separate tap event to each widget. This requires symbolically tracking events from the point where they originate to the point where they are handled. Let $(\$x, \$y)$ denote variables that our approach uses to symbolically track a concrete tap event (x, y). Then, for each call to `contains`(x, y) on a rectangle in the view hierarchy specified by constants $(x_{left},\ x_{right},\ y_{top},\ y_{bottom})$, our approach generates the following constraint or its negation, depending upon whether or not the tap event is contained in the rectangle:

$$(x_{left} \leq \$x < x_{right}) \ \wedge \ (y_{top} \leq \$y < y_{bottom})$$

Our approach starts by sending a random tap event to the current screen of the given app. For our example app, suppose this event clicks the pause button. Then, our approach generates the following *path constraint*:

$$\begin{array}{llr} & (0 \leq \$x < 480) \ \wedge \ (0 \leq \$y < 800) & // \ c_1 \\ \wedge & (0 \leq \$x < 480) \ \wedge \ (38 \leq \$y < 800) & // \ c_2 \\ \wedge & \$x' = \$x \ \wedge \ \$y' = (\$y - 38) & // \ p_1 \\ \wedge & \neg((128 \leq \$x' < 352) \ \wedge \ (447 \leq \$y' < 559)) & // \ c_3 \\ \wedge & (16 \leq \$x' < 464) \ \wedge \ (305 \leq \$y' < 417) & // \ c_4 \\ \wedge & \$x'' = (\$x' - 16) \ \wedge \ \$y'' = (\$y' - 305) & // \ p_2 \\ \wedge & \neg((344 \leq \$x'' < 440) \ \wedge \ (8 \leq \$y'' < 104)) & // \ c_5 \\ \wedge & (232 \leq \$x'' < 328) \ \wedge \ (8 \leq \$y'' < 104) & // \ c_6 \end{array}$$

Constraints c_1 and c_2 capture the fact that the event is tested for containment in `FrameLayout`^{1} and `FrameLayout`^{2}, respectively, and the test passes in both cases. The event is then tested against `LinearLayout`^{2} but the test fails (notice the negation in c_3). Constraints c_4 through c_6 arise from testing the event's containment in `LinearLayout`^{1}, `Button`^{4} (the skip button), and `Button`^{3} (the pause button). We explain constraints p_1 and p_2 below.

Our approach next uses this path constraint to generate concrete tap events to other widgets. Specifically, for each c_i, it uses an off-the-shelf constraint solver to solve the constraint $(\bigwedge_{j=1}^{i-1} c_j) \wedge \neg c_i$ for $\$x$ and $\$y$. If this constraint is

$$
\begin{array}{rrl}
\text{(condition label)} & l & \in \ \mathsf{Label} \qquad \text{(input variable)}\ a \\
\text{(global variable)} & g & \in \ \mathsf{GVar} = \{g_1, \ldots, g_m\} \\
\text{(expression)} & e & ::= a \mid g \mid aop(\bar{e}) \\
\text{(boolean expression)} & b & ::= bop(\bar{e}) \mid \mathsf{True} \mid \mathsf{False} \mid \\
 & & \quad \neg b \mid b \wedge b \mid b \vee b \\
\text{(program)} & s & ::= \mathsf{skip} \mid g = e \mid s_1; s_2 \mid \\
 & & \quad \mathsf{if}\ b^l\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{while}\ b^l\ s
\end{array}
$$

Figure 5: Syntax of programs.

satisfiable, any solution the solver provides is a new concrete tap event guaranteed to take the path dictated by this constraint in the view hierarchy. That path in turn generates a new path constraint and our approach repeats the above process until all widgets in the hierarchy are covered.

We now explain the role of constraints p_1 and p_2 in the path constraint depicted above. These constraints introduce new symbolic variables $\$x'$, $\$y'$, $\$x''$, and $\$y''$. They arise because, as a tap event is dispatched in the Android SDK, various offsets are added to its concrete x and y coordinates, to account for margins, convert from relative to absolute positions, etc. The already-simplified path constraint depicted above highlights the complexity for concolic execution that a real platform like Android demands: we instrument not only the SDK code shown in Figure 4 but *all* SDK code, as well as the code of each app under test. Dropping any of the above constraints due to missed instrumentation can result in the notorious *path divergence* problem [15] in concolic testing, where the concrete and symbolic values diverge and threaten the ability to cover all widgets.

4. GENERATING EVENT SEQUENCES

In this section, we describe how our approach generates sequences of events. To specify our approach fully and to express and prove the formal guarantee of the approach, we use a simple imperative language, which includes the essential features of Android apps. We begin with the explanation of our language and the associated key semantic concepts (Sections 4.1 and 4.2). We then describe our algorithm, proceeding from the top-level routine (Section 4.3) to the main optimization operator (Sections 4.4 and 4.5). Finally, we discuss the formal completeness guarantee of our algorithm (Section 4.6). (Appendix B gives the proofs of all lemmas and the theorem discussed in this section.)

4.1 Core Language

Our programming language is a standard WHILE language with one fixed input variable a and multiple global variables $g_1,\ldots,g_m$ for some fixed m. A program s models an Android app, and it is meant to run repeatedly in response to a sequence of input events provided by an external environment, such as a user of the app. The global variables are threaded in the repetition, so that the final values of these variables in the i-th iteration become the initial values of the variables in the following $(i+1)$-th iteration. In contrast, the input variable a is not threaded, and its value in the i-th iteration comes from the i-th input event. Other than this initialization in each iteration, no statements in the program s can modify the input variable a.

The syntax of the language appears in Figure 5. For simplicity, the language assumes that all the input events are given by integers and stored in the input variable a. It allows such an event in a to participate in constructing complex expressions e, together with global variable g and the

$$
\begin{array}{rrcl}
\text{(integer)} & n & \in & \mathsf{Integers} \\
\text{(global state)} & \gamma & ::= & [g_1 : n_1, \ldots, g_m : n_m] \\
\text{(symbolic global state)} & \Gamma & ::= & [g_1 : e_1, \ldots, g_m : e_m] \\
\text{(branching decision)} & d & ::= & \langle l, \mathit{true} \rangle \mid \langle l, \mathit{false} \rangle \\
\text{(instrumented constraint)} & c & ::= & b^d \\
\text{(path constraint)} & C & ::= & c_1 c_2 \ldots c_k \\
\text{(concolic state)} & \omega & ::= & \langle \gamma, \Gamma, C \rangle \\
\text{(input event sequence)} & \pi & ::= & n_1 n_2 \ldots n_k \\
\text{(set of globals)} & W & \subseteq & \{g_1, \ldots, g_m\} \\
\text{(trace)} & \tau & ::= & \langle C_1, W_1 \rangle \ldots \langle C_k, W_k \rangle
\end{array}
$$

Figure 6: Semantic domains.

application of an arithmetic operator $aop(\bar{e})$, such as $a + g$ and 3. Boolean expressions b combine the expressions using standard comparison operators, such as $=$ and $\leq$, and build conditions on program states. Our language allows five types of programming constructs with the usual semantics: skip for doing nothing; assignments to globals $g = e$; sequential compositions $(s_1; s_2)$; conditional statements (if b^l s_1 else s_2) with an l-labeled boolean b; and loops (while b^l s). Note that although the input variable a can appear on the RHS of an assignment, it is forbidden to occur on the LHS. Thus, once initialized, the value of a never changes during the execution of a program. Note also that all boolean conditions are annotated with labels l. We require the uniqueness of these labels. The labels will be used later to track branches taken during the execution of a program.

Example 1. *The following program is a simplified version of the music player app in our language:*

```
if (g==Stopped)^l0 {
    if (a==Play)^l1 {g = Playing}
    else if (a==Skip)^l2 {g = Skipping} else {skip}
} else {
    if (a==Stop)^l3 {g = Stopped} else {skip}
}
```

To improve the readability, we use macros here: Stopped = Stop = 0*,* Playing = Play = 1*, and* Skipping = Skip = 2*. Initially, the player is in the* Stopped *state (which is the value stored in g), but it can change to the* Playing *or* Skipping *state in response to an input event. When the player gets the* Stop *input event, the player's state goes back to* Stopped*.*

We write Globals(e) and Globals(s) to denote the set of free global variables appearing in e and s, respectively.

Throughout the rest of the paper, we fix the input program and the initial global state given to our algorithm, and denote them by s_{in} and γ_{in}.

4.2 Semantic Domains

We interpret programs using a slightly non-standard operational semantics, which describes the concolic execution of a program, that is, the simultaneous concrete and symbolic execution of the program. Figure 6 summarizes the major semantic domains. The most important are those for concolic states ω, input sequences π, and traces τ.

A ***concolic state*** ω specifies the status of global variables concretely as well as symbolically. It consists of the three components, denoted by $\omega.\gamma$, $\omega.\Gamma$, and $\omega.C$, respectively. The γ component keeps the concrete values of all the global variables, while the Γ component stores the symbolic values of them, specified in terms of expressions. We require that global variables should not occur in these symbolic values; only the input variable a is allowed to appear there.

The C component is a sequence of instrumented constraints $c_1 c_2 \ldots c_k$, where each c_i is a boolean expression b annotated with a label and a boolean value. As for symbolic values, we prohibit global variables from occurring in b. The annotation indicates the branch that generates this boolean value as well as the branching decision observed during the execution of a program.

An ***input event sequence*** π is just a finite sequence of integers, where each integer represents an input event from the environment.

A ***trace*** τ is also a finite sequence, but its element consists of a path constraint C and a subset W of global variables. The element $\langle C, W \rangle$ of τ expresses what happened during the concolic execution of a program with a *single* input event (as opposed to an event sequence). Hence, if τ is of length k, it keeps the information about event sequences of length k. The C part describes the symbolic path constraint collected during the concolic execution for a single event, and the W part stores variables written during the execution. As in the case of concolic state, we adopt the record selection notation, and write $(\tau_i).C$ and $(\tau_i).W$ for the C and W components of the i-th element of τ. Also, we write $\tau\langle C, W\rangle$ to mean the concatenation of τ with a singleton trace $\langle C, W \rangle$.

Our operational semantics defines two evaluation relations: (1) $\langle s, n, \omega \rangle \downarrow \omega' \triangleright W$ and (2) $\langle s, \pi, \gamma \rangle \Downarrow \gamma' \triangleright \tau$. The first relation models the run of s with a single input event n from a concolic initial state ω. It says that the outcome of this execution is ω', and that during the execution, variables in W are written. We point out that the path constraint $\omega'.C$ records all the branches taken during the execution of a program. If the execution encounters a boolean condition b^l that evaluates to True, it still adds $\mathsf{True}^{\langle l, true\rangle}$ to the C part of the current concolic state, and remembers that the true branch is taken. The case that b^l evaluates to False is handled similarly.

The second relation describes the execution of s with an input event sequence. It says that if a program s is run repeatedly for an input sequence π starting from a global state γ, this execution produces a final state γ', and generates a trace τ, which records path constraints and written variables during the execution. Note that while the first relation uses concolic states to trace various symbolic information about execution, the second uses traces for the same purpose.

The rules for the evaluation relations mostly follow from our intended reading of all the parts in the relations. They are given in Appendix A.

Recall that we fixed the input program and the initial global state and decided to denote them by s_{in} and γ_{in}. We say that a trace τ is ***feasible*** if τ can be generated by running s_{in} from γ_{in} with some event sequences, that is,

$$\exists \pi, \gamma'.\ \langle s_{in}, \pi, \gamma_{in} \rangle \Downarrow \gamma' \triangleright \tau.$$

Our algorithm works on feasible traces, as we explain next.

4.3 Algorithm

Our algorithm CONTEST takes a program, an initial global state, and an upper bound k on the length of event sequences to explore. By our convention, s_{in} and γ_{in} denote these program and global state. Then, CONTEST generates a set Σ of feasible traces of length up to k, which represents event sequences up to k that achieve the desired code coverage.

Formally, the output Σ of our algorithm satisfies two correctness conditions.

Algorithm 1 Algorithm CONTEST

```
INPUTS: Program s_in, global state γ_in, bound k ≥ 1.
OUTPUTS: Set of traces of length up to k.
Π_0 = Δ_0 = {ε}
for i = 1 to k do
  Δ_i = symex(s_in, γ_in, Π_{i-1})
  Π_i = prune(Δ_i)
end for
return ⋃_{i=0}^{k} Δ_i
```

1. First, all traces in Σ are feasible. Every $\tau \in \Sigma$ can be generated by running s_{in} with some event sequence π of length up to k.
2. Second, Σ achieves the full coverage in the sense that if a branch of s_{in} is covered by an event sequence π of length up to k, we can find a trace τ in Σ such that every event sequence π' satisfying τ (i.e., $\pi' \models \tau$) also covers the branch.

The top-level routine of CONTEST is given in Algorithm 1. The routine repeatedly applies operations symex and prune in alternation on sets Π_i and Δ_i of traces of length i, starting with the set containing only the empty sequence ϵ. Figure 7 illustrates this iteration process pictorially. The iteration continues until a given bound k is reached, at which point $\bigcup_{i=1}^{k} \Delta_i$ is returned as the result of the routine.

The main work of CONTEST is done mostly by the operations symex and prune. It invokes $\mathsf{symex}(s_{in}, \gamma_{in}, \Pi_{i-1})$ to generate all the feasible one-step extensions of traces in Π_{i-1}. Hence,

$$\begin{array}{l}\mathsf{symex}(s_{in}, \gamma_{in}, \Pi_{i-1}) = \\ \quad \{\tau\langle C, W\rangle \mid \tau \in \Pi_{i-1} \text{ and } \tau\langle C, W \rangle \text{ is feasible}\},\end{array}$$

where $\tau\langle C, W\rangle$ means the concatenation of τ with a single-step trace $\langle C, W\rangle$. The symex operation can be easily implemented following a standard algorithm for concolic execution (modulo the well-known issue with loops), as we did in our implementation for Android.[4] In fact, if we skip the pruning step in CONTEST and set Π_i to Σ_i there (equivalently, $\mathsf{prune}(\Delta)$ returns simply Δ), we get the standard concolic execution algorithm, CLASSIC for exploring all branches that are reachable by event sequences of length k or less.

The goal of the other operation prune is to identify traces that can be thrown away without making the algorithm cover less branches, and to filter out such traces. This filtering is the main optimization employed in our algorithm. It is based on our novel idea of subsumption between traces, which we discuss in detail in the next subsection.

EXAMPLE 2. *We illustrate our algorithm with the music player app in Example 1 and the bound $k = 2$. Initially, the algorithm sets $\Pi_0 = \Delta_0 = \{\epsilon\}$. Then, it extends this empty sequence by calling* symex*, and obtains Δ_1 that contains the following three traces of length 1:*

$$\begin{array}{ll}\tau &= \langle \mathsf{True}^{\langle l_0, true\rangle}(a{=}{=}\mathsf{Play})^{\langle l_1, true\rangle},\ \{g\}\rangle, \\ \tau' &= \langle \mathsf{True}^{\langle l_0, true\rangle}(a{=}{=}\mathsf{Play})^{\langle l_1, false\rangle}(a{=}{=}\mathsf{Skip})^{\langle l_2, true\rangle},\ \{g\}\rangle, \\ \tau'' &= \langle \mathsf{True}^{\langle l_0, true\rangle}(a{=}{=}\mathsf{Play})^{\langle l_1, false\rangle}(a{=}{=}\mathsf{Skip})^{\langle l_2, false\rangle},\ \emptyset\rangle.\end{array}$$

[4] When s_{in} contains loops, the standard concolic execution can fail to terminate. However, symex is well-defined for such programs, because it is not an algorithm but a declarative specification.

$$\{\epsilon\} = \Pi_0 \xrightarrow{\text{symex}} \Delta_1 \underset{\supseteq}{\xrightarrow{\text{prune}}} \Pi_1 \xrightarrow{\text{symex}} \Delta_2 \underset{\supseteq}{\xrightarrow{\text{prune}}} \Pi_2 \xrightarrow{\text{symex}} \ldots$$

Figure 7: Simulation of our CONTEST algorithm.

Trace τ describes the execution that takes the true branches of l_0 and l_1. It also records that variable g is updated in this execution. Traces τ' and τ'' similarly correspond to executions that take different paths through the program.

Next, the algorithm prunes redundant traces from Δ_1. It decides that τ'' is such a trace, filters τ'', and sets $\Pi_1 = \{\tau, \tau'\}$. This filtering decision is based on the fact that the last step of τ'' does not modify any global variables. For now, we advise the reader not to worry about the justification of this filtering; it will be discussed in the following subsections.

Once Δ_1 and Π_1 are computed, the algorithm goes to the next iteration, and computes Δ_2 and Π_2 similarly. The trace set Δ_2 is obtained by calling symex*, which extends traces in Π_1 with one further step:*

$$\begin{aligned}\Delta_2 = \{\ & \tau\langle \mathsf{True}^{\langle l_0, false\rangle} (a{=}{=}\mathsf{Stop})^{\langle l_3, true\rangle}, \{g\}\rangle, \\ & \tau\langle \mathsf{True}^{\langle l_0, false\rangle} (a{=}{=}\mathsf{Stop})^{\langle l_3, false\rangle}, \emptyset\rangle, \\ & \tau'\langle \mathsf{True}^{\langle l_0, false\rangle} (a{=}{=}\mathsf{Stop})^{\langle l_3, true\rangle}, \{g\}\rangle, \\ & \tau'\langle \mathsf{True}^{\langle l_0, false\rangle} (a{=}{=}\mathsf{Stop})^{\langle l_3, false\rangle}, \emptyset\rangle\ \}\end{aligned}$$

Among these traces, only the first and the third have the last step with the nonempty write set, so they survive pruning and form the set Π_2.

After these two iterations, our algorithm returns $\bigcup_{i=0}^{2} \Delta_i$.

4.4 Subsumption

For a feasible trace τ, we define

$$\mathsf{final}(\tau) = \{\gamma' \mid \exists \pi.\, \langle s_{in}, \pi, \gamma_{in}\rangle \Downarrow \gamma' \triangleright \tau\},$$

which consists of the final states of the executions of s_{in} that generate the trace τ.

Let τ and τ' be feasible traces. The trace τ ***is subsumed by*** τ', denoted $\tau \sqsubseteq \tau'$, if and only if $\mathsf{final}(\tau) \subseteq \mathsf{final}(\tau')$. Note that the subsumption compares two traces purely based on their final states, ignoring other information like length or accessed global variables. Hence, the subsumption is appropriate for comparing traces when the traces are used to represent sets of global states, as in our algorithm CONTEST. We lift subsumption on sets T, T' of feasible traces in a standard way: $T \sqsubseteq T' \iff \forall \tau \in T.\, \exists \tau' \in T'.\, \tau \sqsubseteq \tau'$. Both the original and the lifted subsumption relations are preorder, i.e., they are reflexive and transitive.

A typical use of subsumption is to replace a trace set T_{new} by a subset T_{opt} such that $T_{new} \sqsubseteq T_{opt} \cup T_{old}$ for some T_{old}. In this usage scenario, T_{new} represents a set of traces that a concolic testing algorithm originally intends to extend, and T_{old} that of traces that the algorithm has already extended. Reducing T_{new} to T_{opt} entails that fewer traces will be explored, so it boosts the performance of the algorithm.

Why is it ok to reduce T_{new} to T_{opt}? An answer to this question lies in two important properties of the subsumption relation. First, the symex operation preserves the subsumption relationship.

LEMMA 1. *For sets T, T' of feasible traces,*

$$T \sqsubseteq T' \implies \mathsf{symex}(s_{in}, \gamma_{in}, T) \sqsubseteq \mathsf{symex}(s_{in}, \gamma_{in}, T').$$

Algorithm 2 The rprune operation

INPUTS: Set Δ of traces.
OUTPUTS: Set $\Pi = \{\tau \mid \tau \in \Delta \wedge |\tau| \geq 1 \wedge (\tau_{|\tau|}).W = \emptyset\}$

Second, if T is subsumed by T', running symex with T' will cover as many branches as what doing the same thing with T covers. Let

$$\mathsf{branch}(C) = \{\langle l, v\rangle \mid b^{\langle l,v\rangle} = C_i \text{ for some } i \in \{1, \ldots, |C|\}\}.$$

The formal statement of this second property appears in the following lemma:

LEMMA 2. *For all sets T, T' of feasible traces, if $T \sqsubseteq T'$.*

$$\begin{aligned}&\bigcup\{\mathsf{branch}((\tau_{|\tau|}).C) \mid \tau \in \mathsf{symex}(s_{in}, \gamma_{in}, T)\} \\ &\quad \subseteq \bigcup\{\mathsf{branch}((\tau_{|\tau|}).C) \mid \tau \in \mathsf{symex}(s_{in}, \gamma_{in}, T')\}.\end{aligned}$$

In the lemma, $\tau_{|\tau|}$ means the last element in the trace τ and $(\tau_{|\tau|}).C$ chooses the C component of this element. So, the condition compares the branches covered by the last elements of traces.

Using these two lemmas, we can now answer our original question about the subsumption-based optimization. Suppose that T_{opt} is a subset of T_{new} but $T_{new} \sqsubseteq T_{opt} \cup T_{old}$ for some T_{old}. The lemmas imply that every new branch covered by extending T_{new} for the further $k \geq 1$ steps is also covered by doing the same thing for $T_{opt} \cup T_{old}$. More concretely, according to Lemma 1, the extension of T_{new} for the further $k-1$ or smaller steps will continue to be $\sqsubseteq$-related to that of $T_{opt} \cup T_{old}$. Hence, running symex with such extended T_{new} will cover only those branches that can also be covered by doing the same thing for the similarly extended $T_{opt} \cup T_{old}$ (Lemma 2). Since we assume that the k or smaller extensions of T_{old} are already explored, this consequence of the lemmas mean that as long as we care about only newly covered branches, we can safely replace T_{new} by T_{opt}, even when T_{opt} is a subset of T_{new}.

4.5 Pruning

The goal of the pruning operator is to reduce a set Δ of feasible traces to a subset $\Pi \subseteq \Delta$, such that Δ is subsumed by Π and all strict prefixes of Δ:[5]

$$\Delta \ \sqsubseteq\ \Pi \cup \mathsf{sprefix}(\Delta), \qquad (1)$$

where $\mathsf{sprefix}(\Delta) = \{\tau \mid \exists \tau'.\, |\tau'| \geq 1 \wedge \tau\tau' \in \Delta\}$. The reduction brings the gain in performance, while the subsumption relationship (together with an invariant maintained by CONTEST) ensures that no branches would be missed by this optimization.

Our implementation of pruning adopts a simple strategy for achieving the goal. From a given set Δ, the operator filters out all traces whose last step does not involve any writes, and returns the set Π of remaining traces. The implementation appears in Figure 2, and accomplishes our goal, as stated in the following lemma:

LEMMA 3. *For all sets Δ of feasible traces,* rprune(Δ) *is a subset of Δ and satisfies the condition in* (1).

We point out that the pruning operator can be implemented differently from rprune. As long as the pruned set Π

[5] This condition typechecks because all prefixes of feasible traces are again feasible traces so that the RHS of $\sqsubseteq$ contains only feasible traces.

satisfies the subsumption condition in (1), our entire algorithm CONTEST remains relatively complete, meaning that the optimization with pruning will not introduce new uncovered branches. In Appendix C, we include another implementation of pruning that uses the notion of independence.

4.6 Relative Completeness

For $i \geq 0$, let $\mathsf{symex}^i(s_{in}, \gamma_{in}, T)$ be the i-repeated application of $\mathsf{symex}(s_{in}, \gamma_{in}, -)$ to a set T of feasible traces, where the 0-repeated application $\mathsf{symex}^0(s_{in}, \gamma_{in}, T)$ is defined to be T. Also, lift the branch operation to a trace set:

$$\mathsf{branch}(T) = \bigcup\{\mathsf{branch}((\tau_i).C) \mid \tau \in T \wedge i \in \{1, \ldots, |\tau|\}\}$$

THEOREM 4 (COMPLETENESS). *For every $k \geq 1$,*

$$\mathsf{branch}(\text{CONTEST}(s_{in}, \gamma_{in}, k)) = \mathsf{branch}(\textstyle\bigcup_{i=0}^{k} \mathsf{symex}^i(s_{in}, \gamma_{in}, \{\epsilon\})).$$

The RHS of the equation in the theorem represents branches covered by running the standard concolic execution without pruning. The theorem says that our algorithm covers the same set of branches, hence same program statements, as the standard concolic execution.

5. EMPIRICAL EVALUATION

In this section, we present the empirical evaluation of our technique. First, we describe the implementation of CONTEST (Section 5.1). Next, we present the studies, including the subjects used, the empirical setup, and the study results (Sections 5.2–5.4). Finally, we discuss threats to the validity of the studies (Section 5.5).

5.1 Implementation

The implementation of our system uses the Soot framework [31], and consists of 11,000 lines of Java code. Figure 8 shows a dataflow diagram of our system. Our system inputs *Android SDK*, the Android's framework classes, and the Java class files of the *App under test*. Our system outputs a set of tests, *Test inputs*, each of which denotes an event sequence. The script shown in the inlined box is an example of a test that our system can generate: `Tap` generates a tap event on the screen at the specified `X` and `Y` coordinates; `UserWait` simulates a user waiting for the specified time for the app to respond to the preceding event. Tests similar to the one in this script can be automatically executed using Monkey—a tool in the Android software development kit.

```
Tap(248.0,351.0)
UserWait(4000)
Tap(279.0,493.0)
UserWait(4000)
```

Our system consists of four components: Instrumenter, Runner, Concolic testing engine, and Subsumption analyzer. We explain each of them in turn.

Instrumenter inputs *Android SDK*, and the Java class files of the *App under test*, and outputs *Instrumented (SDK+App)*. This component instruments the Java bytecodes of each class of the *App under test* and any third-party libraries that the *App* uses. It also instruments classes in the Android framework (e.g., `android.*`) but this step is performed only once because the way in which a class is instrumented does not depend on any other class.

Instrumenter operates on a three-address form of Java bytecode produced by Soot, called Jimple. Instrumenter performs three types of instrumentations. First, it instruments *App* for concolic execution, which involves two main steps: (1) adds a meta variable (field) that stores the symbolic value corresponding to each variable (field); (2) inserts a new assignment before every assignment such that the new assignment copies the content of meta variable (field) corresponding to the r-value of the original assignment to the meta variable (field) corresponding to l-value of the original assignment. Second, Instrumenter instruments *App* to record fields of Java classes that are written only *during* the app responds to the last event in the sequence of events corresponding to a test. Third, Instrumenter ensures that in *Instrumented (SDK+App)*, user-specified method summaries are symbolically executed instead of the original methods.

Runner inputs *Instrumented (SDK+App)*. The first time the component is called, it generates a test randomly; thereafter, it inputs tests from either the Concolic testing engine or the Subsumption analyzer. Runner outputs *Test inputs* that includes the randomly-generated test and tests that it inputs. For each of those tests in *Test inputs*, it also outputs a *Path constraint* and a *Write set*, which are used internally by the other two components.

Runner executes *Instrumented (App)* with the test on an emulator that uses *Instrumented (SDK)*. Besides these Android framework classes, no other components of the framework, such as Dalvik virtual machine of the Android execution environment, are modified. This feature of our system makes it easily portable to different versions of Android.

Execution of a test generates the path constraint of the path that the *App* takes and *Write set*, which is a set of fields of Java classes that are written during the last event in the input event sequence. Writes to array elements are recorded as writes to one distinguished field. Runner uses a set of (typically 16) emulators each of which can execute a different test at any time. Such parallelism enables our system to perform systematic testing of realistic apps. Execution of an app in a realistic environment, such as an emulator or an actual device, takes orders of magnitude more time than execution of similar desktop applications.

Concolic testing engine inputs *Path constraint* of a path, and outputs *New tests for current iteration*. The component first computes a set of new path constraints by systematically negating each atomic constraint (i.e., conjunct) of the input path constraint, as in standard concolic testing. Then, it checks satisfiability of each of those new path constraints, and generates and outputs new tests corresponding to satisfiable path constraints using the Z3 SMT solver [7].

Subsumption analyzer inputs *Write set*, a set of fields of Java classes that are written when *App* responds to the last event in the event sequence corresponding to a specific test. It may output one *Seed test for next iteration*.

Subsumption analyzer implements the rprune operator in Algorithm 2. It outputs the test that corresponds to its input *Write set* if *Write set* is non-empty. The output test is called the seed test because new tests are generated in the next iteration by extending this test with new events. If *Write set* is empty, Subsumption analyzer outputs no test.

One important feature of Subsumption analyzer is that it can be configured to ignore writes to a given set of fields. This feature is useful because, in Android, many events lead to writes to some memory locations, which fall into two classes: (1) locations that are written to and read from dur-

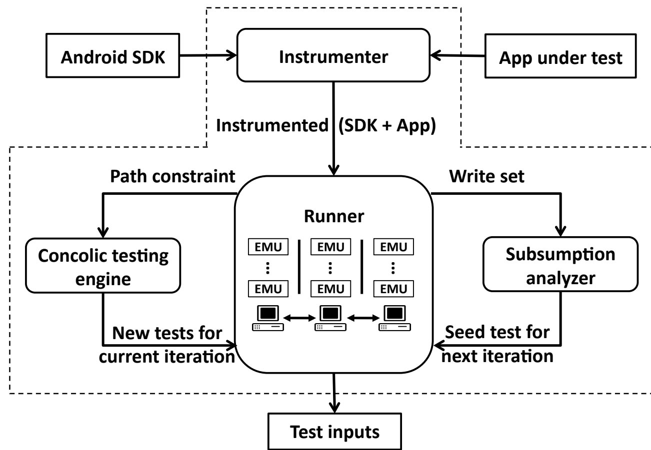


Figure 8: Dataflow diagram of our system.

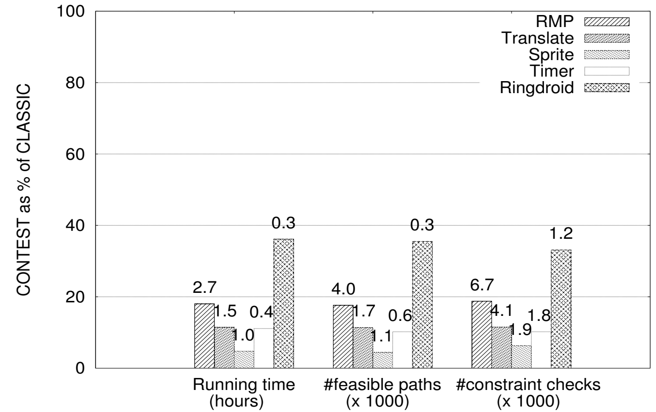


Figure 9: Results of Study 1: Running time, number of feasible paths explored, and number of constraint checks made by CONTEST normalized with respect to those metrics for CLASSIC.

ing the same event of an event sequence (i.e., never written and read across events); (2) locations that result from Android's low-level operations, such as optimizing performance and memory allocation, and correspond to fields of Android classes that are irrelevant to an app's behavior. Subsumption analyzer ignores writes to these two classes of writes because they are irrelevant to an app's behavior in subsequent events of an event sequence.

5.2 Subject Apps

We used five open-source Android apps for our studies. Random Music Player (RMP) is the app that is used as the example in Section 2. Sprite is an app for comparing the relative speeds of various 2D drawing methods on Android. Translate is an app for translating text from one language to another using Google's Translation service on the Web. Timer is an app for providing a countdown timer that plays an alarm when it reaches zero. Ringdroid is an app for recording and editing ring tones.

5.3 Study 1

The goal of this study is to measure the improvement in efficiency of CONTEST over CLASSIC. First, we performed concolic execution for each subject using CONTEST and CLASSIC. We used k=4 for RMP, Translate, and Sprite. However, because CLASSIC did not terminate for the other two apps when k=4 in the 12-hour time limit set for experiments, we used k=3 for Timer and k=2 for Ringdroid. Note that CONTEST terminated for all five apps even for k=4. In this step, we used 16 concurrently running emulators to execute tests and compute path constraints for corresponding program paths. Second, for each algorithm, we computed three metrics:

1. The running time of the algorithm.
2. The number of feasible paths that the algorithm finds.
3. The number of satisfiability checks of path constraints that the algorithm makes.

We measure the running time of the algorithms (metric (1)) because comparing them lets us determine the efficiency of CONTEST over CLASSIC. However, by considering only running time, it may be difficult to determine whether the efficiency of our algorithm will generalize to other application domains and experimental setups. Furthermore, we need to verify that the savings in running time is due to the reduction provided by our algorithm. Thus, we also compute the other two metrics (metrics (2) and (3)).

Figure 9 shows the results of the study for our subjects. In the figure, the horizontal axis represents the three metrics, where each cluster of bars corresponds to one metric. Within each cluster, the five bars represent the corresponding metric for the five subjects. In the first cluster, the height of each bar represents the normalized ratio (expressed as a percentage) of the running time of CONTEST to that of CLASSIC. The number at the top of each bar in this cluster is the running time of CONTEST measured in hours. Similarly, in the second and third clusters, the height of each bar represents the normalized ratio of the number of feasible paths explored and the number of constraint checks made, respectively, by CONTEST to the corresponding entities for CLASSIC. The number at the top of each bar in the second and third clusters is the number of feasible paths explored and the number of constraint checks made, respectively, by CONTEST. For brevity, these numbers are rounded, and shown as multiples of a thousand. For example, the first cluster shows the ratio of the running time of CONTEST to that of CLASSIC: RMP is 18%; Translate is 15%; Sprite is 5%; Timer is 11%; Ringdroid is 36%. This cluster also shows that the running time of CONTEST is 2.7 hours for RMP, 1.5 hours for Translate, 1 hour for Sprite, 0.4 hours for Timer, and 0.3 hours for Ringdroid.

The results of the study show that CONTEST is significantly more efficient than CLASSIC. CONTEST requires only a small fraction (5%–36%) of the running time of CLASSIC to achieve the same completeness guarantee. Thus, using CONTEST provides significant savings in running time over CLASSIC (64%–95%). The results also illustrate why the running time for CONTEST is significantly less than for CLASSIC: CONTEST explores only 4.4%–35.5% of all feasible paths that CLASSIC explores; CONTEST checks significantly fewer constraints (6.2%–33.1%) than CLASSIC.

5.4 Study 2

The goal of this study is to record the number of paths pruned by CONTEST because this reduction in the number of paths explored highlights why CONTEST is more efficient than CLASSIC. To do this, we performed concolic execution

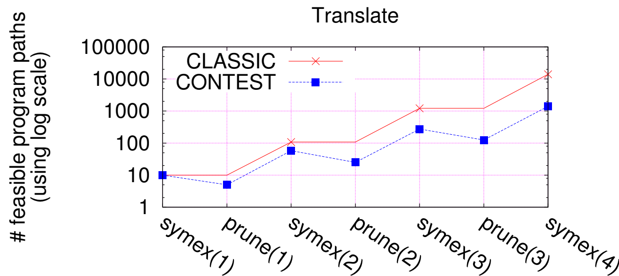


Figure 10: Results of Study 2 for *translate* app: The number of paths (using a logarithmic scale) after symex and prune operations in each iteration.

of each app for k=4, and we recorded the following information for each iteration of Contest and Classic:

1. The number of feasible paths that symex explores; recall that symex explores new feasible paths.
2. The number of feasible paths that remain after prune.

Figure 10 shows the results of the study for one subject app; the results for the remaining apps are similar, and are shown in Appendix D. In each graph, the horizontal axis represents the symex and prune operations performed in each iteration. The vertical axis shows the number of paths using a log scale. For example, the graph for Translate in Figure 10 shows that Contest explores 274 paths in iteration 3. The subsequent pruning step filters out 149 paths. Thus, only the remaining 125 paths are extended in iteration 4. In contrast, Classic explores 1,216 paths in iteration 3, all of which are extended in iteration 4.

The results clearly show the improvement achieved by the pruning that Contest performs. First, the graphs show that Contest explores many fewer paths than Classic, and the rate of improvement increases as the number of iterations increases. For example, in the fourth iteration of symex, Contest explores 1,402 paths and Classic has explored 13,976 paths. Second, the graphs also show that, at each iteration of prune, the number of paths that will then be extended decreases: the descending line in the graphs represents the savings that prune produces. In contrast, the horizontal line for the same interval corresponding to Classic shows that no pruning is being performed.

5.5 Threats to Validity

There are several threats to the validity of our studies. The main threat to internal validity arises because our system is configured to ignore writes to certain fields that do not affect an app's behavior (see Section 5.1 under "Subsumption analyzer"). We mitigate this threat in two ways. First, our implementation ignores only fields of Android's internal classes that are clearly irrelevant to an app's behavior; it never ignores fields of app classes, third-party libraries, or fields of Android classes (e.g., widgets) that store values that can be read by an app. Second, we ran our system using the Classic algorithm (that performs no pruning), and checked if any ignored field is written in one event and read in a later event of an event sequence. Most of the fields that our system is configured to ignore are never read and written across events. For the few that were, we manually confirmed that it is safe to ignore them.

Threats to external validity arise when the results of the experiment cannot be generalized. We evaluated our technique with only five apps. Thus, the efficiency of our technique may vary for other apps. However, our apps are representative of typical Android apps considering the problem that our technique addresses.

6. RELATED WORK

Our work is related to work on GUI testing and on alleviating path explosion in concolic testing.

Menon [21] presented the first framework for generating, running, and evaluating GUI tests. Several papers (e.g., [4, 23, 24, 34]) present components and extensions of this framework. Most existing GUI testing approaches either use capture-replay to infer the GUI model automatically [22, 30] or require users to provide the GUI model [33, 35]. An exception is the work of Ganov et al. [10], which uses symbolic execution to infer data inputs to GUI components. Our work also uses symbolic execution but focuses on event inputs. Our techniques for efficiently generating sequences of events are complementary to the above approaches.

Significant advances have been made in recent years to alleviate path explosion in concolic testing. These include compositional testing [1, 11], using program dependence information to avoid analyzing redundant paths [3, 20, 26, 27], analyzing loops in specialized ways [16, 28], using input grammars [13, 19], using manual models of library classes [17] or constraint solvers that support higher-level program abstractions [2, 32], and using path-exploration heuristics that cover deep internal parts of a program [18, 25]. Our input subsumption idea is complementary to the above ideas for taming path explosion. Our system indeed leverages some of the above ideas. It uses (1) method summaries and models for certain Android framework classes, (2) a grammar to specify input events, and (3) state-of-the-art constraint solving provided by the Z3 SMT solver.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented a technique, Contest to automatically and systematically generate input events to exercise smartphone apps. We described our system that implements Contest for Android, and presented the results of our empirical evaluation of the system on five real apps. The results showed that for our subjects, Contest is significantly more efficient than the naive concolic execution technique, referred to as Classic.

We have at least three important directions for future work. First, Contest only alleviates path explosion. The improved efficiency of Contest over Classic may not be sufficient to handle apps that have significantly more paths than our subjects. An example of such an app is one that has many widgets (e.g., a virtual keyboard). We plan to study other subsumption patterns besides the read-only pattern that we currently exploit to tame path explosion. The independence pattern, described in Appendix C, is an example. Second, our system currently handles only one type of events (i.e., tap events). There are many other types of events such as incoming phone calls and gestures. Extending our system to handle other types of events will widen its applicability to more apps. Third, we intend to conduct a more exhaustive empirical evaluation with more subjects to further confirm Contest's improvement over Classic.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS'08*.

[2] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS'09*.

[3] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*.

[4] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a single model and test prioritization strategies for event-driven software. *IEEE Trans. Software Eng.*, 37(1):48–64, 2011.

[5] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *EuroSys'11*, pages 301–314, 2011.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*.

[9] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS'11*.

[10] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Event listener analysis and symbolic execution for testing gui applications. In *ICFEM'09*.

[11] P. Godefroid. Compositional dynamic test generation. In *POPL'07*.

[12] P. Godefroid. Higher-order test generation. In *PLDI'11*.

[13] P. Godefroid, A. Kiezun, and M. Levin. Grammar-based whitebox fuzzing. In *PLDI'08*.

[14] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI'05*.

[15] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS'08*.

[16] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA'11*.

[17] S. Khurshid and Y. Suen. Generalizing symbolic execution to library classes. In *PASTE'05*.

[18] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE'07*.

[19] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ASE'07*.

[20] R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV'09*.

[21] A. M. Memon. *A comprehensive framework for testing graphical user interfaces.* Ph.D., 2001.

[22] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE'03*.

[23] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for guis. In *FSE'00*.

[24] A. M. Memon and M. L. Soffa. Regression testing of guis. In *ESEC/FSE'03*.

[25] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA'08*.

[26] D. Qi, H. D. T. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. In *FSE'11*.

[27] R. A. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *ISSTA 2010*, pages 195–206, July 2010.

[28] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA'09*.

[29] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*.

[30] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *ICST'11*.

[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON'99*.

[32] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *ICST'10*.

[33] L. J. White and H. Almezen. Generating test cases for gui responsibilities using complete interaction sequences. In *ISSRE'00*.

[34] X. Yuan, M. B. Cohen, and A. M. Memon. Gui interaction testing: Incorporating event context. *IEEE Trans. Software Eng.*, 37(4):559–574, 2011.

[35] X. Yuan and A. M. Memon. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Trans. Software Eng.*, 36(1), 2010.

APPENDIX

This appendix consists of the four parts. In Appendix A, we explain the rules for deriving the evaluation relations in our operational semantics. In Appendix B, we prove the lemmas and the theorem presented in the main text of the paper. In Appendix C, we present another implementation of pruning that uses the notion of independence. Finally, in Appendix D, we present the rest of the graphs for the results in Study 2.

A. RULES FOR EVALUATION RELATIONS

Rules for the evaluation relations are given in Figure 11. Most of the rules in the figure follow from our intended reading of all the parts in the evaluation relations. For example, the first rule for the conditional statement says that if the boolean condition b evaluates to *true*, we extend the C component of a symbolic state with the result b' of symbolically evaluating the condition b, and follow the true branch.

The only unusual rule is the second one in (3) for evaluating input event sequences. This rule describes how to thread the iterative execution of a program. One unusual aspect is that the symbolic global state and the path constraint are reset for each input event. This ensures that the path constraint of a final concolic state restricts only the current input event, not any previous ones, in the input sequence.

B. PROOFS OF LEMMAS AND THEOREM

In this part of the appendix, we provide proofs of Lemmas 1 and 2 stated in Section 4.4 and Lemma 3 and Theorem 4 stated in Section 4.5 in the main text of the paper.

B.1 Proof of Lemma 1

LEMMA 1. *For sets T, T' of feasible traces,*

$$T \sqsubseteq T' \implies \mathsf{symex}(s_{in}, \gamma_{in}, T) \sqsubseteq \mathsf{symex}(s_{in}, \gamma_{in}, T').$$

PROOF. Pick τ from $\mathsf{symex}(s_{in}, \gamma_{in}, T)$. We show show that some τ' in $\mathsf{symex}(s_{in}, \gamma_{in}, T')$ satisfies $\tau \sqsubseteq \tau'$. By the definition of $\mathsf{symex}(s_{in}, \gamma_{in}, T)$, there exist a feasible trace α, a path constraint C, and a set W of global variables such that

$$\alpha \in T \ \wedge \ \tau = \alpha\langle C, W\rangle.$$

By assumption that $T \sqsubseteq T'$, the first conjunct above implies the existence of $\alpha' \in T'$ satisfying $\alpha \sqsubseteq \alpha'$.

We now claim that $\alpha'\langle C, W\rangle$ is the desired trace τ'. To show this claim, it is sufficient to prove that $\mathsf{final}(\alpha\langle C, W\rangle)$ is a subset of $\mathsf{final}(\alpha'\langle C, W\rangle)$. The feasibility of $\alpha'\langle C, W\rangle$ follows from this. Pick γ_1 from $\mathsf{final}(\alpha\langle C, W\rangle)$. Then, there exist $\gamma_0 \in \mathsf{final}(\alpha)$, n, and Γ such that

$$\langle s_{in}, n, \langle \gamma_0, \gamma_0, \epsilon\rangle\rangle \ \downarrow \ \langle \gamma_1, \Gamma, C\rangle \ \triangleright \ W. \quad (4)$$

Since $\mathsf{final}(\alpha) \subseteq \mathsf{final}(\alpha')$, the global state γ_0 must be in $\mathsf{final}(\alpha')$, meaning that for some π',

$$\langle s_{in}, \pi', \gamma_{in}\rangle \ \Downarrow \ \gamma_0 \ \triangleright \ \alpha'. \quad (5)$$

From (5) and (4) follows that

$$\langle s_{in}, \pi' n, \gamma_{in}\rangle \ \Downarrow \ \gamma_0 \ \triangleright \ \alpha'\langle C, W\rangle.$$

Hence, γ_1 is in $\mathsf{final}(\alpha'\langle C, W\rangle)$, as required. $\square$

B.2 Proof of Lemma 2

LEMMA 2. *For all sets T, T' of feasible traces, if $T \sqsubseteq T'$, we have that*

$$\bigcup\{\mathsf{branch}((\tau_{|\tau|}).C) \mid \tau \in \mathsf{symex}(s_{in}, \gamma_{in}, T)\} \subseteq \bigcup\{\mathsf{branch}((\tau_{|\tau|}).C) \mid \tau \in \mathsf{symex}(s_{in}, \gamma_{in}, T')\}.$$

PROOF. We will show that for all $\tau \in \mathsf{symex}(s_{in}, \gamma_{in}, T)$, there exists $\tau' \in \mathsf{symex}(s_{in}, \gamma_{in}, T')$ satisfying

$$\mathsf{branch}((\tau_{|\tau|}).C) \ \subseteq \ \mathsf{branch}((\tau'_{|\tau'|}).C).$$

Pick τ from $\mathsf{symex}(s_{in}, \gamma_{in}, T)$. Then, τ is feasible and has length at least 1. Also, there exist a feasible trace α, a path constraint C, and a set of global variables W such that

$$\tau = \alpha\langle C, W\rangle \ \wedge \ \alpha \in T.$$

Since $T \sqsubseteq T'$, there should be $\alpha' \in T'$ with

$$\alpha \sqsubseteq \alpha'.$$

Let $\tau' = \alpha'\langle C, R, W\rangle$. It is sufficient to prove that τ' is feasible. Since τ is feasible and it is $\alpha\langle C, W\rangle$, there exist n, γ_0, γ_1, and Γ_1 such that

$$\gamma_0 \in \mathsf{final}(\alpha) \ \wedge \ \langle s_{in}, n, \langle \gamma_0, \gamma_0, \epsilon\rangle\rangle \Downarrow \langle \gamma_1, \Gamma_1, C\rangle \triangleright W. \quad (6)$$

Since $\alpha \sqsubseteq \alpha'$, γ_0 is also in $\mathsf{final}(\alpha')$. This and the second conjunct of (6) imply that $\alpha'\langle C, W\rangle$ is feasible. $\square$

B.3 Proof of Lemma 3

LEMMA 3. *For all sets Δ of feasible traces,* $\mathsf{rprune}(\Delta)$ *is a subset of Δ and satisfies the condition in* (1).

PROOF. Let $\Pi = \mathsf{rprune}(\Delta)$. Because of the definition of rprune, Π has to be a subset of Δ. It remains to prove that the condition in (1) holds for Δ and Π. Pick τ in Δ. We should find τ' in $\Pi \cup \mathsf{sprefix}(\Delta)$ such that $\tau \sqsubseteq \tau''$. If τ is in Π, we can choose τ itself as τ''. The condition $\tau \sqsubseteq \tau''$ holds because of the reflexivity of $\sqsubseteq$. If τ is not in Π, we must have that $|\tau| \geq 1$ and $(\tau_{|\tau|}).W = \emptyset$. Let α be the prefix of τ that has length $|\tau| - 1$. Then, α is feasible, and it belongs to $\mathsf{sprefix}(\Delta)$. Furthermore, $\mathsf{final}(\tau) \subseteq \mathsf{final}(\alpha)$, since the additional last step of τ denotes read-only computations. Hence, $\tau \sqsubseteq \alpha$. From what we have just shown follows that α is the desired feasible trace. $\square$

B.4 Proof of Theorem 4

THEOREM 4. *For every $k \geq 1$,*

$$\mathsf{branch}(\text{CONTEST}(s_{in}, \gamma_{in}, k)) = \mathsf{branch}(\textstyle\bigcup_{i=0}^{k} \mathsf{symex}^i(s_{in}, \gamma_{in}, \{\epsilon\})).$$

PROOF. The LHS of the equation is a subset of the RHS, because

$$\text{CONTEST}(s_{in}, \gamma_{in}, k) \subseteq \bigcup_{i=0}^{k} \mathsf{symex}^i(s_{in}, \gamma_{in}, \{\epsilon\})$$

and the branch operator is monotone with respect to the subset relation. In the remainder of this proof, we show that the RHS is also a subset of the LHS.

Let F be a function on sets of traces given by $F(T) = \mathsf{symex}(s_{in}, \gamma_{in}, T)$. Define the operator $\mathsf{lbranch}$ on such trace sets by:

$$\mathsf{lbranch}(T) = \bigcup\{\mathsf{branch}((\tau_{|\tau|}).C) \mid \tau \in T \wedge |\tau| \geq 1\}.$$

$\boxed{\langle s,\ n,\ \omega\rangle \ \downarrow\ \omega' \ \triangleright\ W}$

$$\frac{}{\langle \mathsf{skip},\ n,\ \omega\rangle \ \downarrow\ \omega \ \triangleright\ \emptyset}$$

$$\frac{}{\langle g\!=\!e,\ n,\ \omega\rangle \ \downarrow\ \langle \omega.\gamma[g:n'],\ \omega.\Gamma[g:e'],\ \omega.C\rangle \ \triangleright\ \{g\}} \qquad [\text{where } [\![e]\!](n,\omega)=n' \text{ and } [\![e]\!]^s(\omega)=e]$$

$$\frac{\langle s_1,\ n,\ \omega[C:(\omega.C)b'^{\langle l,true\rangle}]\rangle \ \downarrow\ \omega' \ \triangleright\ W}{\langle \mathsf{if}\ b^l\ s_1\ \mathsf{else}\ s_2,\ n,\ \omega\rangle \ \downarrow\ \omega' \ \triangleright\ W} \qquad [\text{if } [\![b]\!](n,\omega)=\mathit{true} \text{ and } [\![b]\!]^s(\omega)=b']$$

$$\frac{\langle s_2,\ n,\ \omega[C:(\omega.C)(\neg b')^{\langle l,false\rangle}]\rangle \ \downarrow\ \omega' \ \triangleright\ W}{\langle \mathsf{if}\ b^l\ s_1\ \mathsf{else}\ s_2,\ n,\ \omega\rangle \ \downarrow\ \omega' \ \triangleright\ W} \qquad [\text{if } [\![b]\!](n,\omega)=\mathit{false} \text{ and } [\![b]\!]^s(\omega)=b']$$

$$\frac{\langle s_1,\ n,\ \omega\rangle \ \downarrow\ \omega' \ \triangleright\ W \qquad \langle s_2,\ n,\ \omega'\rangle \ \downarrow\ \omega'' \ \triangleright\ W'}{\langle s_1;s_2,\ n,\ \omega\rangle \ \downarrow\ \omega'' \ \triangleright\ W\cup W'}$$

$$\frac{\langle s,\ n,\ \omega[C:(w.C)b'^{\langle l,true\rangle}]\rangle \ \downarrow\ \omega' \ \triangleright\ W' \qquad \langle \mathsf{while}\ b^l\ s,\ n,\ \omega'\rangle \ \downarrow\ \omega'' \ \triangleright\ W''}{\langle \mathsf{while}\ b^l\ s,\ n,\ \omega\rangle \ \downarrow\ \omega'' \ \triangleright\ W'\cup W''} \qquad [\text{if } [\![b]\!](n,\omega)=\mathit{true} \text{ and } [\![b]\!]^s(\omega)=b']$$

$$\frac{}{\langle \mathsf{while}\ b^l\ s,\ n,\ \omega\rangle \ \downarrow\ \omega[C:(w.C)(\neg b')^{\langle l,false\rangle}] \ \triangleright\ \emptyset} \qquad [\text{if } [\![b]\!](n,\omega)=\mathit{false} \text{ and } [\![b]\!]^s(\omega)=b']$$

$\boxed{\langle s,\pi,\gamma\rangle \ \Downarrow\ \gamma' \ \triangleright\ \tau}$

$$\frac{}{\langle s,\epsilon,\gamma\rangle \ \Downarrow\ \gamma \ \triangleright\ \epsilon} \tag{2}$$

$$\frac{\langle s,n,\langle\gamma,\gamma,\epsilon\rangle\rangle \ \downarrow\ \omega \ \triangleright\ W \qquad \langle s,\pi,\omega.\gamma\rangle \ \Downarrow\ \gamma' \ \triangleright\ \tau'}{\langle s,n\pi,\gamma\rangle \ \Downarrow\ \gamma' \ \triangleright\ \langle\omega.C,W\rangle\tau'} \tag{3}$$

Figure 11: Concolic execution semantics.

Intuitively, this operator collects every branch covered by the last step of some trace in T.

We will use the following three facts that hold for all j in $\{0,\dots,k-1\}$:

$$\begin{aligned} \textstyle\bigcup_{i=0}^{j}\Delta_i \ &\sqsubseteq\ \textstyle\bigcup_{i=0}^{j}\Pi_i, \\ \textstyle\bigcup_{i=0}^{j}F^i(\{\epsilon\}) \ &\sqsubseteq\ \textstyle\bigcup_{i=0}^{j}\Pi_i, \\ \mathsf{branch}(\textstyle\bigcup_{i=0}^{j+1}F^i(\{\epsilon\})) \ &=\ \mathsf{lbranch}(\textstyle\bigcup_{i=1}^{j+1}F^i(\{\epsilon\})). \end{aligned}$$

Here Δ_i and Π_i are the trace sets that Contest computes. We prove all of these facts simutaneously by induction on j. The base cases are immediate from the definitions of F^i, Δ_i and Π_i.

The inductive case of the first fact is proved as follows:

$$\begin{aligned} \textstyle\bigcup_{i=0}^{j+1}\Delta_i \ &=\ (\textstyle\bigcup_{i=0}^{j}\Delta_i)\cup\Delta_{j+1} \\ &\sqsubseteq\ (\textstyle\bigcup_{i=0}^{j}\Delta_i)\cup\mathsf{sprefix}(\Delta_{j+1})\cup\Pi_{j+1} \\ &\subseteq\ (\textstyle\bigcup_{i=0}^{j}\Delta_i)\cup\Pi_{j+1} \\ &\sqsubseteq\ (\textstyle\bigcup_{i=0}^{j}\Pi_i)\cup\Pi_{j+1} \ =\ (\textstyle\bigcup_{i=0}^{j+1}\Pi_i). \end{aligned}$$

The $\mathsf{sprefix}(\Delta_{j+1})$ in the first line is the set of all strict prefixes of Δ_{j+1} (i.e., $\mathsf{sprefix}(\Delta_{j+1}) = \{\tau \mid \exists\tau'.\,|\tau'|\geq 1 \wedge \tau\tau' \in \Delta_{j+1}\}$). The derivation proves $(\bigcup_{i=0}^{j+1}\Pi_i) \sqsubseteq (\bigcup_{i=0}^{j+1}\Delta_i)$, because $T\subseteq T'$ implies $T\sqsubseteq T'$ and the subsumption $\sqsubseteq$ is reflexive and transitive. Also, the derivation uses only true steps, as it should. The second step holds because $\mathsf{prune}(\Delta_{j+1}) = \Pi_{j+1}$, the result of the prune operation satisfies the subsumption relationship in (1) (Section 4.4), and the union operator is monotone with respect to $\sqsubseteq$. The third step holds because $\mathsf{sprefix}(\Delta_{j+1}) \subseteq \bigcup_{i=0}^{j}\Delta_i$. The fourth step follows from the induction hypothesis.

For the inductive case of the second fact, we notice that

$$\begin{aligned} F^{j+1}(\{\epsilon\}) \ &\subseteq\ F(\textstyle\bigcup_{i=0}^{j}F^i(\{\epsilon\})) \ \sqsubseteq\ F(\textstyle\bigcup_{i=0}^{j}\Pi_i) \\ &=\ \textstyle\bigcup_{i=0}^{j}F(\Pi_i) \qquad\quad =\ \textstyle\bigcup_{i=1}^{j+1}\Delta_i \\ &\subseteq\ \textstyle\bigcup_{i=0}^{j+1}\Delta_i \qquad\qquad \sqsubseteq\ \textstyle\bigcup_{i=0}^{j+1}\Pi_i \end{aligned}$$

The first step uses the monotonicity of F with respect to the subset relation, and the second uses the induction hypothesis and the fact that F preserves subsumption (Lemma 1). The third holds because F preserves union. The fourth step follows the definition of Δ_i, and the last step from the inductive step of the first fact, which we have already proved. Since the relation $\sqsubseteq$ includes the subset relation and is reflexive and transitive, the above derivation shows that $F^{j+1}(\{\epsilon\}) \sqsubseteq \bigcup_{i=0}^{j+1}\Pi_i$. Combining this and the induction hypothesis, we get the desired

$$\bigcup_{i=0}^{j+1}F^i(\{\epsilon\}) \ =\ F^{j+1}(\{\epsilon\})\cup\bigcup_{i=0}^{j}F^i(\{\epsilon\}) \ \sqsubseteq\ \bigcup_{i=0}^{j+1}\Pi_i.$$

Here we use the fact that the union operator is monotone with respect to $\sqsubseteq$.

For the inductive case of the third fact, we observe that $\mathsf{branch}(F^{j+2}(\{\epsilon\}))$ is a subset of

$$\mathsf{branch}(F^{j+1}(\{\epsilon\}))\cup\mathsf{lbranch}(F^{j+2}(\{\epsilon\})).$$

This superset itself is included in

$$\mathsf{lbranch}(\bigcup_{i=1}^{j+1} F^i(\{\epsilon\})) \cup \mathsf{lbranch}(F^{j+2}(\{\epsilon\}))$$

because of the induction hypothesis. Using this observation and the induction hypothesis again, we complete the proof of this inductive case as follows:

$$\begin{aligned} &\mathsf{branch}(\bigcup_{i=0}^{j+2} F^i(\{\epsilon\})) \\ &= \mathsf{branch}(\bigcup_{i=0}^{j+1} F^i(\{\epsilon\})) \cup \mathsf{branch}(F^{j+2}(\{\epsilon\})) \\ &\subseteq \mathsf{lbranch}(\bigcup_{i=1}^{j+1} F^i(\{\epsilon\})) \cup \mathsf{lbranch}(F^{j+2}(\{\epsilon\})) \\ &= \mathsf{lbranch}(\bigcup_{i=1}^{j+2} F^i(\{\epsilon\})). \end{aligned}$$

The two equalities here use the fact that lbranch preserves the union operator.

Using the three facts just shown, we can complete the proof of this theorem as follows:

$$\begin{aligned} \mathsf{branch}(\bigcup_{i=0}^{k} F^i(\{\epsilon\})) &= \mathsf{lbranch}(\bigcup_{i=1}^{k} F^i(\{\epsilon\})) \\ &= (\mathsf{lbranch} \circ F)(\bigcup_{i=0}^{k-1} F^i(\{\epsilon\})) \\ &\subseteq (\mathsf{lbranch} \circ F)(\bigcup_{i=0}^{k-1} \Pi_i) \\ &= \mathsf{lbranch}(\bigcup_{i=0}^{k-1} F(\Pi_i)) \\ &= \mathsf{lbranch}(\bigcup_{i=1}^{k} \Delta_i) \\ &\subseteq \mathsf{lbranch}(\textsc{Contest}(s_{in}, \gamma_{in}, k)). \end{aligned}$$

The first step is by the third fact, and the second and fourth steps hold because F preserves the union operator. The third step follows from the second fact and Lemma 2. The last two steps are just the unrolling of the definitions of Δ_i and the result of $\textsc{Contest}(s_{in}, \gamma_{in}, k)$. □

C. INDEPENDENCE-BASED PRUNING

Algorithm 3 gives another implementation of prunning, called iprune, which exploits a form of independence. This implementation assumes that the evaluation relations track a set of read global variables, in addition to written ones. This means that the forms of evaluation relations are changed to

$$\langle s, n, \omega \rangle \downarrow \ \omega' \ \triangleright \ W, R \qquad \langle s, \pi, \gamma \rangle \Downarrow \ \gamma' \ \triangleright \ \tau,$$

where R is a set of read variables and τ is now a sequence of triples C, W, R. Also, the rules for these relations are changed appropriately. Lemmas 1 and 2 in Section 4.4 and Theorem 4 in Section 4.6 remain valid even with these changes.

The iprune operator detects two traces τ, τ' in Δ such that τ can be obtained by swapping independent consecutive parts in τ'. In Figure 3, $\alpha\beta\beta'$ corresponds to τ', and β and β' represent consecutive independent parts. Although the iprune operator is not implemented in our system, it illustrates the generality of using our subsumption condition. The following lemma shows that iprune satisfies the condition.

Lemma 5. *The result of* iprune *is a subset of its input trace set, and it always satisfis the subsumption relationship in* (1) *(Section 4.5).*

Proof. Consider a set Δ of feasible traces, and let $\Pi = \mathsf{iprune}(\Delta)$. From the definition of iprune, it is immediate that Π is a subset of Δ. To prove that Π also satisfies the condition in (1), pick τ from Δ. We will have to find τ' in Π such that $\tau \sqsubseteq \tau'$. If τ is aleady in Π, we can just use τ for τ'. Suppoes that τ is not in Π. Then, by the definition of our algorithm, there must be a feasible trace $(\alpha\beta\beta')$ in Π such that (1) $\tau = \alpha\beta'\beta$ and (2) for all i and j,

$$(\beta_i).R \cap (\beta'_j).W = (\beta_i).W \cap (\beta'_j).R = (\beta_i).W \cap (\beta'_j).W = \emptyset.$$

Since Π is a subset of Δ throughout the execution of iprune, we know that $\alpha\beta\beta'$ is a feasible trace. Furthermore, the two properties of this trace above imply that

$$\mathsf{final}(\tau) = \mathsf{final}(\alpha\beta\beta'),$$

so $\tau \sqsubseteq (\alpha\beta\beta')$. From what we have just proven so far follows that $\alpha\beta\beta'$ is the trace τ' that we are looking for. □

Algorithm 3 The iprune operation

INPUTS: Set Δ of traces.
OUTPUTS: Set Π of traces.
$\Pi = \emptyset$
for every $\tau \in \Delta$ **do**
  if there is some trace $(\alpha\beta\beta') \in \Pi$ such that
   (1) $\tau = \alpha\beta'\beta$ and
   (2) $\beta_i.R \cap \beta'_j.W = \beta_i.W \cap \beta'_j.R = \beta_i.W \cap \beta'_j.W = \emptyset$
    for all i and j
  then
   skip
  else
   $\Pi = \Pi \cup \{\tau\}$
  end if
end for
return Π

D. RESULTS OF STUDY 2

In Study 2 (Section 5.4), we presented the results for one of the subject apps. In this part of the Appendix, we present the results for the rest of the subject apps.

The graphs in Figure 12 provide these results. As the graphs show, the results are similar to those discussed in Section 5.4, and show the same reduction in feasible paths explored and constraints checked of Contest over Classic.

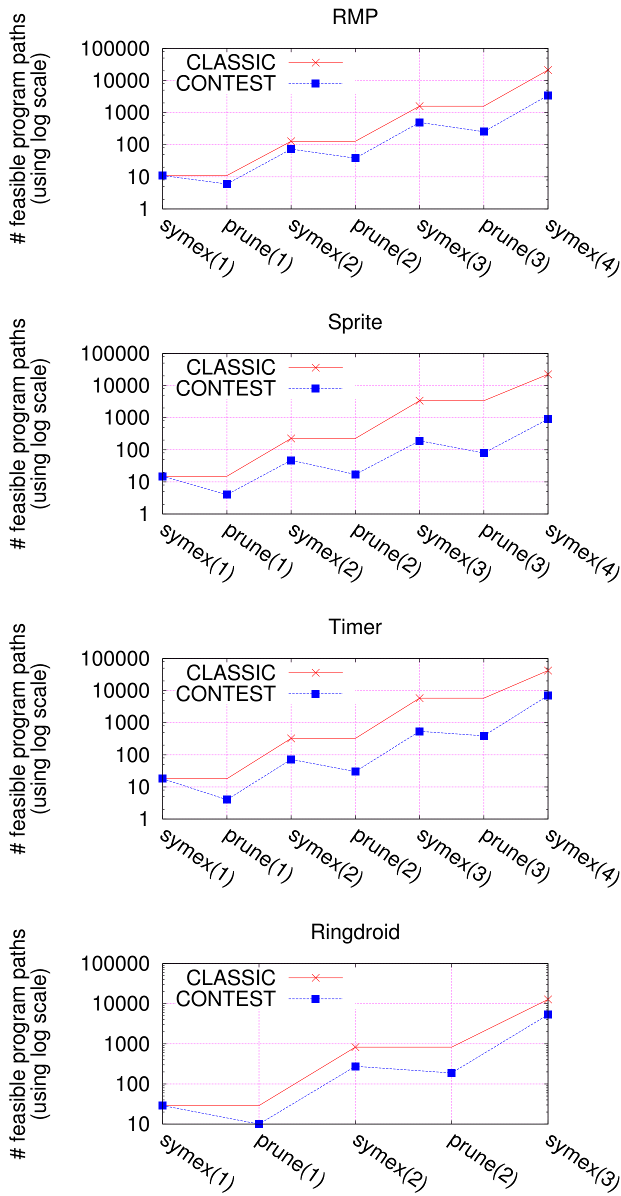


Figure 12: Results of Study 2: The number of paths (using a logarithmic scale) after symex and prune operations in each iteration. Because CLASSIC does not terminate for Timer and Ringdroid when k=4, the reported final numbers of paths for those two apps correspond to the time when the time-limit (12-hours) was met.