TECHNIQUES TO FACILITATE SYMBOLIC EXECUTION OF REAL-WORLD PROGRAMS

A Dissertation
Presented to
The Academic Faculty

by

Saswat Anand

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2012

TECHNIQUES TO FACILITATE SYMBOLIC EXECUTION OF REAL-WORLD PROGRAMS

Approved by:

Mary Jean Harrold, Committee Chair
School of Computer Science
Georgia Institute of Technology

Santosh Pande
School of Computer Science
Georgia Institute of Technology

Mayur Naik
School of Computer Science
Georgia Institute of Technology

Willem Visser
Deptartment of Mathematical Sciences
University of Stellenbosch, South Africa

Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Date Approved: 04 May 2012

To God

ACKNOWLEDGEMENTS

I thank the following individuals for their support, guidance, and company during my PhD.

Aliva Pattnaik, my wife and closest friend. You supported me wholeheartedly for better and worse, in sickness and health, in paper acceptance and rejection, and during internships (when I earned a real salary) and graduate assistantships. You have given me the freedom to choose any career path that will make me happy. I am grateful to you for giving me that freedom, but, more importantly, for enduring what ensues from how I use it.

Prof. Mary Jean Harrold, my advisor and mentor. You encouraged and supported me in my stubborn attempt to work on a difficult topic. You stood beside me when I was going through a string of difficulties in my personal life, and when, continuing with the PhD had lesser priority for me. I am greatly indebted for your support. Your sincerity and meticulosity, which reflects your passion for your job, and the respect that you enjoy in the research community, inspires me to become a faculty. I admire how you maintain personal relationships with not only your students, but also their family. Thank you so much for your affection towards my son, Rrishi. If I become a faculty, I will try to follow your footsteps in my interactions with students.

Prof. Willem Visser, my mentor. You introduced me to the topic of this dissertation, symbolic execution. Thanks so much for inviting me to visit NASA laboratory in the Summer of 2005, when I had only little credential.

Prof. Alex Orso, my mentor. I deeply appreciate your help in writing papers and preparing for presentations, and many stimulating research discussions.

Prof. Santosh Pande, my mentor. Thank you very much for the research insights and advices that I have received from you through my PhD.

Prof. Mayur Naik, my mentor and friend. You made the last year of my PhD very productive. The work I did with you is one of the most interesting works that I have done so far, and is one that will hopefully have significant impact.

George Baah, my office-mate and close friend. You, like me, were more focused on solving the problem with a meagre research assistantship, than graduating somehow and getting a real job. Without your company, I would have had lesser funny stories to tell my future colleagues and posterity about my PhD days. Let's now get outta here!

Hina Shah, my colleague and close friend. Thank you very much for giving mental and emotional support to both Aliva and me during times when the stress of PhD took the better of me.

Mr. Biswanath Pattanaik, my father-in-law. I greatly appreciate your understanding and support through all these years.

Mrs. Pragyan Patnaik and Mr. Nandalal Patnaik, my parents. You instilled in me the interest for study, which has led me to this point. I am indebted to you for life!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Symbolic execution is a program-analysis technique that is used to address several problems that arise in developing high-quality software. Despite the fact that the symbolic-execution technique is well understood, and performing symbolic execution on simple programs is straightforward, it is still not possible to apply the technique to the general class of large, real-world software. A symbolic-execution system can be effectively applied to large, real-world software if it has at least two features: efficiency and automation. However, efficient and automatic symbolic execution of real-world programs is a lofty goal due to both theoretical and practical reasons. Theoretically, achieving this goal requires solving an intractable problem (i.e., solving constraints). Practically, achieving this goal requires overwhelming effort to implement a symbolic-execution system that can precisely and automatically symbolically execute real-world programs.

This research makes three major contributions.

1. Three new techniques that address three important problems of symbolic execution. Compared to existing techniques, the new techniques

   - reduce the manual effort that may be required to symbolically execute those programs that either generate complex constraints or parts of which cannot be symbolically executed due to limitations of a symbolic-execution system.
   - improve the usefulness of symbolic execution (e.g., expose more bugs in a program) by enabling discovery of more feasible paths within a given time budget.

2. A novel approach that uses symbolic execution to generate test inputs for Apps that run on modern mobile devices such as smartphones and tablets.

3. Implementations of the above techniques and empirical results obtained from applying those techniques to real-world programs that demonstrate their effectiveness.

CHAPTER I

INTRODUCTION

Today, computers play indispensable roles in almost every human activity, from our daily mundane chores to great scientific and technological endeavors. In the future, our reliance on computers will grow even further. This reliance requires tools that facilitate rapid development of highly-dependable software. Most of these tools require reasoning about software behavior, and to do this, these tools use *program-analysis techniques* that analyze either the text or the runtime behavior of a program. For example, control-flow analyses are concerned with program paths (i.e., executable sequences of program statements). Data-flow analyses are concerned with how data, on which a program operates, flows between different program statements.

Symbolic execution [46] is a program-analysis technique that interprets a program using symbolic inputs along a path, and computes a constraint, called a path constraint, for that path over those symbolic inputs. If the path constraint is satisfiable, any solution of the constraint represents a program input that exercises the path. Chapter 2 presents an overview of the symbolic-execution technique.

1.1 Problems in Applying Symbolic Execution to Real-world Software

Although the symbolic-execution technique is well understood, and performing symbolic execution on simple programs is straightforward, it is still not possible to apply the technique to real-world software. A symbolic-execution system can be effectively applied to real-world software if it has at least two features: efficiency and automation. First, in most applications of symbolic execution (e.g., to improve code coverage or expose bugs), ideally, the goal is to discover all feasible program paths. For example, to expose bugs effectively, it is necessary to discover a large subset of all feasible paths, and show that either each of those paths is bug-free or many of them expose bugs. Thus, an *efficient* symbolic-execution system must be able discover as many distinct feasible program paths as possible in the available time

limit. Second, applying symbolic execution to any program should ideally not require any effort from the user. If manual effort is required, then the required manual effort should be acceptable in practice.

However, large, real-world programs have at least three characteristics that make it difficult to symbolically execute those programs efficiently and automatically. First, those programs typically have a large number of program paths. Second, those programs commonly use multiple programming languages, databases, file systems, networks, large frameworks, and advanced programming-language features. Moreover, parts of those programs may be available only in binary form. A symbolic-execution system for those programs, can be complex, and requires an overwhelming amount of effort to build. For example, in Java, native methods and features such as reflection that are dependent on the internals of a Java virtual machine (JVM) complicate the implementation of a symbolic-execution system. However, these features are widely-used in real-world Java programs. One study [7] shows that the number of times native methods are called by programs in the SPEC JVM98 benchmark ranges from 45 thousand to 5 million. Native methods are written in languages such as C, and thus, they do not have Java bytecode representations. Third, symbolic execution of these programs may produce path constraints that cannot be solved by state-of-the-art constraint solvers.

The aforementioned three characteristics of real-world programs pose three important problems that reduce the effectiveness of symbolic execution. Effectiveness is reduced because it may not be possible to generate test inputs for interesting program paths (e.g., those expose bugs) in a given time budget.

Path-explosion. It is difficult to symbolically execute a significantly large subset of all program paths because (1) most real-world software have an extremely large number of paths, and (2) symbolic execution of each program path can incur high computational overhead. Thus, in reasonable time, only a small subset of all paths can be symbolically executed. This problem needs to be addressed for efficiency of a symbolic-execution system.

Path-divergence. It is challenging to compute precise path constraints for real-world programs. The imprecision can arise if the symbolic-execution system is either incomplete

or buggy. Incomplete systems, which symbolically execute only parts of a program, are commonly used because implementing a complete symbolic-execution system involves overwhelming effort. Also, the complexity of a complete system can introduce implementation-level bugs. The imprecision of path constraints leads to path-divergence: the path that the program takes for the generated test input diverges from the path for which test input is generated. Because of the path-divergence problem, a symbolic-execution system may fail to discover a significant number of feasible program paths. If a symbolic-execution system requires the user to provide models for or rewrite all problematic parts of a program, which it cannot symbolically execute, the required manual effort can be impractically high when the size of the problematic code is large.

Complex-constraint. It may not always be possible to solve path constraints because solving the general class of constraints is intractable[1]. Thus, a constraint solver may fail to find solutions to complex, but satisfiable, path constraints such as constraints involving non-linear operations such as multiplication and division and mathematical functions (e.g., `sin` and `log`). The inability to solve path constraints reduces the number of distinct feasible paths a symbolic-execution system can discover. The user may be required to rewrite parts of the program so that the offending constraints are not produced. However, this rewriting requires a user to identify those parts of the program that may generate problematic constraints. Identifying those program parts can require significant manual effort because only a subset of all occurrences of problematic operations (e.g., `sin`) in the code, that operate on symbolic values can potentially generate complex constraints.

1.2 Prior Work

Although a rich body of prior research[2] addresses the three problems, those problems have been only partially solved. In the following, we present an overview of techniques that address the three problems.

[1]The problem is undecidable in general. Under the assumption of finite-length bit-vector representation of numbers, although the problem is decidable, solving it can take too long in practice for its solutions to be useful.

[2]A partial bibliography of papers published in the last decade on symbolic execution and its applications can be found at `http://sites.google.com/site/symexbib/`; References [16] and [63] provide an overview of prior research on symbolic execution.

1.2.1 Techniques for the Path-explosion Problem

Many techniques have been proposed to alleviate the path-explosion problem, and they can be classified into four broad classes. Techniques in the first class avoid exploring paths through certain parts of a program by using a specification of how those parts affect symbolic execution. Some techniques [2, 34] that automatically compute the specification referred to as *summary* in terms of pre-and post-conditions by symbolic execution through all paths of a function. Then, instead of repeatedly analyzing a function for each of its call sites, the summary is used. Other techniques [9, 44, 79] use specifications that are manually created. In some of those techniques [9, 79], specifications of commonly-used abstract data types such as strings and regular expressions are encoded internally in the constraint solver. One of the main limitations of techniques in this class is that their generated constraints can be too complex to solve. In the summary-based techniques, each constraint is complex because it represents a disjunction of path constraints of multiple paths through the summarized methods. The constraints that other techniques generate, which use specialized domains (e.g., Strings), can be challenging to solve in general.

Techniques in the second class [10, 51, 55, 67] are goal-driven: they avoid exploring a large number of paths that are not relevant to the goal of generating test input to cover a specific program entity (e.g., program statement). Ma et al. [51] explore only those paths that lead to the goal. Other techniques [10, 55, 67] use a program's data or control dependencies to choose and symbolically execute only a subset of relevant paths. However, these techniques do not address the path-explosion problem fully because their effectiveness depends on the structure of data or control dependencies, which can differ significantly between programs.

Techniques in the third class are specialized with respect to a program construct or characterstics of a class of programs. Some techniques [38, 68] in this class focus on analyzing loops in a program in a smarter way because loops cause dramatic growth in the number of paths. Other techniques [35, 54] in this class aim to efficiently generate test data for programs that take highly structured inputs (e.g., parser) by leveraging the program's input grammar. Techniques in this class are limited because they are specialized.

Techniques in the fourth class use specific heuristics to choose a subset of all paths to symbolically execute, but while still satisfying the purpose (e.g., obtain high code coverage) of using symbolic execution. The technique presented by Tomb, Brat and Visser [77], which uses symbolic execution to expose bugs, does not symbolically execute paths that span many methods. Other techniques [18, 37, 53, 60] in this class use specific path-exploration strategies that enable generation of test data that cover deep internal parts of a program, which are difficult to cover otherwise. Techniques in this class can fail to discover program behavior that a systematic (but inefficient) technique can discover because of their use of heuristics.

1.2.2 Techniques for the Path-divergence Problem

There are two orthogonal approaches to address the path-divergence problem. One approach is to reduce the effort required in implementation of a symbolic-execution system that can symbolically execute all parts of a program, and thus, compute precise path constraints. The other approach is to reduce the effort that is required from the user of a symbolic-execution system to specify models for parts of a program that cannot be symbolically executed.

Godefroid and Taly [39] propose a technique based on the first approach, which partially automates construction of a symbolic-execution system for x86 binary code. Their technique automatically synthesizes functions corresponding to each instruction that is to be symbolically executed such that those functions update the program's symbolic state according to the semantics of corresponding instructions. This technique does not address other problems (e.g., native methods of Java, multiple programming languages) that also make computing precise path constraints challenging.

Xiao et al. [83] recently proposed a technique based on the second approach, which identifies and reports problems that prevent a test-input generation tool, which uses symbolic execution, from achieving high structural code coverage. One reason for the failure to achieve high coverage is that the external libraries that a program uses are not symbolically executed. Their technique uses a dynamic data-dependence analysis to identify those

methods in the external libraries that cause reduction in code covergage, and require the user to specify models for only those methods.

1.2.3 Techniques for the Complex-constraint Problem

Techniques that address the problem of solving complex constraints can be classified into two classes. The first class consists of two techniques. The first technique is dynamic symbolic execution, which is described in Section 2.1. This technique executes a program both concretely[3] and symbolically along the path for which the path constraint is to be computed. To address the complex-constraint problem, during symbolic execution, if the path constraint becomes too complex to solve, the dynamic-symbolic-execution technique simplifies the constraint by replacing symbolic values with concrete values from concrete execution. In the second technique, Pasareanu, Rungta, and Visser [62] also proposed to use concrete values to simplify complex constraints. However, unlike dynamic symbolic execution, they do not use concrete values obtained from normal execution, but instead they identify a part of the complex constraint that can be solved, and use concrete solutions of the solvable part to simplify the complex constraint. It is not clear from the limited published empirical results to what extent these two techniques solve the complex-constraint problem for realistic programs.

Techniques in the second class [11, 72, 48] model the problem of finding solutions of a constraint over N variables as a search problem in an N-dimensional space. The goal of the search to find a point in the N-dimensional space such that the coordinates of the point represent one solution of the constraint. These techniques use meta-heuristic search methods [33] to solve such search problems. The advantage of using meta-heuristic methods is that those methods can naturally handle constraints over floating point variables and constraints involving arbitrary mathematical functions such as `sin` and `log`. The limitations of such techniques arise because of the incompleteness of the meta-heuristic search methods that may fail to find solutions to a constraint efficiently, even if it is satisfiable.

[3] *Concrete execution* refers to normal execution of a program with non-symbolic values.

1.3 Goals of This Research

The overall motivation for this research is to improve programmer productivity and software quality. Because symbolic execution is a technique that is used to address various problems (see Section 2.2) that arise in developing high-quality software, to address the aforementioned goal, this research focuses on facilitating application of symbolic execution to real-world programs. As mentioned in Section 1.1, applying symbolic execution to real-world programs requires solving three important problems: path-explosion, path-divergence, and complex-constraint. Thus, the goal of this research is to develop techniques that address those three problems.

The goal, with respect to path-explosion problem, is to develop techniques that will (1) reduce the overhead of symbolic execution of individual program paths, and (2) choose and symbolically execute only a subset of all paths that the naive symbolic-execution technique would explore. With respect to the path-divergence problem and the complex-constraint problem, the goal is to develop techniques that reduce the manual effort that is required from a user (of a symbolic-execution system). Manual effort may be required to provide models for or suitably rewrite problematic parts of a program if either (1) the path-divergence problem arises because the implementation of the symbolic execution system is incomplete (e.g., cannot symbolically execute all parts of a program) or (2) the underlying solver cannot solve the constraints.

1.4 Overview of the Dissertation

Efficient and automatic symbolic execution of real-world programs is a significant goal for both theoretical and practical reasons. Theoretically, achieving this goal requires solving an intractable problem (i.e., solving constraints). Practically, achieving this goal requires overwhelming effort to implement a symbolic-execution system that can precisely and automatically symbolically execute real-world programs. These theoretical and practical challenges lead to the General Problem described in Section 1.1. This research addresses the three specific problems (described in Section 1.1)—path-explosion, path-divergence, and complex-constraint—that lead to the General Problem. This research developed three

new techniques—type dependence analysis, heap cloning, and Contest—that partially solve those three problems.

1.4.1 Complex-constraint Problem

The ideal solution for the complex-constraint problem is a constraint solver capable of solving all types of constraints. However, because constraint solving is an intractable problem, our solution is based on a more pragmatic approach. We require the user to provide models for or rewrite those parts of a program that generate complex constraints so that complex constraints are not generated when the (modified) program and models are symbolically executed. However, manually identifying which parts of a program can generate complex constraints and developing models or rewriting can be labor intensive. The user has to first examine the code to understand the root cause and then, develop models or rewrite the program.

This research developed a technique, called *type-dependence analysis* that helps the user to statically identify all problematic parts of a program that may generate complex constraints. Before performing symbolic execution, the user can run this analysis and fix the problematic parts that the analysis reports. One question that arises here is whether it is useful to identify all problematic parts statically. A reasonable alternative to our static-analysis-based solution is to symbolically execute a program, and if a computed path constraint is too complex to solve then identify program parts that generated those constraints. However, our static-analysis-based approach provides advantages similar to those of a compiler that finds all compilation errors before executing a program. Finding all problematic parts at the same time provides an overall understanding of whether and how those parts are interrelated. If they are interrelated, then the user can develop a better solution to address those problems, compared to discovering one problem at a time in the aforementioned alternative solution.

1.4.2 Path-divergence Problem

The ideal solution for the path-divergence problem, which arises partly due to incompleteness in the implementation of a symbolic-execution system, is a complete system that can

symbolically execute all parts of a program. However, because implementing such a complete system may require overwhelming effort, our solution is based on a more pragmatic approach. We require the user to specify models for or rewrite parts of a program that a symbolic-execution system cannot symbolically execute, but that introduce imprecision in path constraints. However, manually identifying those problematic parts can be labor intensive.

Two complimentary techniques developed in this research reduce the burden on the user for identifying problematic parts of a program. The first technique is the type-dependence analysis that also addresses the complex-constraint problem as described in Section 1.4.1. Type-dependence analysis statically identifies and reports those parts of a program that the underlying symbolic-execution system cannot symbolically execute, but that may operate on symbolic values, and thus may introduce imprecision. The second technique, called *heap cloning*, is a program-transformation technique. Through dynamic analysis, heap cloning identifies those program parts that cannot be symbolically executed, but that actually introduce imprecision during symbolic execution due to sideeffects of problematic code.

Both techniques are complimentary because any one technique is not clearly superior to the other. On one hand, the type-dependence analysis can produce false positives, but does not produce false negatives. On the other hand, heap cloning can produce false negatives, but does not produce false positives. In the context of path-divergence problem, if an analysis produces false positives it means that the analysis may determine that some parts of the program may introduce imprecision, whereas they do not actually introduce imprecision. An analysis that produces false positives increase the manual effort. If an analysis produces false negatives it means that the analysis fails to identify parts that indeed introduce imprecision. An analysis that produces false negatives do not help the user to identify imprecision. Thus, relying on only one of the two techniques is not beneficial. The false-positive problem of the type-dependence is expected because it uses static analysis, and the reasons (e.g., context-sensitivity) for such imprecision in static analyses are well-understood. In contrast, standard dynamic analyses usually do not generate false negatives. However, in our context, a dynamic analysis can generate false negatives because detecting

whether imprecision is introduced requires the knowledge of internals of the problematic code, which we do not have because the code cannot not analyzed.

1.4.3 Path-explosion Problem

Two complimentary techniques developed in this research address the path-explosion problem. The first technique is the heap cloning technique, which also addresses the path-divergence problem as described in Section 1.4.2. Heap cloning solves the problems (described in Section 2.1.2.2) that arise in instrumenting Java's standard library classes. By solving those problems, heap cloning enables implementation of precise dynamic-symbolic-execution systems for Java that use standard Java virtual machines for efficient symbolic execution. Because such systems are more efficient than systems that do not use standard virtual machines, they can symbolically execute more paths within a given time budget, and thus, alleviate the path-explosion problem.

The second technique [3], called *Contest*, addresses the path-explosion problem that arises in symbolic execution of event-driven programs. Contest leverages the specific structure of those programs, which input a sequence of events. The technique uses a variation of the standard breadth-first path-exploration algorithm. In each iteration, Contest explores paths that are extensions of feasible paths that were discovered in the previous iteration. Contest's improved efficiency compared to the breadth-first algorithm comes from a notion of subsumption between event sequences that Contest uses to choose and extend only a subset of all feasible paths discovered in the previous iteration, whereas the breadth-first algorithm extends all of them. Even if some paths from the previous iteration are not extended, the subsumption condition ensures that Contest is complete. Completness of Contest means that Contest is guaranteed to generate an event sequence that leads to execution of a branch if any symbolic-execution based path-exploration algorithm can generate an event sequence to execute that branch.

This dissertation also presents a novel approach that uses symbolic execution to generate test inputs for a specific class of programs, called *Apps*, that run on modern mobile devices such as smartphones and tablets. These apps input sequences of events such as a tap on

the device's touch screen and a key press on the device's keyboard. The approach uses the Contest technique to address the path-explosion that arises while generating long event sequences.

1.5 Contributions

This research makes three major contributions.

1. Three new techniques that address three important problems of symbolic execution. Compared to existing techniques, the new techniques

   - reduce the manual effort that may be required to symbolically execute those programs that either generate complex constraints or parts of which cannot be symbolically executed due to limitations of a symbolic-execution system.
   - improve the usefulness of symbolic execution (e.g., expose more bugs in a program) by enabling discovery of more feasible paths within a given time budget.

2. A novel approach that uses symbolic execution to generate test inputs for Apps that run on modern mobile devices such as smartphones and tablets.

3. Implementations of the above techniques and empirical results obtained from applying those techniques to real-world programs that demonstrate their effectiveness.

CHAPTER II

SYMBOLIC EXECUTION OVERVIEW

Symbolic execution [46] is a program-analysis technique that uses symbolic values, instead of concrete values, as program inputs, and represents the values of program variables as symbolic expressions of those inputs. At any point during symbolic execution, the state of a symbolically-executed program includes the symbolic values of program variables at that point, a path constraint on the symbolic values to reach that point, and a program counter.

Definition 1 (Conjunct)**.** *A* conjunct *is an atomic constraint that is generated as a result of different program statements that update the path constraint, such as a branch whose outcome depends on symbolic values.*

Definition 2 (Path Constraint)**.** *The* path constraint (PC) *of a program path p is a boolean formula over the symbolic inputs, which is a logical conjunction of conjuncts that the program inputs must satisfy for an execution to follow the path p.*

At each branch point during symbolic execution, the PC is updated with conjucts on the inputs such that (1) if the PC becomes unsatisfiable, the corresponding program path is infeasible, and symbolic execution does not continue further along that path and (2) if the PC is satisfiable, any solution of the PC is a program input that executes the corresponding path. The *program counter* identifies the next statement to be executed.

To illustrate, consider the code fragment[1] in Figure 1(a) that swaps the values of integer variables `x` and `y`, when the initial value of `x` is greater than the initial value of `y`; we reference statements in the figure by their line numbers. Figure 1(b) shows the symbolic-execution tree for the code fragment. A *symbolic-execution tree* is a compact representation of the execution paths followed during the symbolic execution of a program. In the tree, nodes represent program states, and edges represent transitions between states. The numbers

[1]This example, which we use to illustrate symbolic execution,is taken from Reference [43].

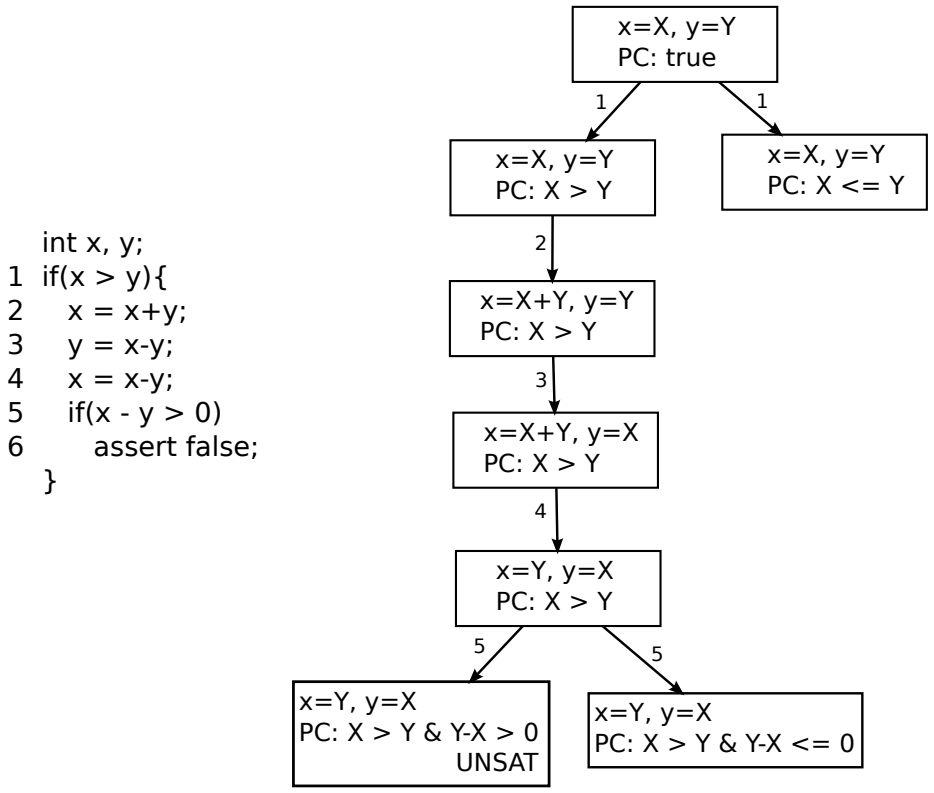


Figure 1: (a) Code that swaps two integers, (b) the corresponding symbolic-execution tree, and (c) test data and path constraints corresponding to different program paths.

shown at the upper right corners of nodes represent values of program counters. Before execution of statement 1, the PC is initialized to true because statement 1 is executed for any program input, and `x` and `y` are given symbolic values X and Y, respectively. The PC is updated appropriately after execution of `if` statements 1 and 5. The table in Figure 1(c) shows the PC's and their solutions (if they exist) that correspond to three program paths through the code fragment. For example, the PC of path (1,2,3,4,5,8) is $X > Y \& Y - X <= 0$. Thus, a program input that causes the program to take that path is obtained by solving the PC. One such program input is $X = 2, Y = 1$. For another example, the PC of path (1,2,3,4,5,6) is an unsatisfiable constraint $X > Y \& Y - X > 0$, which means that there is no program input for which the program will take that (infeasible) path.

Although the symbolic-execution technique was first proposed in the mid-1970's, the technique has received much attention recently from researchers for two reasons. First,

the application of symbolic execution on large, real-world programs requires solving complex and large constraints. During the last decade, many powerful constraint solvers (e.g., Z3 [22], Yices [24], STP [29]) have been developed. Use of those constraint solvers has enabled the application of symbolic execution to a larger and a wider range of programs. Second, symbolic execution is computationally more expensive than other program analyses. The limited computational capability of older-generation computers made it impossible to symbolically execute large programs. However, today's commodity computers are arguably more powerful than the supercomputers (e.g., Cray) of the 1980's. Thus, today, the barrier to applying symbolic execution to large, real-world programs is significantly lower than it was a decade ago. However, it is still challenging to symbolically execute large, real-world programs.

2.1 Dynamic Symbolic Execution (DSE)

Real-world programs commonly use multiple programming languages, databases, file systems, networks, large frameworks, and advanced programming-language features. Moreover, parts of those programs may be available only in binary form. As a result, it may not be possible to symbolically execute some problematic parts of those programs. There are three alternatives that address this problem.

The first alternative is to ask the user to specify models for or rewrite those parts of a program. However, this alternative does not scale to large programs because it may require significant manual effort. The second alternative is to completely ignore the problematic parts from symbolic execution as if those parts do not exist in the program. This alternative can introduce significant imprecision because the effects of the problematic parts are completely ignored. The third alternative is to use a technique, called dynamic symbolic-execution (DSE), which is also known as concolic execution [36, 70]. This technique does not require any manual effort, and also does not completely ignore the effects of the problematic parts on the program state. For example, in Java, problematic parts such as native methods can be executed only concretely.

Definition 3 (Dynamic Symbolic Execution (DSE))**.** *In DSE, if the path constraint of*

a program path p is to computed, the program is executed concretely with some program inputs that take the path p, and as the program executes along p, symbolic execution along p is performed and the path condition is computed. In the case of when some parts of a program cannot be symbolically executed due to limitations of the symbolic-execution system, then those parts are executed only concretely, while the rest of the program is executed both symbolically and concretely.

On one hand, DSE can compute path constraints without any manual effort even in the presence of some problematic parts. On the other hand, DSE suffers from two problems. First, DSE may fail to compute precise path constraints. Second, implementing DSE for Java is problematic. To illustrate the two problems, we use the example Java program shown in Figure 2. The program P, which consists of two classes: `A` and `L`, inputs an integer (line 11) performs several simple operations. The native method `complex`, which is invoked in line 16 of Figure 2, reads the current value of the field `f`, and stores its corresponding absolute value back into field `f`. Although a DSE implementation for Java may not analyze the code (in the source or binary form) of native methods, for the purpose of discussion, suppose that the following Java code is equivalent to the code of the `complex` method.

```
void complex() {
   int i = this.f;
   if(i < 0)
      i = -i;
   this.f = i;
}
```

2.1.1 Imprecision in Path constraints

Definition 4 (Precise path constraint)**.** *The path constraint PC of a program path p is* precise *if the program takes the path p when it is executed with test inputs that are solutions of PC.*

In DSE, the imprecision in path constraints may arise because problematic parts of a program are executed only concretely. This concrete-only execution, in contrast to both

```
1  class L{
2    int f;
3    L(int v){ this.f = v; }
4    native void complex();
5  }
6  class A{
7    L l;
8    A(int v){ this.l = new L(val); }
9    public static void main(String[] args){
10     init();
11     int i = intInput();
12     A a = new A(i);
13     L x = a.l;
14     int j = x.f + 3;
15     if(j < 0)
16       x.complex();
17     if(x.f < 0) error();
18   }
19 }
```

Figure 2: Example that shows dynamic symbolic execution can compute imprecise path constraints.

symbolic and concrete execution, of problematic parts ignores the effects of those parts on the path constraint and the program state. Specifically, imprecision arises in two cases:

- The value of a memory location is updated inside the problematic parts, whereas the symbolic value corresponding to that memory location is not updated.
- A branching statement in the problematic parts add a conjunct to the path constraint.

Consider the path that the program takes for input `-10`. The path covers all statements in the `main` method. In DSE , suppose we introduce a symbol I to represent the symbolic program input, and symbolically execute along the path. As the concrete input value (i.e., `-10`) propagates through the program variables, the symbolic value is propagated accordingly. For example, because the concrete input value is stored in the field `f` at line 13, the memory location corresponding to the field is mapped to symbolic value I. The correct path constraint for this path is

$$X + 3 < 0 \wedge X < 0 \wedge -X >= 0$$

The first and third conjuncts correspond to the branching statements of lines 15 and 17, respectively. The second conjunct corresponds the branching statement inside the `complex` method. Note that the third conjunct accounts for the side-effects of the native method `complex`, which on the current path reads the value (i.e., `-10`) of the field `f` at the time of its invocation, and then stores the absolute value (i.e., `10`) into the field `f`.

However, it is difficult to automatically compute the correct path constraint in the presence of Java's native methods because symbolic execution of native methods is challenging, if not impossible. As a result, in this example, a DSE system will execute the native method `complex` only concretely and other parts of the program both concretely and symbolically. Concrete-only execution of `complex` will ignore the update to field `f` that is made inside `complex`, and incorrectly compute the path constraint as

$$X + 3 < 0 \wedge X >= 0$$

This path constraint is incorrect because (1) it is missing the conjunct corresponding to the branch in `complex` method, and (2) the second conjunct in this path constraint is different from the third conjunct of the correct path constraint, even if both conjuncts are added to the respective path constraints as a result of execution of the statement in line 17. Moreover, note that this path constraint is unsatisfiable, and as a result, DSE will incorrectly conclude that the path that the constraint corresponds to is infeasible. The conclusion is incorrect because the input `-10` executes that path.

Benefits of precise path constraints. One issue related to precision of path constraint is whether the increase in precision is beneficial. The benefits of increased precision depends on the specific application of symbolic execution. One example where precision of path constraints is important is a recent technique [20] that anonymizes user inputs that lead to software failures in the field using symbolic execution. Given one set of inputs I, the technique uses symbolic execution to generate another set of inputs I_a that are different from I, but that cause the program to take the same path as I. The software developers then use I_a to reproduce the failure. In this application of symbolic execution, if the computed path constraint is imprecise, then a new set of inputs I_a obtained from solving the path

constraints may not take the same path as I. As a result, the technique may not be able to reproduce the failure.

2.1.2 Challenges in Implementing DSE for Java

There are two existing approaches to implement a (dynamic-) symbolic-execution system.

2.1.2.1 Custom-interpreter approach

This approach, which is used in JFuzz [42], Symbolic Pathfinder [60], Kiasan [23], and symbolic-execution systems used in References [66, 77], uses a custom interpreter that executes the program according to the semantics of symbolic execution. It is problematic to use this approach to implement DSE because native methods cannot be concretely executed by a standard JVM . The interpreter manages the program's state and thus, a JVM cannot operate on such an external representation of program's state. Although recent work [62] demonstrated how this approach can concretely execute side-effect free native methods, handling native methods with side-effects still remains problematic.

For example, consider the native method `complex` in the program shown Figure 2. Recall that `complex` reads and updates fields of the receiver object on which the method is invoked. Because the method reads and writes to the program heap, it would not be possible to perform DSE using a custom-interpreter approach.

One potential solution to this problem is to translate the program state before an invocation of a native method to a representation that is familiar to a standard JVM, and after the invocation, translate the program state to the representation that the custom-interpreter uses. However, this solution may not scale well in general because, even "well-behaved" native methods can access not only the program heap that is reachable from their arguments, but also parts of heap that are reachable from any static fields. As a result, for this solution to work in general, significantly large portions of program's heap must be translated before and after calls to native methods.

2.1.2.2 Instrumentation-based Approach

The second approach instruments a program, and executes the instrumented code on a standard JVM. The program is instrumented such that execution of the instrumented program essentially performs symbolic execution of the original program. Two existing techniques are based on this approach. The first technique [43] is used in existing symbolic-execution systems such as JPF-SE [5], STINGER [4], Juzi [32], and the system used in Reference [50]. The second technique is used in existing systems such as JCUTE [69], LCT [47], and TaDa [40]. The main difference between the two techniques is the type of instrumentation they perform. In the first technique, the declared type of each program variable is replaced with another type in the instrumented program so that variables in the instrumented program can store symbolic values. The second technique, instead of changing declared types of variables, maintains an explicit map from variables to symbolic values. Each technique has its unique advantages and disadvantages.

In contrast to the custom-interpreter-based approach, in this approach, it is straightforward to only concretely execute problematic parts of a program by the JVM because both symbolic and concrete execution of the non-problematic parts of the program is achieved by executing the instrumented code on a standard JVM. However, another type of problem arises in this approach that makes computing precise path constraints challenging. Under this approach, the instrumentation code performs the symbolic execution. Thus, for precise symbolic execution, it is necessary to instrument not only the program that is to be symbolically executed, but also the library classes that the program uses.

However, two problems make it difficult to instrument those library classes, and without such instrumentation, the path constraints can be imprecise. First, standard JVMs make implicit assumptions about the internal structures of the core library classes, such as `java.lang.String`. Thus, any instrumentation that violates those assumptions can cause the virtual machine to crash. For example, Sun's virtual machine will crash when fields of user-defined types are added to `java.lang.String` or when types of any fields of the core classes are changed. Second, if library classes are instrumented to use some user-defined classes, those user-defined classes must not use library classes. Otherwise, the result may

be non-terminating recursive calls. For example, suppose `L` is a library class, and the instrumented `L` uses a non-library class `Expression`. If `Expression` internally uses class `L`, it will lead to non-terminating recursive calls. Although in theory, classes such as `Expression` can be written without using any library classes, it is cumbersome to do in practice.

The failure to instrument library classes results in additional code that cannot be symbolically executed on top of the code that is usually problematic (e.g., native methods). This failure exacerbates the imprecision problem (described in Section 2.1.1) that the DSE technique suffers from.

2.2 Applications and Tools

Although symbolic execution has been used to generate test inputs for many different goals, the most well-known use of this approach is to generate test inputs to improve code coverage and expose software bugs (e.g., [15, 37, 43]). Other uses of this approach include privacy-preserving error reporting (e.g., [17]), automatic generation of security exploits (e.g., [12]), load testing (e.g., [86]), fault localization (e.g., [64]), regression testing (e.g., [66]), robustness testing (e.g., [52]), inputs anonymization for testing of database-based applications (e.g., [40]), and testing of graphical user interfaces (e.g., [30]),

For example, Castro, Costa, and Martin [17] use symbolic execution to generate test inputs that can reproduce, at the developer's site, a software failure that occurs at a user's site, without compromising the privacy of the user. Zhang, Elbaum, and Dwyer [86] generate test inputs that leads to a significant increase in program's response time and resource usage. Qi et al. [64] generate test inputs that are similar to a given program input that causes the software to fail, but that do not cause failure. Such newly-generated test inputs are then used to localize the cause of the failure. Santelices et al. [66] generate test inputs that exposes difference in program's behaviors between two versions of an evolving software. Majumdar and Saha [52] use symbolic execution to generate test inputs whose slight perturbation causes a significant difference in a program's output.

A number of tools for symbolic execution are currently available in public domain. For Java, available tools include Symbolic Pathfinder [61], JFuzz [42], and LCT [47]. For C,

tools that are available include Klee [15], S2E [18], and Crest [14]. Finally, Pex [76] is a symbolic execution tool for .NET languages. SAGE [37] is not currently available in public domain, but it has been shown to be very effective on real-world x86 binaries.

CHAPTER III

TYPE-DEPENDENCE ANALYSIS: IDENTIFYING PROBLEMATIC PROGRAM PARTS

This chapter presents a technique that addresses the complex-constraint problem and the path-divergence problem, which are described in Section 1.1. Recall that the complex-constraint problem concerns the capabilities of the underlying decision procedure used to check satisfiability and solve path conditions. If the underlying decision procedure is incapable of (or inefficient in) handling the types of constraints produced during symbolic execution, users must rewrite parts of the program so that the offending constraints are not produced. However, this rewriting requires a user to identify those parts of the program that may generate problematic constraints, which is a difficult task.

The path-divergence problem may arise if symbolic values flow outside the boundaries of the software being symbolically executed. In these cases (e.g., when a symbolic value is passed as a parameter to an external library call), the computed path constraints can be imprecise (see Section 2.1.1). In real-world applications, there can be many instances of this problem, such as calls to native methods in Java and third-party pre-compiled libraries. To address this issue, users must replace the calls to external components that may be reached by symbolic values with calls to stubs that model the components' behaviors. Like the complex-constraint problem, performing this instrumentation requires manual intervention: the users must identify the external calls that may be problematic before actually performing symbolic execution.

To facilitate symbolic execution of real-world applications, where the two aforementioned problems are frequently encountered, we developed and present in this chapter a new approach. Our approach is based on the insight that both problems are caused by the flow of symbolic values to *problematic variables*, such as parameters of library calls or operands of expressions that cannot be handled by the underlying decision procedure. Our

approach is based on a novel static analysis, called type-dependence analysis, that identifies problematic variables before performing symbolic executions. Our type-dependence analysis formulates the problem of identifying variables that may assume symbolic values as a value-flow analysis problem. The analysis is context- and field-sensitive, which has the advantage of providing reasonably precise results. The benefit of our analysis is that it can automatically detect parts of the program that may be problematic for symbolic execution (e.g., a modulo operation that involves at least one symbolic operand). For any such part, the analysis reports to the users the identified problem, together with contextual information, to help them understand the issue and perform necessary program changes.

This chapter also presents a technique that leverages the results of the analysis to instrument a program for symbolic execution. The basic idea behind the instrumentation is to instrument a program such that execution of the instrumented program essentially performs symbolic execution of the original program. Naively applying such instrumentation to the entire application leads to two problems. First, in practice, Java virtual machines make implicit assumptions about the internal structures of some components. Instrumenting such components is thus problematic (see Section 2.1.2.2). Second, symbolic operations are more expensive than their concrete counterparts, even when they operate on concrete values (the extra overhead is incurred in checking whether a value is symbolic or concrete). Therefore, instrumenting those components of the program that may not interact with symbolic values introduces inefficiencies. Because type-dependence analysis can identify which variables may be symbolic, our technique avoids instrumenting parts of the code that have no interactions with symbolic values, thus improving both applicability and efficiency of symbolic execution.

To evaluate our type-dependence analysis and instrumentation technique, we implemented them in a tool, called STINGERand used the tool to perform an empirical evaluation on two real-world Java programs. The results of the studies show that our analysis can be effective in (1) statically identifying areas of the code that would be problematic for symbolic execution, (2) providing useful feedback to the users to guide them in the resolution of the problems, and (3) limiting the instrumentation necessary for symbolic execution.

The contributions of the work presented in this chapter are

- A context- and field-sensitive static flow analysis that can identify the variables in a program that may hold symbolic values, given a set of symbolic inputs. The analysis results enable static identification of program segments that are potentially problematic for symbolic execution and can guide users in instrumenting the program to eliminate the problems.
- A general instrumentation technique that leverages the type-dependence analysis to instrument programs into "symbolic programs" (i.e., programs whose execution essentially performs symbolic execution of the original program).
- A tool, STINGER, that implements our approach for Java.
- A set of empirical studies, performed on two real-world programs, whose results show the usefulness of our approach.

3.1 Type-dependence Analysis

This section presents our type-dependence analysis, which computes the set of program entities that may store symbolic values when a program is symbolically executed. We target a typical scenario in which the user selects a set of variables to hold symbolic input values for a program and then symbolically executes the program. In this context, the type of the selected variables and of other variables that can hold values derived from these selected variables must be symbolic.

Before discussing the details of our analysis, consider a motivating example that illustrates some of the issues that the analysis can help to address. Suppose that we want to symbolically execute the Java program shown in Figure 3, and that `s` represents the (symbolic) input to the program (as shown by the assignment of `Symbolic.integer()` to `s`). On initial inspection, the program contains three potentially-problematic cases: the use of the modulo (`%`) operation, which is not supported by most constraint solver, in line 22; the invocation of native method `clone` in line 23; and the invocation of native method `isPrime`

```
1  public class Object{
2      public static native Object clone();
3  }
4
5  public class M extends Object{
6      int m;
7
8      M(int x){
9        this.m = x;
10     }
11
12     int getM(){
13       return this.m;
14     }
15
16     static native boolean isPrime(int x);
17
18     public static void main(String[] arg){
19         int s = Symbolic.integer();
20         M a = new M(s); M b = new M(4);
21         int p = a.getM(); int q = b.getM();
22         if(isPrime(p) && q % 3 == 0)
23             M c = (M) a.clone();
24     }
25 }
```

Figure 3: Motivating example for type-dependence analysis.

in line 22. However, a more careful inspection reveals that the first two cases are not problematic: the modulo operator never operates on symbolic values and native method `clone` can access only fields of class `Object`,[1] none of which may store symbolic values. As for the third potentially-problematic case, a symbolic value is passed as an argument to native method `isPrime` and is likely to be problematic because the method expects a concrete value. The type-dependence analysis can discover that the first two cases are not problematic but the third case is. For this third case, the analysis can provide context information to help the user understand the problem and address it.

The analysis is called type-dependence analysis because it identifies type dependence between variables.

[1]This conclusion is based on the common assumption that native methods do not use dynamic type discovery and thus access only fields of declared types of their parameters [75].

(assignment)	`p = x`	$\Longrightarrow$	$p \leftarrow x$
(cast)	`p = (ctype) x`	$\Longrightarrow$	$p \leftarrow x$
(binop)	`p = x` $\oplus$ `y`	$\Longrightarrow$	$p \leftarrow x,\ p \leftarrow y$
(uniop)	`p = -x`	$\Longrightarrow$	$p \leftarrow x$
(load)	`p = o.f`	$\Longrightarrow$	$p \xleftarrow{get[f]} o$
(store)	`o.f = x`	$\Longrightarrow$	$o \xleftarrow{put[f]} x$
(static-load)	`p = C.f`	$\Longrightarrow$	$p \leftarrow f$
(static-store)	`C.f = x`	$\Longrightarrow$	$f \leftarrow x$
(return)	`return x`	$\Longrightarrow$	$R_m \leftarrow x$ where, m is the concerned method
(array-new)	`a = new t[size]`	$\Longrightarrow$	$a \xleftarrow{put[length]} size$
(array-assign1)	`a[i] = x`	$\Longrightarrow$	$a \xleftarrow{put[elem]} x$
(array-assign2)	`p = a[i]`	$\Longrightarrow$	$p \xleftarrow{get[elem]} a$
(array-length)	`p = a.length`	$\Longrightarrow$	$p \xleftarrow{get[length]} a$
(invocation)	`x = a.foo(`$a_1, \ldots, a_n$`)`	$\Longrightarrow$	$x \leftarrow R_{foo}, P^1_{foo} \leftarrow a_1, \ldots, P^n_{foo} \leftarrow a_n$

Figure 4: Rules for building the type-dependence graph.

Definition 5 (Type Dependent)**.** *For a given type-correct program, an entity x is* type dependent *on an entity y if and only if, to maintain type correctness, x's type may need to be changed as a consequence of a change in y's type.*

The type-dependence analysis computes a conservative approximation of the type-dependence relation between a given set of entities and the other entities in the program. Type-dependence analysis consists of two phases. The first phase builds a Type-Dependence Graph (TDG) for the program, which encodes direct type-dependence information between program entities. The second phase uses the TDG to identify transitive type dependences.

3.1.1 Building the Type-Dependence Graph

In the first phase, the analysis builds the Type-Dependence Graph (TDG).

Definition 6 (Type-Dependence Graph)**.** *A* Type-Dependence Graph (TDG) *is a directed graph* (N, E)*. N is a set of nodes, each of which represents one of several entities: a static field, a local variable of a method, a field of primitive type, a parameter of a method, or the return value of a method. E is a set of directed edges. An edge $x \leftarrow y$ in E indicates that there is a direct type dependence between the entity represented by y and the entity represented by x (i.e., x is directly type-dependent on y).*

To build the TDG, the analysis processes each program statement once and adds an edge to the graph for each relevant statement, according to the rules shown in Figure 4. For example, the rule for the assignment statement, `p = x` adds an edge from a node x corresponding to variable `x` to a node p corresponding to variable `p`. This edge represents that during symbolic execution if the variable `x` stores symbolic values, then the variable `p` also stores a symbolic values because of the assignment of `x` to `p`.

In the figure, $\oplus$ represents binary operators such as `+` and `-`. $o.f$ represents field f of object o. P_m^i and R_m represent the i^{th} parameter of method m and the return value of method m, respectively. For the definition of the rules, the analysis treats arrays as objects with two fields, *elem* and *length*, that represent all array elements and the length of the array, respectively.

The analysis does not add any edges for those types of statements (e.g., object allocation statement) that are not shown in the figure. The analysis also performs a simple optimization to keep the size of the TDG small. The analysis does add an edge from the TDG if the source of the edge corresponds to a constant literal (e.g., `-11`). This optimization is safe because a constant literal is obviously not a variable that can store symbolic values, and thus, symbolic value cannot flow to the program entity represented by the target of the concerned edge.

3.1.2 Computing Type-Dependent Entities

In the second phase, given a user-specified set of program's input variables, Sym_0, the analysis computes the following five sets.

- Sym, is a set local variables, static fields, formal parameters, and return values of scalar types.

- Sym_{arr} is a set of local variables that can store array objects.
- Sym_{fld} is a set of instance fields.
- Arr_{obj} is a set of array objects.
- Arr_{stmt} is a set of program statements that allocate array objects.

Elements of each set except $Arry_{stmt}$ are type-dependent on variables in Sym_0. A statement s that allocates an array is in $Array_{stmt}$ if the type of the array that s allocates may need to be changed as a consequence of a change in the type of some variable in Sym_0 to maintain type correctness.

For clarity, we first present a context-insensitive version of the analysis and then describe how it can be extended to be context-sensitive. The context-insensitive version of the analysis is represented by the inference rules shown in Figure 5. The analysis initializes Sym to Sym_0 and applies two inference rules—Rule 1 and Rule 2—until a fix-point on Sym is reached. After Sym is computed, the other four sets are computed from Sym using Rule 3-8.

Rule 1 states that an entity p is added to the Sym set if there is another entity x in Sym on which p is directly type dependent. Rule 2 captures transitive type dependence through heap references. It states that entity p must be added to Sym if there is another entity x in Sym and two object references y and q such that (1) p is directly type dependent on a field f of q, (2) the same field f of y is directly type dependent on x, and (3) y and q may point to the same object (expressed using the notation $alias(y, q)$). Without loss of generality, the analysis assumes that points-to information is computed by some analysis (e.g., [73]). Note that this analysis is *field-sensitive* because in Rule 2, the labels $get[f_1]$ and $put[f_2]$ must refer to the same fields, which is in contrast to a *field-based* analysis that does not distinguish between different fields of an object.

Rule 3, Rule 4, and Rule 5 use the notation $pt(a)$ to represent the set of all abstract objects (including array objects) that variable a may point to. Rule 3 states that an abstract array object r is in Arr_{obj} if an array-type variable a may point to r, and a may store a value (i.e., the value of variable x) that is not type-compatible with a's current element

$$\frac{p \leftarrow x,\ x \in Sym}{p \in Sym} \tag{1}$$

$$\frac{p \xleftarrow{get[f_1]} q,\ y \xleftarrow{put[f_2]} x,\ f_1 = f_2,\ alias(y,q),\ x \in Sym}{p \in Sym} \tag{2}$$

$$\frac{r \in pt(a),\ a \xleftarrow{put[elem]} x,\ x \in Sym}{r \in Arr_{obj}} \tag{3}$$

$$\frac{r \in pt(a),\ a \xleftarrow{put[length]} x,\ x \in Sym}{r \in Arr_{obj}} \tag{4}$$

$$\frac{r \in Arr_{obj},\ r \in pt(v)}{v \in Sym_{arr}} \tag{5}$$

$$\frac{alloc(s,r),\ r \in Arr_{obj}}{s \in Arr_{stmt}} \tag{6}$$

$$\frac{y \xleftarrow{put[f]} x,\ x \in Sym}{f \in Sym_{fld}} \tag{7}$$

$$\frac{y \xleftarrow{put[f]} x,\ x \in Sym_{arr}}{f \in Sym_{fld}} \tag{8}$$

Figure 5: Context-insensitive inference rules for type dependence analysis.

type. Rule 4 states that r is in Arr_{obj} if a may point to r and *length* of a may not be of integer type as a result of change in the types of variables in Sym_0. Rule 5 states that a variable v is in Sym_{arr} if v may point to some array object r in Arr_{obj}. Rule 6 states that a statement s that allocates an array is in $Arry_{stmt}$ if s corresponds to the abstract array object r, which is represented by the notation $alloc(s,r)$, and r is Arr_{obj}.

Finally, Rule 7 and Rule 8 compute type-dependent instance fields. Rule 7 (Rule 8) states that a field f is type-dependent on a variable in Sym_0 if a primitive-type (array-type) local variable x that is type-dependent on Sym_0 is stored into field f of some reference-type variable y.

Context-sensitive Analysis. The context-insensitive analysis described above may compute unnecessarily-large *Sym* sets. In the example in Figure 3, for instance, the analysis would not distinguish between the two calls to the `getM` method and, thus, would not be able to detect that variable `q` is not type-dependent on variable `s`. To improve the precision

of the analysis, we define a context-sensitive version of the TDG using an approach similar to cloning-based context-sensitive points-to analysis [81]. First, we create multiple nodes for each entity—one for each calling context of the method that contains the entity. The only exceptions are entities that correspond to global variables (e.g., static fields in Java) that are represented with a single node in the context-sensitive TDG. Second, we create copies of the TDG's edges so that if an edge exists between two nodes, there is an edge between corresponding (context-specific) copies of the nodes. Finally, because cloning-based approaches can lead to an exponential explosion in the size of the graphs, we use binary decision diagrams [1] as the data structure to compactly represent context-sensitive TDGs.

After building the context-sensitive TDG, the analysis uses a context-sensitive version of the inference rules described in Figure 5 to compute Sym. We obtain the context-sensitive inference rules by modifying the context-insensitive rules: we identify each entity in the rule with respect to a specific context c. The context-sensitive version of Rule 1 in Figure 5, for instance, is

$$\frac{p^c \leftarrow x^c,\ x^c \in Sym}{p^c \in Sym}$$

3.2 Selective Program Instrumentation

In the instrumentation-based approach (described in Section 2.1.2.2) of implementing symbolic-execution systems, a naive instrumentation technique would instrument every statement, variable, and field of the entire program and its libraries based on the assumption that any program variable can potentially store symbolic values. In practice, this approach is not feasible for two problems. First, in the case of virtual-machine based programming languages (e.g., Java), instrumentation of Java's standard library classes is problematic (described in Section 2.1.2.2). Second, the instrumented program produced from naive instrumentation can be more inefficient than a program that is instrumented using the knowledge of only the subset of all variable that may store symbolic values.

This section presents a program-instrumentation technique that leverages the results of type-dependence analysis to instrument, in an automated way, only a subset of the program. By doing this, our technique mitigates (when it does not completely solve) the two aforementioned problems.

3.2.1 Box and Unbox Operators

Our instrumentation technique supports two operators that enable selective program instrumentations: box and unbox.

Definition 7 (Box Operator)**.** *The box operator converts a concrete value to a corresponding symbolic value.*

Definition 8 (Unbox Operator)**.** *The unbox operator converts a symbolic value created by the box operator to the corresponding concrete value.*

These operators are needed to handle program entities that must be of symbolic types for type correctness, but may store either symbolic or concrete values depending on the contexts. The operators let these entities store (boxed) concrete values whenever necessary. The technique automatically adds to the program appropriate boxing and unboxing operators to enable assignments between entities of symbolic and concrete types. Note that unboxing a symbolic value (i.e., a symbolic value that is not the result of a boxing operation) would cause a run-time error. However, the technique guarantees that such a situation will never occur due to its use of the results of the conservative type-dependence analysis.

Before presenting the formal definition of the technique, we illustrate some features of our approach by showing, in Figure 6, the instrumented version of the example program from Figure 3. In the instrumented program, `Expression` represents the type of symbolic expressions, and methods `makeSymbolic` and `makeConcrete_int` represent box and unbox operators, respectively. For each field that may store a symbolic value, such as `m`, the instrumentation adds a new field of symbolic type. Similarly, for each method that may operate on symbolic values, a new method is added that may take symbolic values as arguments or return symbolic values. Note that because the analysis determines that variable `q` can never store a symbolic value at runtime, `q`'s type is unchanged, and the `%` operation is not replaced by its corresponding symbolic operation. In contrast, `p`'s type is changed to `Expression` because the analysis determines that it may store a symbolic value. When a symbolic version of a method is created, only those calls that may pass and/or receive symbolic values

```
1  public class M {
2      int m;
3      Expression m$Stinger;
4
5      M(int x) {
6        this(Symbolic.makeSymbolic(x));
7      }
8
9      int getM() {
10       return Symbolic.makeConcrete_int(getM$Stinger());
11     }
12
13     static native boolean isPrime(int i);
14
15     M(Expression x) {
16       this.m$Stinger = x;
17     }
18
19     Expression getM$Stinger() {
20       return this.m$Stinger;
21     }
22
23     static native boolean isPrime$Stinger(Expression expression);
24
25     public static void main(String[] strings) {
26         M a = new M(Symbolic.symbolic_int());
27         M b = new M(4);
28         Expression p = a.getM$Stinger();
29         int q = b.getM();
30         if (isPrime$Stinger(p) && q % 3 == 0)
31             M c = (M) a.clone();
32     }
33 }
```

Figure 6: Instrumented version of the example from Figure 3.

are changed to invoke the new method. In the example, for instance, `getM$Stinger()` is called on `a` because a symbolic value may be returned by the method at that callsite. Conversely, the call to `getM()` on `b` is unchanged, as only concrete values are returned at the corresponding callsite.

Source language

$l \in \text{Local},\ f \in \text{Field},\ r \in \text{RefType},\ m \in \text{Method}$

$n \in \text{NumType}$	n	::=	int | short | char | long | byte | float | double
$\tau \in \text{Type}$	τ	::=	n | boolean | r | $\tau[\,]$
$i \in \text{Immediate}$	i	::=	l | const
$e \in \text{Expr}$	e	::=	i | i_1 binop i_2 | unop i | $l.f^{(\tau)}$ | $C.f^{(\tau)}$ | $(\tau)\ i$ | $l[i]^{(\tau)}$ | $l.\text{length}^{(\tau)}$ | new $\tau[i]$ | $m(e_1, \ldots)$
$s \in \text{Stmt})$	s	::=	$l = e$ | $l.f = i$ | $C.f = i$ | $l[i_1]^{(\tau)} = i_2$ | return e

binop $\in \{+, -, *, /, \%, =, >, \geq, <, \leq, \neq\}$
unop $\in \{-, !\}$

Extension for symbolic execution

$\tilde{l} \in \text{SymLocal},\ \tilde{f} \in \text{SymField}$

$\tilde{\tau} \in \text{SymType}$	$\tilde{\tau}$	::=	Expr | ExprArray | BoolArray | RefArray
$\tilde{i} \in \text{SymImmediate}$	$\tilde{i}$	::=	$\tilde{l}$
$\tilde{e} \in \text{SymExpr}$	$\tilde{e}$	::=	$\text{box}^{\tilde{\tau}}(e)$ | $\tilde{i}$ | $\text{symbinop}(\tilde{e_1}, \tilde{e_2})$ | symunop $\tilde{i}$ | $l.\tilde{f}$ | $\text{cast}^{\tau}(\tilde{e})$ | $\text{array\_get}^{\tilde{\tau}}(\tilde{l}, \tilde{e})$ | $\text{array\_len}^{\tilde{\tau}}(\tilde{l})$ | $\text{new\_array}^{\tilde{\tau}}(\tilde{e})$
$\tilde{s} \in \text{SymStmt}$	$\tilde{s}$	::=	$\tilde{l} = \tilde{e}$ | $l = \text{unbox}^{\tilde{\tau}}(\tilde{e})$ | $a.\tilde{f} = \tilde{e}$ | $\text{array\_set}^{\tilde{\tau}}(\tilde{l}, \tilde{e_1}, \tilde{e_2})$

symbinop $\in$ {_plus,_minus,_mul,_div,_mod,_eq, _gt, _ge, _lt, _le, _ne}
symunop $\in$ {_neg,_not}

Figure 7: Syntax of the source language and its extensions for symbolic execution.

3.2.2 Source and Target Languages

We define the instrumentation on a subset of Java, referred to as *source* language, that contains only those types of statements that are instrumented. The instrumentation of a program in *source* language produces a program in *target* language. Figure 7 presents the source language and its extensions for symbolic execution. The target language is the union of the source language and its extensions.

Both the source and the target languages are statically and explicitly typed according to Java's type rules. Types in the source language include all types supported by Java. The target language supports four symbolic types, namely Expr, ExprArray, BoolArray,

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\textit{stype} : \text{Type} \rightarrow \text{SymType}} \\
\textit{stype}(\tau) & = & \textsc{Expr} & \tau \in \text{NumType} \\
\textit{stype}([]\tau) & = & \textsc{ExprArray} & \tau \in \text{NumType} \\
\textit{stype}([]\text{boolean}) & = & \textsc{BoolArray} & \\
\textit{stype}([]r) & = & \textsc{RefArray} &
\end{array}
$$

Figure 8: Definition of *stype* maps concrete type to symbolic type.

and RefArray, that represent types of symbolic expressions, arrays of symbolic expressions, arrays of boolean values, and arrays of references, respectively. Each symbolic array type can also have symbolic length. The correspondence between concrete and symbolic types (for concrete types that have a corresponding symbolic type) is defined by function *stype*, shown in Figure 8.

Expressions include local variables, constants, unary and binary operations, field references, casts, array references, array length and array allocation expressions. In the source language, τ represents the element type of array l in terms $l[i]^{(\tau)}$ and $l.\texttt{length}^{(\tau)}$, and the type of field f in terms $l.f^{(\tau)}$ and $C.f^{(\tau)}$.

In the target language, there is one syntactic category for each category in the source language, represented by the same symbol with a tilde on the top. In addition, for each unary, binary, and comparison operator in the source language, the target language provides a corresponding operator that operates on symbolic values. The definition of the language extensions uses the following terminology (where $\tilde{\tau}$ denotes the type of array element).

- array_get$^{\tilde{\tau}}(\tilde{l}, \tilde{e})$ is an operation that returns the $\tilde{e}^{th}$ element of symbolic array $\tilde{l}$
- array_len$^{\tilde{\tau}}(\tilde{l})$ returns the length of $\tilde{l}$; new_array$^{\tilde{\tau}}(\tilde{e})$ allocates a symbolic array of size $\tilde{e}$
- array_set$^{\tilde{\tau}}(\tilde{l}, \tilde{e}_1, \tilde{e}_2)$ stores symbolic expression $\tilde{e}_2$ as the element at index $\tilde{e}_1$ of array $\tilde{l}$.

The box operator is represented by $box^{\tilde{\tau}}(e)$, which transforms the concrete value e into the corresponding symbolic value of type $\tilde{\tau}$. Analogously, $unbox^{\tilde{\tau}}(\tilde{e})$ indicates the instrumentation of the symbolic value contained in $\tilde{e}$, of type $\tilde{\tau}$, into its original concrete value and type.

3.2.3 Instrumentation

The instrumentation is performed in two steps. In the first step, the instrumentation adds new fields, methods, and local variables of symbolic types to the program. For each field that may store symbolic values, the instrumentation adds a new field with corresponding symbolic type. For each method m that may operate on symbolic values, the instrumentation adds a new method m_s, which may potentially have parameters and return value of symbolic types. Also, for each of m's local variables v, if v may store symbolic values, the instrumentation adds a local variable of corresponding symbolic type to m_s; otherwise, the original v is added to m_s. Finally, the instrumentation moves all statements of m to m_s, and transforms m into a proxy that invokes m_s and performs boxing and unboxing of parameters and/or return values as needed. Note that, even if the analysis is context-sensitive, it generates at most one variant of each method because the results of the analysis are unified over all contexts.

In the second step of the instrumentation, statements in the newly-added methods are instrumented according to the rules provided in Figure 9 and Figure 10. Each rule defines how a specific statement in the source language is instrumented and is applicable only if the respective guard is satisfied. The rules use the following notations:

- For a given local variable or field x that may store symbolic values, $\overline{x}$ represents the corresponding entity of symbolic type added by the instrumentation in the first step. If x cannot store a symbolic value, then $\overline{x}$ simply represents the original entity. In particular, if x is a constant, $\overline{x}$ always represents x.

- $\overline{\tau}$, represent the symbolic type corresponding to a concrete type τ, as defined by function *stype*.

- For an expression e of concrete type τ, $< e >$ represents $\texttt{box}^{\overline{\tau}}(e)$ (i.e., e boxed as a value of its corresponding symbolic type). For an expression $\tilde{e}$ of a symbolic type, $< \tilde{e} >$ represents $\tilde{e}$ itself.

Original statement	Instrumented statement	Guard	
$[\![l = i]\!]$	$[\![\overline{l} = < \overline{i} >]\!]$	$\overline{l} \neq l$	(1)
$[\![l = i_1 \texttt{ binop } i_2]\!]$	$[\![\overline{l} = \texttt{box}^{\text{EXPR}}(i_1 \texttt{ binop } i_2)]\!]$	$\overline{l} \neq l, \overline{i_1} = i_1, \overline{i_2} = i_2$	(2)
	$[\![\overline{l} = \texttt{symbinop(} < \overline{i_1} >, < \overline{i_2} > \texttt{)}]\!]$	$\overline{i_1} \neq i_1$ or $\overline{i_2} \neq i_2$	(3)
$[\![l = \texttt{unop } i]\!]$	$[\![\overline{l} = \texttt{box}^{\text{EXPR}}(\texttt{unop } i)]\!]$	$\overline{l} \neq l, \overline{i} = i$	(4)
	$[\![\overline{l} = \texttt{symunop}(\overline{i})]\!]$	$\overline{i} \neq i$	(5)
$[\![l.f = i]\!]$	$[\![l.\overline{f} = < \overline{i} >]\!]$	$\overline{f} \neq f$	(6)
$[\![l_1 = l_2.f^{(\tau)}]\!]$	$[\![\overline{l_1} = < l_2.\overline{f} >]\!]$	$\overline{l_1} \neq l_1$	(7)
	$[\![l_1 = \texttt{unbox}^{\overline{\tau}}(l_2.\overline{f})]\!]$	$\overline{l_1} = l_1, \overline{f} \neq f$	(8)
$[\![C.f = i]\!]$	$[\![C.\overline{f} = < \overline{i} >]\!]$	$\overline{f} \neq f$	(9)
$[\![l = C.f^{(\tau)}]\!]$	$[\![\overline{l} = < C.\overline{f} >]\!]$	$\overline{l} \neq l$	(10)
	$[\![l = \texttt{unbox}^{\overline{\tau}}(C.\overline{f})]\!]$	$\overline{l} = l, \overline{f} \neq f$	(11)
$[\![l_1 = (\tau)\ l_2]\!]$	$[\![\overline{l_1} = \texttt{box}^{\overline{\tau}}((\tau)\ l_2)]\!]$	$\overline{l_1} \neq l_1, \overline{l_2} = l_2$	(12)
	$[\![\overline{l_1} = \texttt{cast}^{\tau}(\overline{l_2})]\!]$	$\overline{l_1} \neq l_1, \overline{l_2} \neq l_2$	(13)

Figure 9: Instrumentation rules for program statements.

As examples, we discuss instrumentation rules for only two types of statements: assignments of a local variable or a constant to a local variable (Rule 1) and assignments of a field to a local variable (Rule 7 and Rule 8). Other rules are based on similar intuition.

According to Rule 1, assignment statements of the form $l = i$ are instrumented only if l may store a symbolic value. If so, a local variable of symbolic type that corresponds to l, $\overline{l}$, is added and becomes the l-value of the instrumented statement. If i is a non-constant local variable and has a corresponding local variable of symbolic type, $\overline{i}$, $\overline{i}$ becomes the r-value of the instrumented statement. Otherwise, if i is either a constant or a local variable without a corresponding local of symbolic type, i's value is boxed and assigned to $\overline{l}$.

We discuss Rule 7 and Rule 8 concerning statements of type $l_1 = l_2.f^{(\tau)}$ because those rules make use of the unbox operator. There are two rules that involve these statements. In

the first case (Rule 7), l_1 has a corresponding local of symbolic type, $\overline{l}_1$, which becomes the l-value of the instrumented statement. If field f has a corresponding field of symbolic type, $\overline{f}$, the value of $\overline{f}$ of l_2 is assigned to $\overline{l_1}$; otherwise, the value of field f of l_2 is boxed and assigned to $\overline{l_1}$. In the second case (Rule 8), where l_1 does not have a corresponding local of symbolic type, but f has a corresponding field of symbolic type, $l_2.f$'s value is unboxed and assigned to l_1.

Figure 10 shows the instrumentation rules for statements referencing arrays.

3.3 Empirical Evaluation

This section first discusses our implementation of STINGER, then it describes our subject programs, and finally, it presents the results of our study that addresses three research questions.

3.3.1 Implementation

STINGER works on Java bytecode and leverages the SOOT framework [78]. The type-dependence analysis is implemented using Jedd [49], a Java language extension that supports the use of binary decision diagrams to store and manipulate relations.

Figure 11 shows a dataflow diagram of STINGER. STINGER inputs a program P in Java bytecode and a set of program variables that store the program's input (called Sym_0 in Section 3.1). STINGER outputs P' that is a instrumented version of P and a report of (1) problematic parts of P that may generate complex constraints that the underlying constraint solver cannot solve, and (2) parts of P where, symbolic values that may flow into parts of programs that are not symbolically executed (e.g., native methods in Java). If P has some problematic parts, the user may choose to examine those parts and provide models for or rewrite those parts and finally, run STINGER again.

STINGER performs two steps. In Step 1, STINGER performs type-dependence analysis, and identifies and produces a report of problematic parts of P. Step 1 also produces the set, Sym that contains all program entities that may store symbolic values. In Step 2, STINGER uses Sym to only selectively instrument P to produce P'.

Original statement Instrumented statement	Guard	
$[\![s : l = \texttt{new}\ \tau[i]]\!]$		
$\quad[\![\overline{l} = \texttt{new\_array}^{\overline{\tau}}(< \overline{i} >)]\!]$	$s \in Arr_s$	(11)
$[\![l_1 = l_2.\texttt{length}^{(\tau)}]\!]$		
$\quad[\![\overline{l_1} = \texttt{box}^{\text{EXPR}}(l_2.\texttt{length})]\!]$	$\overline{l_1} \neq l_1, \overline{l_2} = l_2$	(12)
$\quad[\![l_1 = \texttt{unbox}^{\text{EXPR}}(\texttt{array\_len}^{\overline{\tau}}(\overline{l_2}))]\!]$	$\overline{l_1} = l_1, \overline{l_2} \neq l_2$	(13)
$\quad[\![\overline{l_1} = \texttt{array\_len}^{\overline{\tau}}(\overline{l_2})]\!]$	$\overline{l_1} \neq l_1, \overline{l_2} \neq l_2$	(14)
$[\![l[i_1]^{(\tau)} = i_2]\!]$		
$\quad[\![\texttt{array\_set}^{\overline{\tau}}(< \overline{l} >, < \overline{i_1} >, < \overline{i_2} >)]\!]$	$\tau \in$ NumType, $\overline{l} \neq l$ or $\overline{i_1} \neq i_1$	(15)
$\quad[\![\texttt{array\_set}^{\overline{\tau}}(< \overline{l} >, < \overline{i_1} >, i_2)]\!]$	$\tau \in$ RefType or $\tau =$ boolean, $\overline{l} \neq l$ or $\overline{i_1} \neq i_1$	(16)
$[\![l_1 = l_2[i]^{(\tau)}]\!]$		
$\quad[\![\overline{l_1} = \texttt{box}^{\tilde{\tau}}(l_2[i])]\!]$	$\overline{l_1} \neq l_1, \overline{l_2} = l_2, \overline{i} = i$	(17)
$\quad[\![\overline{l_1} = \texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >)]\!]$	$\tau \in$ NumType, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i, \overline{l_1} \neq l_1$	(18)
$\quad[\![l_1 = \texttt{unbox}^{\text{EXPR}}(\texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >))]\!]$	$\tau \in$ NumType, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i, \overline{l_1} = l_1$	(19)
$\quad[\![l_1 = \texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >)]\!]$	$\tau =$ boolean, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i$	(20)
$\quad[\![l_1 = (\tau)\ \texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >)]\!]$	$\tau \in$ RefType, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i$	(21)

Figure 10: Tranformation rules for statements that reference arrays.

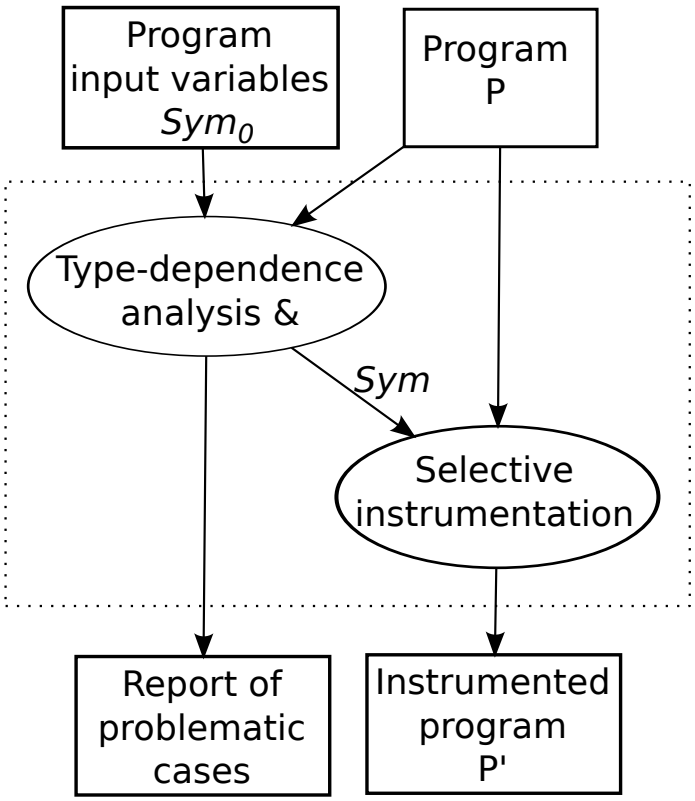


Figure 11: The STINGER type-dependence analyzer and instrumentor.

3.3.2 Study

As subjects for our study, we used two freely-available Java programs: NANOXML and ANTLR. NANOXML (`http://nanoxml.cyberelf.be/`) is an XML-parsing library that consists of approximately 6KLOC. We selected NANOXML because it is small yet not trivial, and lets us evaluate the technique and inspect the results in detail. ANTLR (`http://www.antlr.org/`) is a widely-used language-independent lexer and parser generator that consists of 46KLOC. ANTLR was selected because it is a relatively large and complex software that can provide more confidence in the generality of the results. NANOXML inputs a file containing an XML document, and ANTLR inputs a file containing the grammar of a language. We changed both applications so that they input an array of symbolic characters *arr* instead of reading from a file. We then ran STINGER and specified *arr* as the only element in the initial set of symbolic entities. STINGER produced, for each program, a report and a instrumented version of the program.

	Metrics	NanoXML	Antlr
RQ1	Number of problematic operations	48	82
	Number of methods with problematic operations	10	23
RQ2	Number of native calls that may be reached by symbolic values	3	8
	Total number of native calls	27	48
RQ3	Number of methods transformed	89	253
	Number of reachable methods	438	1176
	Number of statements transformed	1253	4052
	Number of statements in all transformed methods	2642	8547

Figure 12: Empirical results.

In the study, we investigated three research questions:

RQ1: How effective is our technique in identifying parts of the code responsible for constraints that cannot be handled by a constraint solver that is used by the symbolic-execution system?

RQ2: How often do symbolic values flow outside the boundaries of the program being symbolically executed? When that happens, can our analysis correctly identify and report problematic cases beforehand?

RQ3: To what extent can the use of our analysis reduce the instrumentation needed to perform symbolic execution?

To address our research questions, we ran stinger on the subjects, and measured several metrics as shown in Fig. 12. In the figure, the number of methods includes both methods of the application and methods in the Java standard library, which may also need to be transformed when symbolically executing a program. We first discuss the results for each research question independently, and then discuss the precision of the analysis.

RQ1. stinger finds 48 (for NanoXML) and 82 (for Antlr) cases that would be problematic for a constraint solver like Yices [24]. In this context, the problematic cases are those that involve bit-wise and modulo operations over symbolic values. These problematic cases reside in 10 and 23 methods of NanoXML and Antlr, respectively. These cases would be reported to the user, who would then need to modify the methods (or replace them with stubs) to eliminate the problem. After inspecting stinger's report, we found that many of these problematic constraints arise because of the use of modulo operators in

classes `HashMap` and `HashTable`. Replacing these classes with another implementation of a map, such as `TreeMap`, eliminates the problem. The remaining problematic methods were methods operating on characters (e.g., to change a character from upper to lower case). We were able to rewrite these methods and eliminate the use of bit-wise operators in them by assuming that the input characters are ASCII characters.

RQ2. For the two subject programs, the only instances of symbolic values that may flow outside the boundaries of the program consist of calls to native methods. STINGER determines that for 3 of the 27 (for NANOXML) and for 8 of the 48 (for ANTLR) calls to native methods, a symbolic value may actually be passed as a parameter, either through a primitive value or as a field of an object. Based on these results, we first observe that, for the two (real) applications considered, symbolic values may indeed cross the program boundaries and create problems for symbolic execution. We also observe that the technique is successful in identifying such problematic cases and in identifying methods that, although potentially problematic, are guaranteed to never be actually reached by symbolic values. For NANOXML and ANTLR, the analysis lets users focus their attention on only 15% of the potentially-problematic calls.

RQ3. The analysis discovers that symbolic values are confined within approximately one fifth of the total number of methods for both subjects. Furthermore, within methods that may handle symbolic values, less than half of the statements are actually affected by these values. STINGER is therefore able to instrument the program so that half of the statements can be executed without incurring any overhead due to symbolic execution.

Precision. The analysis is conservative and can be imprecise in some cases (i.e., it may conclude that a variable may store symbolic values even if it never does so in reality). Although context-sensitivity increases the precision significantly, the underlying points-to analysis does not scale well for the subjects, and STINGER can thus produce imprecise results. For example, for NANOXML, we found that many standard library classes are unnecessarily instrumented because of the imprecision of the analysis. This imprecision could be reduced by using a demand-driven, highly-precise points-to analysis.

3.4 Related Work

The work that is most closely related to this work is a technique that was recently presented by Xiao et al. [83]. That technique identifies and reports problems that prevent a symbolic-execution based test-input generation tool from achieving high structural code coverage. Such a tool may fail to achieve high coverage if only parts of a program are symbolically executed, and that leads to imprecise path constraints. The main difference between the two works is that our technique involves conservative static analysis, while the other technique is based on dynamic analysis. On one hand, it is possible that the other technique may fail to identify all sources of imprecision. On the other hand, our technique can generate false positives.

This work is also related to prior works that provide tool support for abstraction in model checking (e.g., [21, 25]). In Reference [21], type inference is used to identify a set of variables that can be removed from a program when building a model for model checking. In Referece [25], a framework for type inference and subsequent program transformation is proposed. In both approaches, the type-inference algorithm used is not as precise as the type-dependence analysis. Precision is crucial for the goal of reducing manual intervention and reducing the transformations that must be performed. However, unlike this work, where only one kind of abstraction (concrete to symbolic) is supported, the framework in Reference [25] allows multiple user-defined abstractions.

Finally, the type-dependence analysis bears similarity to other approaches based on flow analysis, such as taint analysis [71] and information-flow analysis [80]. The demand-driven formulation of type-dependence analysis is similar to the formulation of points-to analysis in [73], and the cloning-based approach to interprocedural analysis and use of binary decision diagrams to make context-sensitive analysis scale were studied in [81] and [6], respectively.

3.5 Summary

This chapter presents type-dependence analysis that addresses two problems—path-divergence and complex-constraint—that hinder the application of symbolic execution to real-world software: (1) the generation of constraints that the underlying constraint solver cannot solve

and (2) the flow of symbolic values outside the program boundary. The type-dependence analysis automatically and accurately identifies places in the program where these two problems occur. The results of the analysis can be used to help users address the identified problems. The chapter also presents a program-instrumentation technique that leverages the analysis results to selectively instrument applications into applications that can be symbolically executed. We have implemented the analysis and transformation techniques in a tool, STINGER. In our empirical evaluation, we applied STINGER to two Java applications. The results show that the problems that we target do occur in real-world applications, at least for the subjects considered, and that our analysis can identify these problems automatically and help users to address them. Moreover, we show that our analysis is precise enough to allow for transforming only the part of the code actually affected by symbolic values at runtime.

CHAPTER IV

HEAP-CLONING: ENABLING PRECISE AND EFFICIENT DYNAMIC SYMBOLIC EXECUTION

Real-world programs typically contain some problematic program parts (e.g., native methods in Java) that cannot be symbolically executed. Dynamic symbolic execution (DSE) (described in Section 2.1) is a technique that does not require any manual effort even in the presence of such problematic parts, and at the same time, it also does not completely ignore the side effects of those problematic parts. However, DSE suffers from two problems: the path-divergence problem and the problem that arises in implementation of an efficient DSE system for Java, and that, exacerbates the path-explosion problem. This chapter presents a technique that addresses those two problems.

First, DSE may compute imprecise path constraints that may result in path divergence (see Section 2.1.1). Existing approaches for implementing DSE [36, 70] do not address this problem. In general, manually-specified models, which model effects of problematic parts, are necessary for eliminating such imprecision. However, an automatic method for identifying problematic parts of the program whose side effects may introduce imprecision, and thus, need to be modeled, can significantly reduce the required manual effort.

Second, implementing a DSE system for Java that can symbolically execute each path efficiently and compute precise path constraints is challenging. Both efficiency and precision are important. A system that can efficiently symbolically execute each path can symbolically execute more paths within a time budget, and thus, can alleviate the path-explosion problem. A system that can compute precise path constraints can alleviate the path-divergence problem. Recall that there are two approaches—custom-interpreter-based and instrumentation-based—for implementing symbolic-execution systems for Java. The instrumentation-based approach is significantly more efficient because instrumented programs can be executed on standard JVM's, which are orders of magnitude faster than

custom interpreters. However, in the instrumentation-based approach, the inability to instrument some of Java's standard library classes (described in Section 2.1.2.2) results in more problematic code that cannot be symbolically executed, in addition to the code that are typically problematic (e.g., native methods). This increased size of problematic code exacerbates the above-mentioned imprecision problem of the DSE technique.

To address the two above-mentioned problems—imprecision of path constraints, implementation difficulty— of DSE, we developed a novel program-transformation technique, which we call *heap cloning*. The heap-cloning technique transforms the subject program P such that during the execution of the transformed program, the heap consists of two partitions: a "concrete heap" consisting of instances of the original classes of P, and a "symbolic heap" consisting of instances of copies of the original classes, which heap cloning generates. Only methods that are not symbolically executed such as native methods and class initializers access the "concrete heap", whereas all Java methods access the "symbolic heap." Through such restricted access to the two heap partitions and a mechanism to maintain consistency between the partitions, heap cloning can in most cases determine the side effects of parts of a program that are not symbolically executed. To determine the side effects, heap cloning compares the value of a memory location in the "symbolic heap" with the value of the corresponding location in the "concrete heap."

Heap cloning provides two benefits. First, using heap cloning, a DSE system can help the user identify only the subset of all methods that are not symbolically executed, but whose side effects may cause imprecision in path constraints. As a result, the required manual effort is reduced–a user is required to examine and provide models for only a subset of methods that are not symbolically executed. Second, heap cloning enables implementation of an efficient and precise DSE system for Java, using the instrumentation-based approach. Heap cloning achieves the effect of instrumenting all standard library classes of Java by instrumenting copies of the library classes. As a result, the additional imprecision that arises in the instrumentation-based approach is eliminated.

This chapter also presents CINGER (Concolic INput GENeratoR), a system we developed

that implements heap cloning and DSE , along with the results of empirical studies we performed using CINGER to evaluate the effectiveness of the heap-cloning technique. CINGER uses the program-instrumentation based approach to symbolic execution described in Section 2.1.2.2, and leverages the optimized instrumentation technique described in Section 3.2, that aims to improve the efficiency of computing path constraints.

We performed three studies on five real-world example programs that extensively use Java's standard library classes and other language features, such as reflection and native methods. The first study shows that heap cloning can reduce the manual effort required to identify native methods that CINGER cannot symbolically execute, but that possibly introduce imprecision. The second study shows that by leveraging heap cloning, the additional imprecision that arises in the instrumentation approach can be eliminated. The third study shows that by leveraging heap cloning and using instrumentation-based approach, each program path can be symbolically executed significantly more efficiently compared to the custom-interpreter-based approach.

The main contributions of the work presented in this chapter are

- A novel program-transformation technique, heap cloning, that (1) reduces the manual effort required to write models for problematic parts of a program and (2) enables precise and efficient symbolic execution of each program path.

- A practical DSE system, CINGER that leverages the heap-cloning technique and the optimized instrumentation approach described in Chapter 3.2.

- An empirical evaluation of the technique that demonstrates that it is possible to efficiently compute precise path constraints for a set of real-world programs, which may use native methods and Java's standard library classes, with little (if any) manual effort.

4.1 Heap-cloning Technique

This section first presents the notation that is used, and then it presents the heap-cloning technique. Throughout the section, P is used to represent the original program and P' to

Object class

$[\![\texttt{class Object } \{\bullet\ \texttt{K};\ \overline{\texttt{M}}\}]\!] = \texttt{interface Object}^{\texttt{I}}_{\texttt{HC}}$ (1)

$\texttt{class Object}_{\texttt{HC}} \texttt{ implements Object}^{\texttt{I}}_{\texttt{HC}}\{$ (2)

$\quad \texttt{Object shadow;}$

$= \quad [\![\texttt{K}]\!]\ ;$

$\quad \overline{[\![\texttt{M}]\!]}$

$\}$

Other classes

$\texttt{class C}_{\texttt{HC}} \texttt{ extends D}_{\texttt{HC}} \texttt{ implements } \overline{\texttt{I}_{\texttt{HC}}}\ \{$ (3)

$\quad \overline{\texttt{E}_{\texttt{HC}}}\ \overline{\texttt{f}_{\texttt{HC}}}\texttt{;}$ (4)

$[\![\texttt{class C extends D implements } \overline{\texttt{I}}\{\ \overline{\texttt{E}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{K}}\texttt{;}\ \overline{\texttt{M}}\ \}]\!]$

$= \quad \overline{[\![\texttt{K}]\!]}\texttt{,}$

$\quad \texttt{C}_{\texttt{HC}}()\{\texttt{r = new C; r.C(null); this.shadow = r;}\}\texttt{,}$ (5)

$\quad \texttt{C}_{\texttt{HC}}\texttt{(C s)}\{\texttt{this.shadow = s;}\}\texttt{;}$ (6)

$\quad \overline{[\![\texttt{M}]\!]}$ (7)

$\}$ (8)

$\texttt{class C extends D implements } \overline{\texttt{I}}\texttt{,Wrapper;}\{$ (9)

$\quad \overline{\texttt{E}}\ \overline{\texttt{f}}\texttt{;}$ (10)

$\quad \overline{\texttt{K}}\texttt{,}$

$= \quad \texttt{C(Dummy}_{\texttt{HC}}\texttt{ d)}\{\texttt{super(d);}\}\texttt{;}$ (11)

$\quad \overline{\texttt{M}}\texttt{,}$

$\quad \texttt{C}_{\texttt{HC}}\texttt{ wrap()}\{\texttt{r = new C}_{\texttt{HC}}\texttt{; r.C}_{\texttt{HC}}\texttt{(this); return r;}\}$ (12)

$\}$ (13)

Constructor

$[\![\texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{x}})\{\overline{\texttt{S}}\}]\!] = \texttt{void C}^{\texttt{K}}_{\texttt{HC}}\texttt{(C}_{\texttt{HC}}\texttt{ y, } \overline{\texttt{D}_{\texttt{HC}}}\ \overline{\texttt{x}})\{\overline{[\![\texttt{S}]\!]}\}$ (14)

Non-native method

$[\![\texttt{D m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\overline{\texttt{S}}\}]\!] = \texttt{D}_{\texttt{HC}}\texttt{ m}_{\texttt{HC}}(\overline{\texttt{C}_{\texttt{HC}}}\ \overline{\texttt{x}})\{\overline{[\![\texttt{S}]\!]}\}$ (15)

Native method

$\texttt{D}_{\texttt{HC}}\texttt{ m}_{\texttt{HC}}(\overline{\texttt{C}_{\texttt{HC}}}\ \overline{\texttt{x}})\{$ (16)

$[\![\texttt{native D m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\texttt{;}]\!] = \quad \texttt{D r} = \texttt{m}(\overline{\texttt{shadow}(\texttt{x})})\texttt{;}$ (17)

$\quad \texttt{return r.wrap();}$ (18)

$\}$

Figure 13: Heap-cloning rules for transforming classes and methods. The numbers in the parentheses shown next to the rules are used too refer to corresponding lines.

represent the transformed program produced by applying heap cloning to P.

Figure 13 and Figure 14 present the transformation rules of the heap-cloning technique. Figure 13 presents rules for transforming classes and methods, whereas Figure 14 presents rules for transforming program statements. Our notation is inspired by that used in the specification of Featherweight Java [41]. Here, rules for only a representative subset of the

Allocation			
$[\![\mathtt{v} = \mathtt{new\ C}]\!]$	=	$\mathtt{v} = \mathtt{new}\ \mathtt{C_{HC}}$	(19)
Constructor invocation			
$[\![\mathtt{v.C}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})]\!]$	=	$\mathtt{C^K_{HC}}(\mathtt{C_{HC}}\ \mathtt{v}, \overline{\mathtt{D_{HC}}}\ \overline{\mathtt{x}})$	(20)
Non-static method invocation			
$[\![\mathtt{v.m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})]\!]$	=	$\mathtt{v.m_{HC}}(\overline{\mathtt{D_{HC}}}\ \overline{\mathtt{x}})$	(21)
Static method invocation			
$[\![\mathtt{C.m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})]\!]$	=	$\mathtt{C_{HC}.m_{HC}}(\overline{\mathtt{D_{HC}}}\ \overline{\mathtt{x}})$	(22)
Store			
$[\![\mathtt{v_1.f} = \mathtt{v_2}]\!]$	=	$\mathtt{v_1.f_{HC}} = \mathtt{v_2};$	(23)
		$\sigma(\mathtt{v_1})\mathtt{.f} = \pi(\mathtt{v_2});$	(24)
Load			
$[\![\mathtt{v_1} = \mathtt{v_2.f}]\!]$	=	$\mathtt{v} = \mathtt{v_2.f_{HC}};$	(25)
		$\mathtt{v_s} = \sigma(\mathtt{v_2})\mathtt{.f};$	(26)
		$\mathtt{if(v == v_s)\ goto\ l};$	(27)
		$\mathtt{v_2.f_{HC}} = \mathtt{v_s.wrap()};$	(28)
		$\mathtt{l} : \mathtt{v_1} = \mathtt{v_2.f_{HC}};$	(29)
Assignment			
$[\![\mathtt{v_1} = \mathtt{v_2}]\!]$	=	$\mathtt{v_1} = \mathtt{v_2}$	(30)
$[\![\mathtt{v} = \mathtt{null}]\!]$	=	$\mathtt{v} = \mathtt{null}$	(31)
Reference equality			
$[\![\ \mathtt{if(v_1 == v_2)\ goto\ l}\ ]\!]$	=	$\mathtt{if}(\pi(\mathtt{v_1}) == \pi(\mathtt{v_2}))\ \mathtt{goto\ l}$	(32) (33)
Cast			
$[\![\mathtt{v_1} = (\mathtt{C})\ \mathtt{v_2}]\!]$	=	$\mathtt{v_1} = (\mathtt{C_{HC}})\ \mathtt{v_2}$	(34)

Auxiliary functions:

$$\sigma(\mathtt{v}) = \begin{cases} \mathtt{null} & \mathtt{v} == \mathtt{null} \\ \mathtt{v} & \mathtt{otherwise} \end{cases} \qquad \pi(\mathtt{v}) = \begin{cases} \sigma(\mathtt{v}) & \mathtt{v\ is\ of\ reference\ type} \\ \mathtt{v} & \mathtt{otherwise} \end{cases}$$

Figure 14: Heap-cloning rules to transform program statements. The numbers in the parentheses shown next to the rules are used too refer to corresponding lines.

Java language that is relevant to heap cloning are presented.

$\overline{\mathtt{C}}$ is the shorthand for a possibly empty sequence $\mathtt{C}_1, \ldots, \mathtt{C}_n$, and similarly for $\overline{\mathtt{F}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{S}}$, etc. The empty sequence is written as $\bullet$ and concatenation of sequences is denoted using a comma. Operations on pairs of sequences are abbreviated by writing "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$" for "$\mathtt{C}_1\mathtt{f}_1, \ldots, \mathtt{C}_n\mathtt{f}_n$". The meta-variables `C` and `D` range over class names; `f` ranges over field names; `m` ranges over method names; `x` and `v` ranges over variables.

The class declaration "`class C extends D implements` $\mathtt{I}_1, \mathtt{I}_2$ $\{\overline{\mathtt{D}}\ \overline{\mathtt{f}}\mathtt{;}\ \mathtt{K}\mathtt{;}\ \overline{\mathtt{M}}\}$" represents

a class named C. D is C's superclass, and C implements interfaces $\texttt{I}_1$ and $\texttt{I}_2$. The class has fields $\overline{\texttt{f}}$ with types $\overline{\texttt{D}}$, a single constructor K, and a suite of methods $\overline{\texttt{M}}$. The method declaration "D m($\overline{\texttt{C}}$ $\overline{\texttt{x}}$){ return e; }" introduces a method named m with result type D and parameters $\overline{\texttt{x}}$ of types $\overline{\texttt{C}}$. The body of the method is the single statement return e;. this represents a special variable that store the reference to the receiver of a non-static method call.

In rest of this section, the transformation rules in Figure 13 and Figure 14 are referred to by the numbers shown in parentheses next to the rules. Those numbers are shown as superscripts in parentheses (e.g., text$^{(100)}$) in the text describing the rule.

4.1.1 Creating New Classes

The heap-cloning technique creates a new class (interface) $\texttt{A}_{\texttt{HC}}$, which we call the *Heap Cloning* (abbreviated *HC*) class, corresponding to every class (interface) A of the original program $P^{(3-8)}$. Like other regular classes, the heap-cloning technique also creates HC classes corresponding to Java's built-in array classes. For example, a regular class $\texttt{char[]}_{\texttt{HC}}$ is created corresponding to a one-dimensional array of characters. In rest of this chapter, we use the following convention: names of HC classes have HC as subscripts, and the HC class corresponding to an original class A is named $\texttt{A}_{\texttt{HC}}$.

Heap cloning aims to minimize the dissimilarity between the class hierarchy consisting of original classes and interfaces of P and the hierarchy consisting of HC classes and interfaces. The similarity between the two hierarchies facilitates type-safe transformation of the program. The inheritance relationship of HC classes and interfaces is based on the two rules$^{(3)}$: (1) if a class A is a subclass of B, $\texttt{A}_{\texttt{HC}}$ is a subclass of $\texttt{A}_{\texttt{HC}}$; (2) for each interface I that a class A implements, $\texttt{A}_{\texttt{HC}}$ implements the interface $\texttt{I}_{\texttt{HC}}$.

Figure 15 shows an example of the way in which the heap-cloning technique transforms the class hierarchy of a program. Figure 15(a) shows the original class hierarchy, consisting of a class A, the built-in character array class char[], and the class Object. Figure 15(b) shows the class hierarchy consisting of the generated HC classes. A solid arrow from a class A to a class B means that class A extends class B. A dotted arrow from a class or interface A

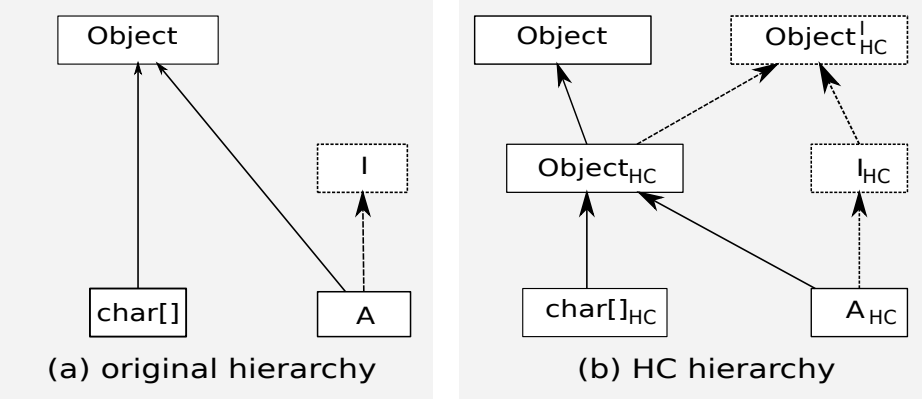


Figure 15: Class hierarchy before and after the heap-cloning transformation.

to an interface `B` means that `A` implements `B`. $\texttt{Object}^{\texttt{I}}_{\texttt{HC}}$ is a special interface introduced by heap cloning. Every HC interface and the $\texttt{Object}_{\texttt{HC}}$ implement this special interface(2).

For every field, `f`, of an original class `A`, heap cloning adds a field $\texttt{f}_{\texttt{HC}}$ to $\texttt{A}_{\texttt{HC}}$(4). If the type of `f` corresponds to class `X`, then $\texttt{f}_{\texttt{HC}}$ has type of class $\texttt{X}_{\texttt{HC}}$(4). For each method `m` of an original class `A`, the heap-cloning technique adds a method $\texttt{m}_{\texttt{HC}}$ to $\texttt{A}_{\texttt{HC}}$(14,15,16). If `m` has a method body, then `m`'s body is copied into $\texttt{m}_{\texttt{HC}}$(15). If `m` is a native method, then $\texttt{m}_{\texttt{HC}}$ delegates the calls to `m`(16–18). The next section discusses this delegation.

Heap cloning transforms every invocation statement in the HC classes so that it will invoke methods of the HC classes. Thus, when the transformed program is executed, Java methods of the HC classes, and only native methods and static class initializers of the original classes, are executed.

4.1.2 Concrete and Symbolic Heap Partitions

When the transformed program P' is executed, its heap consists of two paritions: concrete heap partition and symbolic heap partition.

Definition 9 (Shadow Object)**.** *For every object of a class* `A` *contained in the heap of* P *at some program point during the execution of* P*, the heap of* P' *contains one object* o_s *of class* `A`*. The heap of* P' *also contains at least one object* o *of class* $\texttt{A}_{\texttt{HC}}$ *at the corresponding program point in* P'*'s execution.* o_s *is referred to as the* shadow object *of* o*.*

Definition 10 (Concrete Heap Partition)**.** *The partition of the heap during P''s execution that consists of objects of the original class (i.e., shadow objects) is referred to as the* concrete heap partition*.*

Definition 11 (Symbolic Heap Partition)**.** *The partition of the heap that consists of objects of HC classes is referred to as the* symbolic heap partition*.*

During P''s execution, for an object o and its field $\texttt{f}_{\texttt{HC}}$, the value of the heap location $o.\texttt{f}_{\texttt{HC}}$ remains consistent with the value of the heap location $o_s.\texttt{f}$, where o_s is the shadow object of o and f is the shadow field of $\texttt{f}_{\texttt{HC}}$.

Definition 12 (Consistent)**.** *For an object o and its shadow object o_s, the value, v of $o.\texttt{f}_{\texttt{HC}}$ is* consistent *with the value v_s of $o_s.\texttt{f}$ if (1)* f *is of primitive type and $v = v_s$, or (2)* f *is of non-primitive type and v_s is shadow object corresponding to v.*

Recall that during P''s execution, some code, such as native methods and class initializers, of the original classes of P get executed. Because the two heap partitions are kept consistent, that code can read from and write to the concrete heap partition. Heap cloning handles native methods as follows. If m is a native method, then $\texttt{m}_{\texttt{HC}}$ delegates the calls to m with arguments that are the shadow objects corresponding to the HC objects that it receives as parameters. Also, if m returns a heap object, $\texttt{m}_{\texttt{HC}}$ wraps it as an object of corresponding HC class, and returns the wrapped object. The wrapping operation is discussed later in this section. In contrast to native methods, handling class initializers does not require any method delegation because those methods are implicitly invoked by a JVM.

Heap cloning adds a special field shadow to $\texttt{Object}_{\texttt{HC}}$ that stores the shadow object corresponding to an object of a HC class$^{(2)}$. The object o stores a reference to its shadow object o_s in a designated field shadow. The program is transformed such that when an object o of a HC class $\texttt{A}_{\texttt{HC}}$ is allocated, a shadow object o_s of the corresponding original class A is also allocated, and is stored in the special field shadow of $o^{(5)}$. To allocate the shadow object, a new constructor is added to the original class $\texttt{A}^{(11)}$.

To illustrate, consider program P shown in Figure 2. P allocates an object of class L (line 8) and an object of class A (line 12). Figure 16(a) shows the snapshot of P's heap

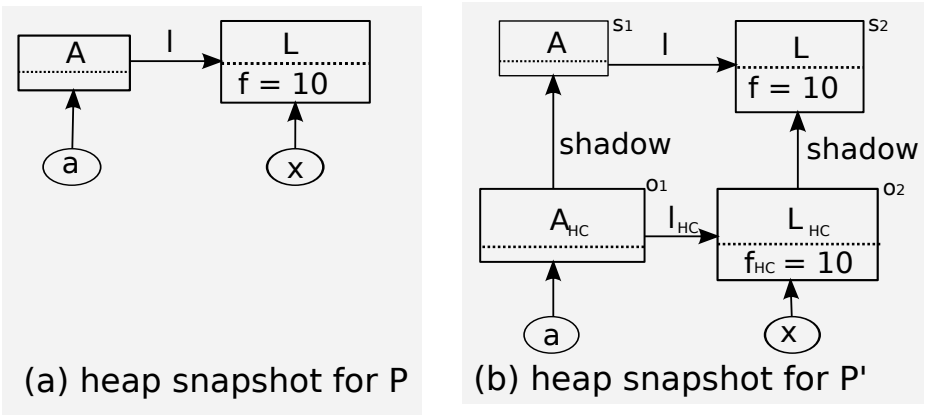


Figure 16: Snapshots of the heaps at line 15 of (a) the original program P (Figure 2) and (b) the transformed program P', when the programs are executed with input `10`.

just before execution of line 15, when P is executed with input `10`. In Figure 16, each rectangle (shown with a dotted divider) represents a heap object. The label (if there is one) shown near the upper right corner of each rectangle refers to the object that it represents. Ovals represent program variables, and arrows emanating from ovals represent values of reference-type variables. For an object, the name of the class of the object is shown above the divider, and values of the primitive-type fields (if any) of the object are shown below the divider. The value of a reference-type field of an object is represented by an arrow labeled with the field name and originating from the rectangle representing the object.

Figure 16(b) shows the snapshot of P''s heap just before execution of line 15, when P' is executed with input `10`. For every object that exists in P's heap, P''s heap contains two objects: one object o of a HC class and the other object of the original class that is the shadow object of o. For example, objects s_1 and s_2 are the shadow objects of o_1 and o_2, respectively. Note that the value of the heap location $o_2.\texttt{f}_{\texttt{HC}}$ (i.e., `10`) is consistent with the value of $s_2.\texttt{f}$ (i.e., `10`) because the field `f` is of `int` type, and the two values are identical. Similarly, the value (i.e., the reference to o_2) of $o_1.\texttt{l}_{\texttt{HC}}$ is consistent with the value (i.e., the reference to s_2) of $s_1.\texttt{l}$ because the field `l` is of reference type and s_2 is the shadow object of o_2.

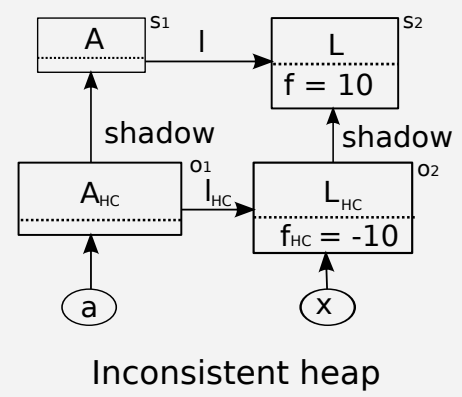


Figure 17: Inconsistent heap that will arise without heap cloning.

4.1.3 Handling Side Effects of Uninstrumented Code

Side effects of uninstrumented code (e.g., native methods) manifest as inconsistencies between the concrete and symbolic heap partitions. In heap cloning, two values—one in concrete partition and another in symbolic partition—are stored corresponding to every heap location. Instrumented Java methods update values corresponding to a heap location in both partitions. In contrast, uninstrumented code updates the value corresponding to a heap location only in the concrete partition. As a result, any inconsistency in values of a heap location in the two partitions is an indication that the heap location was updated inside uninstrumented code. Such inconsistency is detected when the value of the concerned heap location is later read after execution of the problematic code. If such an inconsistency is detected and the concerned heap location is mapped to a symbolic value at the time of reading, a symbolic execution system that leverages heap cloning can notify the user that an imprecision might have been introduced.

To illustrate, recall that the native method `complex`, which is invoked in line 16 of Figure 2, reads the current value of the field `f`, and stores its corresponding absolute value back into the field `f`. Suppose the transformed program is executed with input `-10`. Figure 17 shows the potential heap snapshot of the transformed program at line 17 in this execution. In the figure, note that the value of $o_2.\mathtt{f_{HC}}$ is not consistent with the value of $s_2.\mathtt{f}$ because, $s_2.\mathtt{f}$ is updated inside the `complex` native method.

If consistency between two heap partitions is not maintained, the transformed program

produced by the heap-cloning technique may behave differently from the original program for some inputs. For example, in this program, method `error` will never be called because at line 17, the value of the heap location `x.f` is always non-negative. However, in the transformed program produced by heap cloning, if the above-mentioned inconsistency is not eliminated, the `error` method will be called because at line 17, the value of the heap location $o_2.\mathtt{f_{HC}}$ (i.e., -10) will be used to decide which branch is taken. The heap-cloning technique eliminates the potential inconsistency in this example, by copying the value of $s_2.\mathtt{f}$ (i.e., 10) in the heap location $o_2.\mathtt{f_{HC}}$, before the branch decision is made. Following paragraphs present the transformation rules concerning load and store statements. Those rules ensure consistency between the two heap partitions.

In P', an update of a non-static field $\mathtt{f_{HC}}$ of an object o to a new value v, is accompanied by an update to heap location $o_s.\mathtt{f}$, where o_s is the shadow object corresponding to $o^{(23,24)}$. If the field `f` is of primitive type, values of both heap locations, $o.\mathtt{f_{HC}}$ and $o_s.\mathtt{f}$ are updated to the new value v. If `f` is of reference type, then the values of $o.\mathtt{f_{HC}}$ and $o_s.\mathtt{f}$ are updated to v and v_s, respectively, where v_s is shadow object corresponding to v.

The transformation rule for a load statement handles cases in which a field value in the concrete heap partition is changed outside the methods of HC classes (e.g., native methods). In P', a load from field $\mathtt{f_{HC}}$ of an object o involves three steps:

1. Read the current values v and v_s of both heap locations $o.\mathtt{f_{HC}}$ and $o_s.\mathtt{f}$, respectively$^{(25,26)}$.

2. If v and v_s are consistent, return $v^{(27)}$.

3. If v and v_s are inconsistent, update the value of the heap location, $o.\mathtt{f_{HC}}$ to a value v_{new} that is consistent with ${v_s}^{(28)}$, and return v_{new}. For primitive type values, v_{new} is v_s. For reference type values, v_{new} is obtained from wrapping v_s.

4.1.4 Wrapping Concrete Objects

In two cases, it is necessary to wrap objects of original classes as objects of HC classes. The first case concerns values returned from native methods. For example, a native method `m` of a class `X` may return an object of a class `A`. The method $\mathtt{m_{HC}}$ in $\mathtt{X_{HC}}$ that corresponds to `m`,

delegates the call to `m`, and thus, must wrap the object that `m` returns as an object of $\texttt{A}_{\texttt{HC}}$. The second case concerns situations where a field value of a heap object is updated outside the methods of the HC classes (e.g., native methods).

To support the wrapping operation, the technique performs two transformations.

- Every original class, except `Object`, is transformed to implement an interface `Wrapper`, which defines only one method `wrap`[9]. The implementation of `wrap` in an original class `A`, wraps the receiver object as an object of the corresponding HC class $\texttt{A}_{\texttt{HC}}$[12].
- A special constructor is added to every HC class $\texttt{A}_{\texttt{HC}}$ that is used in the `wrap` implementation of the original class `A`[6].

Note that, as a result of wrapping an object of the original class as described above, one object of an original class `A` can be the shadow object of multiple objects of the HC class $\texttt{A}_{\texttt{HC}}$. This may cause the transformed program to behave differently from the original program. Specifically, the difference in behavior may arise because of program operations, such as reference equality check, use of the `hashcode` method of the `Object` class, and use of monitors for thread synchronization that explicitly or implicitly compares identities of two objects. However, the heap-cloning technique eliminates this potential problem. In P', shadow objects as operands to these problematic operations, instead of objects of HC classes[32].

4.1.5 Eliminating Imprecision in the Instrumentation Approach

In the instrumentation approach, the inability to flexibly instrument Java's standard library classes leads to imprecision. However, this inability to instrument library classes does not lead to any imprecision while symbolically executing the transformed program P' produced by heap cloning because P' does not use the original library classes. Instead, with a few exceptions, P' uses the HC classes corresponding to the library classes. As a result, symbolic values cannot flow into original library classes in the transformed program, and thus, it is not necessary to instrument library classes for symbolic execution. In the exceptional cases, native methods and static class initializers of the library classes are executed, and thus,

can potentially lead to computation of imprecise path constraints. This issue of precision is further explained in Section 4.1.6.

Heap cloning instruments the original library classes to maintain consistency between two heap partitions. There are several types of minor instrumentation that are applied to these classes: (1) addition of getter and setter methods to read and update fields of those classes, (2) addition of a constructor[11], (3) making each of those classes (except `Object`) implement the interface Wrapper[9] and the addition of the method `wrap()`[11]. However, such instrumentation of original library classes is not fundamental to heap cloning—it can be avoided using alternative solutions that use reflection or low-level API classes (e.g., `sun.misc.Unsafe`). Compared to those alternative solutions, instrumentation provides efficiency and portability across JVM's, and at the same time, in our experience, such instrumentations have not caused any problem in the JVM's operation.

4.1.6 Requirement of Manual Effort

One issue related to the precision of path constraints is whether heap cloning can always automatically compute precise path constraints. Heap cloning can automatically compute precise path constraints of a path only if the precision of the path constraint is not affected by side-effects of methods (e.g., native methods) that are only concretely executed. If imprecision is introduced because some of those methods update memory locations that have symbolic values corresponding to them, heap cloning detects, and notifies the user of such updates. Such notification can help the user to identify the problematic methods. To ensure computation of precise path constraints, the user can then provide models of those methods such that those models update the symbolic values of updated memory locations.

4.2 Empirical Evaluation

This section first discusses our implementation of CINGER, then it describes our subject programs, and finally, it presents the results of our three studies.

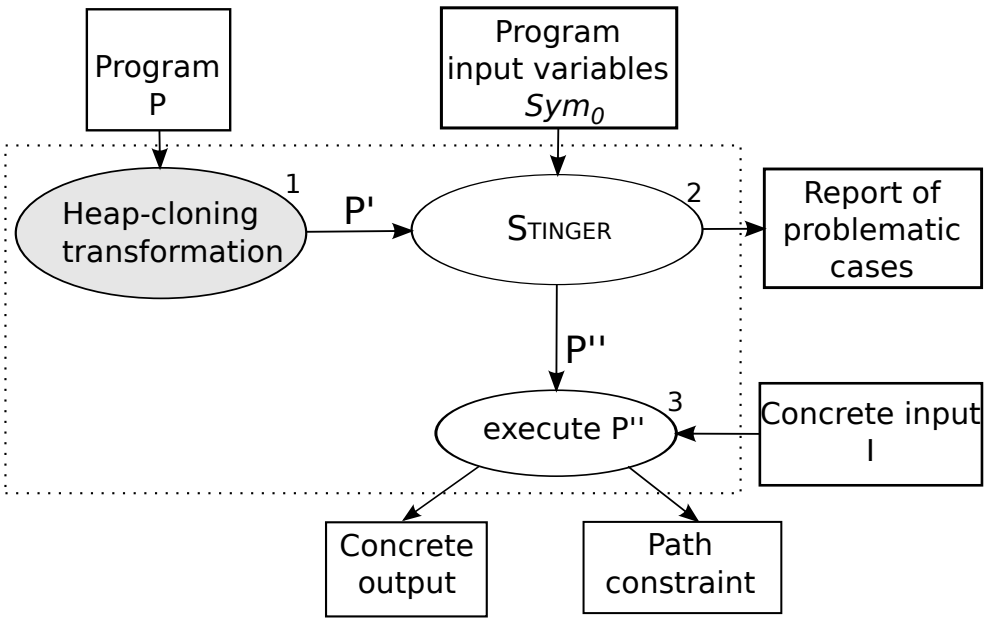


Figure 18: The CINGER dynamic symbolic-execution system.

4.2.1 Implementation

CINGER contains approximately 10,000 lines of Java code, and uses the Soot framework [78]. To compute path constraints, CINGER transforms all types of Java constructs and operators, such as bitwise operations, array accesses, arrays with symbolic length, and type conversion of primitive-type values.

Figure 18 shows a dataflow diagram of CINGER. CINGER inputs a program P and a concrete input (i.e., non-symbolic program input) I. CINGER generates P'' that is a transformed version of P and produces a report of (1) problematic parts of P that may generate complex constraints that the underlying constraint solver cannot solve, and (2) parts of P where, symbolic values that may flow into parts of programs that are not symbolically executed (e.g., native methods in Java). If P has some problematic parts, the user may choose to examine those parts and provide models for or rewrite those parts and finally, run CINGER again. CINGER executes P'' with input I on a standard JVM . Execution of P'' produces (1) the same output as the output produced by executing the original program P with input I and (2) the path constraint of the path that P takes for I.

Transformation of P to P'' involves two program-transformation steps. In Step 1, the

Heap-cloning transformation technique transforms P to P'. Recall that P' consists of original classes of P along with HC classes produced by the heap-cloning technique. In Step 2, only the HC classes of P' are instrumented using STINGER to produce the final transformed program P''. Recall that STINGER, which is described in Section 3.3.1, uses the type-dependence analysis and instrumentation technique described in Section 3.2.

4.2.2 Studies

As subjects for our studies, we used six publicly available Java programs: NanoXML, JLex, Sat4J (2 versions), BCEL, and Lucene. NanoXML is a light-weight parser for XML that inputs an XML file and an optional Document Type Definition (DTD) file. JLex is a lexical analyzer generator for Java that inputs a JLex specification file. Sat4J is a satisfiability solver that inputs a formula that can be different formats such as conjunctive normal form or constraint satisfaction problem. BCEL is a library for analyzing and manipulating Java class files that inputs a set of Java class files. Lucene is a text-search engine that inputs a set of text files.

For each subject, we created a test suite consisting of test cases that we gathered from the web or created manually. Table 1 provides details about the subjects. The first column lists the name of the subject. For each subject, the second column shows the number of methods that were covered by the test suite. The third and fourth columns show the number of classes containing at least one covered method and the number of lines of code corresponding to all covered methods, respectively. Additionally, each column shows the numbers of corresponding entities belonging to user-defined (User) classes and Java's standard library classes (Library). For example, for JLex, 136 user-defined methods and 717 library methods are covered by the set of test cases. Our subjects are significantly larger than subjects used in prior works that applied symbolic execution to Java programs at the whole-program level.

4.2.2.1 Study 1: Reduction in Required Manual Effort to Specify Models

The goal of this study is to assess whether heap cloning can reduce the manual effort required to specify models for native methods that introduce imprecision.

We performed this assessment in three parts. In the first part, we estimated the number

Table 1: Subject programs for the studies.

Subject	Methods		Classes		Lines of Code	
	User	Library	User	Library	User	Library
NanoXML	87	709	15	161	1,230	14,604
JLex	136	717	27	158	6,566	13,702
Sat4J-Dimacs	292	789	45	190	3,908	17,195
Sat4J-CSP	329	1,486	51	339	4,125	39,617
BCEL	112	614	45	149	2,321	12,659
Lucene	942	1,852	215	409	20,821	56,622

Table 2: Results of Study 1.

Subject	Number of Covered Native Methods
NanoXML	4
JLex	6
Sat4J-Dimacs	0
Sat4J-CSP	9
BCEL	4
Lucene	33

of native methods that would need to be modeled if the user did not know which native methods could introduce imprecision. To do this, for each subject program we counted the number of unique native methods that are executed and have at least one reference-type parameter. We counted only native methods with reference-type parameters because performing DSE in the presence of native methods that take only primitive-type (e.g., `int`) parameters is straight-forward. The number of counted native methods is a lower limit on the number of methods for which the user may have to provide models because there could be other native methods that accessed program's heap through static fields.

Table 2 shows the results of this part. For each subject, the second column shows the number of such native methods that are executed for at least one input of the corresponding test suite. The data in the table show that each of our subjects, except Sat4J-Dimacs, calls at least one problematic native method. Thus, if the user did not know which native methods could introduce imprecision, then she would have to model each of those methods. Although the number of the problematic methods is small, it may still require significant manual effort to create models for those methods.

In the second part, we enabled CINGER to generate a notification whenever it detected

an update to a memory location such that (1) the update occurred inside uninstrumented code, and (2) that memory location was mapped to some symbolic value. Those notifications pointed to only one native method `arraycopy` in the class `java.lang.System` that performed such updates, and this meant that only this method needed to be modeled.

In the third part, we assessed whether the user must write models for one or more of the above-mentioned problematic methods if standard symbolic execution (i.e., not DSE) is performed using JPF. We chose JPF because models for a large number of native methods and other problematic classes (e.g., `java.lang.Class`) are already available for JPF . We estimated whether the user would need to write new models in addition to those that are already available. To do this, we implemented a symbolic-execution system, Jazz, on top of JPF. We applied Jazz to our subjects to compute path constraints for each path that corresponds to a test case for the subjects.

Jazz could not automatically handle all our selected subjects because of the lack of models for some native methods. For Sat4J and BCEL, we manually removed those statements of the program that called the problematic native methods—removal of those statements did not affect the functionality of the programs. As an example, for Sat4J-Dimacs and Sat4J-CSP, we removed the code that output a banner at the beginning of the application. For Lucene, we did not use this approach because of our unfamiliarity with the Lucene program and the problematic native methods (of `java.lang.management.ManagementFactory` class). Thus, we could not apply `Jazz` to Lucene.

In summary, without knowing which methods introduced imprecision, the user must write models for each method that is counted in Table 2. Even the set of models that is currently available for JPF is not sufficient for our subjects, and thus, the user must write models for additional methods. In contrast, if heap cloning identifies only one method that introduced imprecision, the user has to specify a model only for that method, which can provide a significant savings.

Table 3: Results of Study 2.

Subject	Average Number of Conjuncts	Average Percentage of of Conjuncts Generated Inside Library Classes
NanoXML	19,582	25.24%
JLex	65,068	17.47%
Sat4J-Dimacs	60,351	0.00%
Sat4J-CSP	629,078	66.90%
BCEL	34,161	89.76%
Lucene	47,248	0.00%

4.2.2.2 Study 2: Increase in Precision in Instrumentation Approach

The goal of this study is to assess the increase in precision that a symbolic-execution system based on the instrumentation approach would achieve by leveraging the heap-cloning technique. Path constraints computed using this approach can be imprecise because computed path constraints may not contain conjuncts that arise from symbolic execution of library code that may not be instrumented due to problems described in Section 2.1.2.2.

To do this assessment, we performed three steps.

1. We symbolically executed each subject program using CINGER to compute path constraints corresponding to paths that the programs took for each test case in the subject's test suite.
2. We computed the number of conjuncts in those path constraints, and computed the average over all these conjuncts.
3. We computed the percentage of conjuncts that are generated inside the library classes, and computed the average over these percentages.

Table 3 shows the results of the study. In the table, for each subject, the second column shows the average number of conjuncts in path constraints over different inputs. The third column shows the average percentage of all conjuncts of a path constraint generated inside methods of HC classes corresponding to Java's standard library classes.

The table shows that the percentage of conjuncts generated inside library classes varies widely across the subjects. For BCEL, approximately 90% of the conjuncts on a path constraint are generated inside library classes, whereas, for Sat4J-dimacs and Lucene, no

conjuncts are generated inside library classes. The number of conjuncts generated inside library classes depends on whether the library classes operate on symbolic values. Lucene and Sat4J-Dimacs use their own input processing front-ends (e.g., scanner), and thus, symbolic input values do not flow into the library classes.

In summary, the results of this study show that a significant percentage of conjuncts on path constraints are indeed generated inside Java's standard library classes, and CINGER is able compute them precisely.

4.2.2.3 Study 3: Overhead of Computing Path Constraints

The goal of this study is to compare the overhead of computing precise path constraints using the custom-interpreter-based and the instrumentation-based approaches.

For this study, we performed the following steps.

1. We configured[1] JAZZ not to use JPF 's features of JPF such as backtracking, state-matching, or partial-order reduction, that incur overhead, but that JAZZ does not use. We used the recent version of JPF that was available at the time of the study.

2. We executed the subject programs on a standard Java virtual machine normally (without symbolic execution), recorded the times for the executions, measured the average execution times over all tests of the subject's test suite, and used this time as a baseline.

3. We applied CINGER and JAZZ to perform DSE on each subject for each its test cases, computed the average of time over all test cases for each subject.

4. We computed *slowdown ratio* as the ratio of average symbolic-execution time to the baseline.

Table 4 shows the results of this study. The second column in the table shows the average execution times values (in milliseconds) corresponding to concrete executions of the original programs. The third and fourth columns show the *slowdown ratios* corresponding

[1]We used the options: "`+vm.storage.class= +vm.por=false +search.class=gov.nasa.jpf.search.Simulation`"

Table 4: Results of Study 3.

Subject	Average (concrete) Execution Time (in ms)	Slowdown for Cinger	Slowdown for Jazz
NanoXML	100	2.5	16.43
JLex	230	6.82	72.23
Sat4J-Dimacs	210	4.10	376.56
Sat4J-CSP	290	4.74	109.53
BCEL	145	3.69	21.34
Lucene	300	2.62	-

to symbolic executions of the program using CINGER and JAZZ, respectively. All execution times are inclusive of time taken by JVM to start up, garbage collect, etc. JAZZ did not terminate for two of the four inputs for Sat4J-Dimacs and one of the four inputs for Sat4J-CSP. Both CINGER and JAZZ terminated for all other inputs. The third column in Table 4 shows that, for the subjects, symbolic execution with JAZZ is about 17–377 times slower than concrete execution. In contrast, the second column shows that symbolic execution with CINGER is about 3–7 times slower than concrete execution.

The time to compute path constraints depends on the size of the input that is treated symbolically, and that in turn depend on the subjects and the specific application of symbolic execution. Computing path constraints may not always consume the greatest time in symbolic execution. The time to solve those path constraints may also require significant time. Thus, a slight improvement in efficiency in computing path constraints may not make one DSE system more scalable. However, we believe that CINGER's order-of-magnitude difference in efficiency over JAZZ suggests that CINGER can be more appropriate than other alternatives.

We also found that JAZZ inherits high-memory overhead from JPF . For some inputs for which JAZZ did not terminate, JAZZ ran out of heap space that we set to 6 GB. To assess the possibility that JAZZ (not the underlying JPF) is the source of the high overhead, we concretely executed the subjects on Java Pathfinder and measured the execution times. We found that JAZZ adds insignificant overhead for symbolic execution, which confirms previous results [42]. In summary, the results of this study show that CINGER can compute path constraints as precisely, but more efficiently than JAZZ.

4.3 Related work

The requirement to transform Java's standard library classes arises in a number of domains, such as distributed computing and software testing. Thus, a number of domain-specific techniques (e.g., [65, 75]) have been presented that address the problem by leveraging domain-specific knowledge, and thus, they are not suitable for symbolic execution. Prior work [8, 27] presents general techniques to address this problem. Our heap-cloning technique builds on the Twin Class Hierarchy (TWH) approach [27], in which copies of the classes are used instead of the original classes. However, for native methods, TWH requires manually-specified conversion routines that convert objects of original classes to objects of copy classes, and vice versa. In heap cloning, because a concrete-heap partition is kept consistent with the symbolic-heap partition, native methods can be executed on the concrete-heap partition without requiring any conversion routines. Compared to the technique presented in Reference [8], heap cloning is a more general solution (i.e., not specific to an API), and allows arbitrary instrumentation (e.g., adding fields) of library classes.

4.4 Summary

The dynamic symbolic-execution technique can automatically compute path constraints even for a program that has some parts that cannot be symbolically executed. However, the technique has two problems: (1) it may compute imprecise path constraints, and (2) implementing the technique for Java is problematic.

This chapter presents a novel program-transformation technique—heap cloning—that addresses those two problems. Heap cloning can be used in dynamic symbolic execution to identify problematic parts of a program whose side effects can introduce imprecision in path constraints. Such identification reduces the manual effort required from the user of a dynamic-symbolic-execution system to provide models for or rewrite parts of a program that cannot be symbolically executed. Also, heap cloning enables implementation of a precise and efficient DSE system the instrumentation-based approach. In this chapter, we also present a CINGER, a dynamic-symbolic-execution system that leverages heap cloning. Empirical results from using CINGER on a set of real-world programs show that heap cloning is useful.

CHAPTER V

CONTEST: SCALING SYSTEMATIC TESTING OF EVENT-DRIVEN PROGRAMS

This chapter presents a new technique to address the path-explosion problem (described in Section 1.1) that arises in symbolic execution of event-driven programs. Event-driven programs, which input and react to possibly unbounded sequence of events, are ubiquitous. Stream-processing programs, web servers, Graphical User Interfaces (GUIs), and embedded systems are all examples of event-driven programs. To systematically test event-driven programs, we address the branch-coverage problem.

> **Branch-coverage Problem:**[1] Given a constant bound $k \geq 1$, efficiently compute a set of event sequences that executes each branch of an event-driven program that can be executed by some event sequence of length up to k.

Intuitively, our solution to the above problem uses symbolic execution in a way that is similar to how symbolic execution is used to generate test inputs for other types of programs. In our solution, each event is represented by a set of symbolic values, and we generate sequences of such events up to length k by systematically discovering feasible program paths through symbolic execution. However, the naive application of symbolic execution, hereafter called CLASSIC, does not scale because of the path-explosion problem. Event-driven programs, like any other type of programs, can have an extremely large number of program paths, and this number increases exponentially with the increase in the length of input event sequence.

Our new technique, called CONTEST, leverages the specific structure of the inputs (i.e., event sequences) of event-driven programs to address the path-explosion problem. The

[1] The problem of generating test inputs that violate safety properties can also be formulated as the branch-coverage problem, by converting a given `assert(e)` statement to `if (e) assert(false)`, and generating a test input that covers the `true` branch of the `if` statement.

key idea behind CONTEST is a novel notion of subsumption between event sequences. If an event sequence π is *subsumed* by another event sequence π', then CONTEST does not generate event sequences that are extensions of π. Instead, CONTEST generates events sequences that are extensions of π'. While such pruning of π produces compounded savings as the value k increases, CONTEST is still complete with respect to CLASSIC.

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 satisfied frequently in a specific subclass of event-driven programs that we used for evaluating CONTEST.

5.1 Smartphone Apps: A subclass of event-driven programs

To evaluate CONTEST, we chose a specific subclass of event-driven programs, called Apps, that run on mobile devices with advanced computing ability and connectivity, such as smartphones and tablets. Such mobile devices are becoming increasingly prevalent. Apps pervade virtually all activities ranging from mundane to mission-critical. Thus, there is a growing need for program-analysis tools in all stages of an app's life-cycle, including development, testing, auditing, and deployment.

Apps are also good targets for symbolic-execution based testing technique because they have many features that make static analysis challenging: use of a vast software framework, asynchrony, inter-process communication, databases, and GUIs. As a result, many proposed approaches for analyzing apps are based on dynamic analysis (e.g., [19, 26, 28]).

The most common type of inputs to an app are events such as a tap on the device's touch screen, a key press on the device's keyboard, and arrival of an SMS message. This chapter presents a technique 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 solution currently does not generate those types of events. In the context of systematic testing of apps, the previously-mentioned problem of generating event sequences of length up to k poses two separate challenges: (1) how to generate single

(i.e., $k = 1$) events and (2) how to extend single events to sequences of events (i.e., $k > 1$). We next look at each of these in turn.

5.1.1 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 [56, 74] or *model-based* techniques that require users to provide the model [82, 85]. 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, even when the platform detects 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.

Our solution to this problem, which uses dynamic symbolic execution (DSE), is general and fully-automatic. By general, we mean it is not specific to the type of mobile device (e.g., smartphone and tablet), the programming language that the app is written in (e.g., Java and Objective C), the operating system running on the device (e.g., Android and iOS), or the software framework that the app uses.

5.1.2 Generating Event Sequences.

Our DSE-based approach for generating single events can be extended naturally to iteratively compute sets of increasingly longer sequences of events. However, generating event sequences leads to the path-explosion problem. For instance, for a music player app, Classic (i.e., naive application of DSE) produces 11 one-event sequences, 128 two-event sequences, 1,590 three-event sequences, and 21K four-event sequences. In contrast, using $k = 4$, our algorithm Contest explores only 16% of the event sequences (3,445 out of 21,117) that Classic explores.

We have implemented our solution for testing apps in a system for Android, the currently dominant smartphone app platform. Our system instruments and exercises the given app in a mobile device emulator that runs an instrumented Android platform. 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 symbolic execution such as function summarization [34] and constraint solving [22]. We demonstrate the effectiveness of our system on five Android apps.

The main contributions of the work presented in this chapter are

1. A DSE-based 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.

2. A novel way to systematically generate events to exercise apps. Our approach, based on DSE, is fully automatic and general, in contrast to existing model-based or capture-replay-based approaches.

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 symbolic execution. We demonstrate the effectiveness of our system on five Android apps.

5.2 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 19 (Section 5.2.1). We then describe how we generate events to exercise this app (Section 5.2.2) and how we extend them to sequences of events (Section 5.2.3).

5.2.1 Example: Music Player App

Android apps are incomplete programs: they implement parts of the Android platform's Application Programming Interface (API) and lack the `main` method, which is the entry method for Java programs. When the music player app is started, the platform creates an

instance of the app's main activity `MainActivity`, and calls its `onCreate()` method. This method displays the main screen, depicted in Figure 20(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 platform 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 20(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 illustration, 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.

5.2.2 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 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*,[2] where each node denotes the rectangular bounding box of a different widget, and the node's parent denotes its containing widget. Figure 21 shows

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

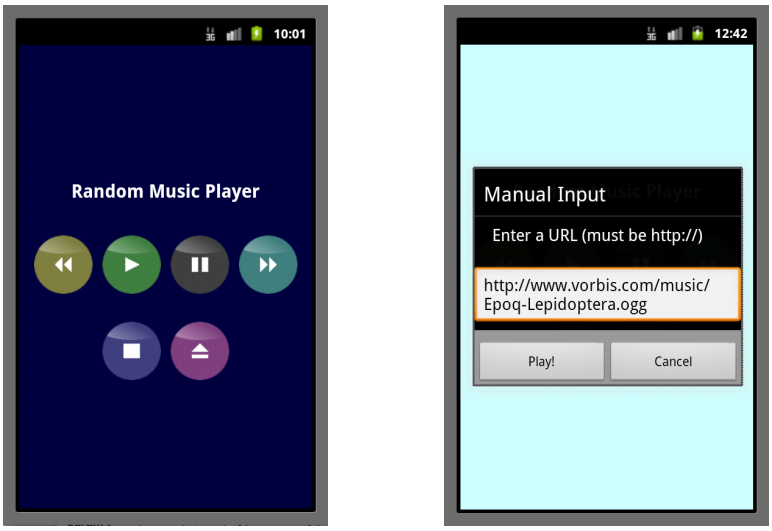
```
public class MainActivity extends Activity {
    Button mRewindButton, mPlayButton, mEjectButton, ...;
    public void onCreate(...) {
        setContentView(R.layout.main);
        mPlayButton = findViewById(R.id.playbutton);
        mPlayButton.setOnClickListener(this);
        ... // similar for 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 19: Source code snippet of music player app.

the view hierarchy for the main screen of the music player app depicted in Figure 20(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.

Existing approaches either infer the hierarchy automatically (capture-replay [56, 74]) or require users to provide it (model-based approaches [82, 85]); both have limitations. Capture-replay approaches are ad hoc: although the music player app uses only platform-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



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

Figure 20: Screen shots of music player app.

approach is based on DSE. 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 platform 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 platform 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 21. Section 5.3 describes how our approach generates these events in further detail.

5.2.3 Generating Event Sequences

Our DSE-based approach for generating single events can be extended naturally to iteratively compute sets of increasingly longer sequences of events. However, the Classic DSE approach [36, 70] causes the computed sets to grow rapidly—for the above music player app, Classic produces 11 one-tap sequences, 128 two-taps sequences, 1,590 three-taps

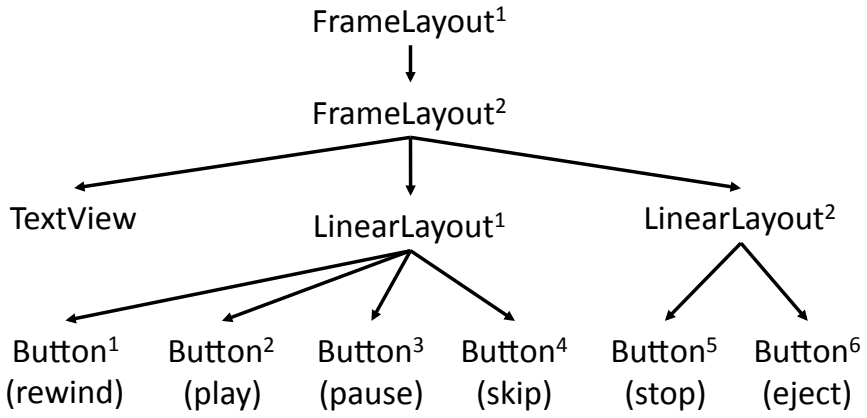


Figure 21: View hierarchy of main screen.

sequences, and 21K four-taps 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 21), 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 platform 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 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 19). 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 5.4 presents a formal description of subsumption, the read-only instantiation of subsumption, and the formal guarantees it provides.

5.3 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 22 shows the simplified code of the `dispatchEvent()` method of Android platform 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 platform 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

```
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 22: Source code snippet of Android platform.

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 21. 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).

Our approach uses DSE 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*:

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

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 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 platform, 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 symbolic execution that a real platform like Android demands: we instrument not only the platform code shown in Figure 22 but *all* platform code, as well as the code of each app under test. Dropping any of the above constraints due to missed instrumentation

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

Figure 23: Syntax of programs.

can result in the *path divergence* problem, where the concrete and symbolic values diverge and threaten the ability to cover all widgets.

5.4 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 5.4.1 and 5.4.2). We then describe our algorithm, proceeding from the top-level routine (Section 5.4.3) to the main optimization operator (Sections 5.4.4 and 5.4.5). Finally, we discuss the formal completeness guarantee of our algorithm (Section 5.4.7). Appendix A gives the proofs of all lemmas and the theorem discussed in this section.

5.4.1 Core Language

Our programming language is a standard WHILE language with one fixed input variable a and multiple global variables g_1,...,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 23. 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 variables g's and the 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.

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

Figure 24: Semantic domains.

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

5.4.2 Semantic Domains

We interpret programs using a slightly non-standard operational semantics, which describes the DSE of a program, that is, the simultaneous concrete and symbolic execution of the program. Figure 24 summarizes the major semantic domains. The most important are those for DSE states ω, input sequences π, and traces τ.

A ***DSE 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 DSE 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 DSE for a single event, and the W part stores variables written during the execution. As in the case of DSE 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 DSE 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 DSE 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 DSE 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 Figure 25. 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 (5.4.2) 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 DSE state restricts only the current input event, not any previous ones, in the input sequence.

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.

5.4.3 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.

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 26 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

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

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

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

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

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

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

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

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

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

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

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

Figure 25: Dynamic symbolic execution semantics.

Algorithm 1 Algorithm CONTEST

INPUTS: Program s_{in}, global state γ_{in}, bound $k \geq 1$.
OUTPUTS: Set of traces of length up to k.
$\Pi_0 = \Delta_0 = \{\epsilon\}$
for $i = 1$ *to* k **do**
  $\Delta_i = \mathsf{symex}(s_{in}, \gamma_{in}, \Pi_{i-1})$
  $\Pi_i = \mathsf{prune}(\Delta_i)$
end for
return $\bigcup_{i=0}^{k} \Delta_i$

Π_{i-1}. Hence,

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

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 DSE (modulo the well-known issue with loops), as we did in our implementation for Android.[3] 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 DSE 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:*

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

Trace τ describes the execution that takes the true branches of l_0 and l_1. It also records that

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

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

Figure 26: Simulation of our CONTEST algorithm.

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 $\bigcup_{i=0}^{2} \Delta_i$.

5.4.4 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 DSE 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 improves 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,*

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

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 $T \sqsubseteq T'$.*

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

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

Algorithm 2 The rprune operation

INPUTS: Set Δ of traces.

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

$k \geq 1$ 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}.

5.4.5 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:[4]

$$\Delta \sqsubseteq \Pi \cup \mathsf{sprefix}(\Delta), \tag{9}$$

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,* $\mathsf{rprune}(\Delta)$ *is a subset of Δ and satisfies the condition in* (9)*.*

We point out that the pruning operator can be implemented differently from rprune. As long as the pruned set Π satisfies the subsumption condition in (9), our entire algorithm

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

CONTEST remains relatively complete, meaning that the optimization with pruning will not introduce new uncovered branches. The following section presents another implementation of pruning that uses the notion of independence.

5.4.6 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 5.4.4 and Theorem 1 in Section 5.4.7 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 4. *The result of* iprune *is a subset of its input trace set, and it always satisfis the subsumption relationship in* (9)*.*

5.4.7 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:

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

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) $\beta_i.R \cap \beta'_j.W = \beta_i.W \cap \beta'_j.R = \beta_i.W \cap \beta'_j.W = \emptyset$ **then**
      for all i and j
   skip
  else
   $\Pi = \Pi \cup \{\tau\}$
  end if
end for
return Π

Theorem 1 (Completeness)**.** *For every* $k \geq 1$*,*

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

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

5.5 Empirical Evaluation

In this section, we present the empirical evaluation of our technique. First, we describe the implementation of Contest in Section 5.5.1. Next, we present the studies, including the subjects used, the empirical setup, the study results, and threats to the validity of the studies in Section 5.5.2.

5.5.1 Implementation

The implementation of our system uses the Soot framework [78], and consists of 11,000 lines of Java code. Figure 28 shows a dataflow diagram of our system. Our system inputs *Android framework*, 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 Figure 27 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

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

Figure 27: An example of the test script that our system generates.

a user waiting for the specified time for the app to respond to the preceding event. Our system adds `UserWait` events with a fixed waiting time after every tap event. Tests similar to the one in this script can be automatically executed using Monkey—a tool in the Android software development kit.

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

Instrumenter inputs *Android framework*, and the Java class files of the *App under test*, and outputs *Instrumented (framework+app)*. This component instruments the Java bytecodes of each class of the *App under test* and any third-party libraries that the *App under test* 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 four types of instrumentations. First, it instruments *Android framework* and *App under test* for DSE, 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 *Android framework* and *App under test* 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 instruments the *Android framework* to introduce symbols for each tap's `X` and `Y` coordinates. Fourth, Instrumenter ensures that

in *Instrumented (framework+app)*, user-specified method summaries are symbolically executed instead of the original methods.

Runner inputs *Instrumented (framework+app)*. The first time the component is called, it generates a test randomly; thereafter, it inputs tests from either the DSE 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 (framework)*. 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.

DSE 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 DSE. 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 [22].

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*.

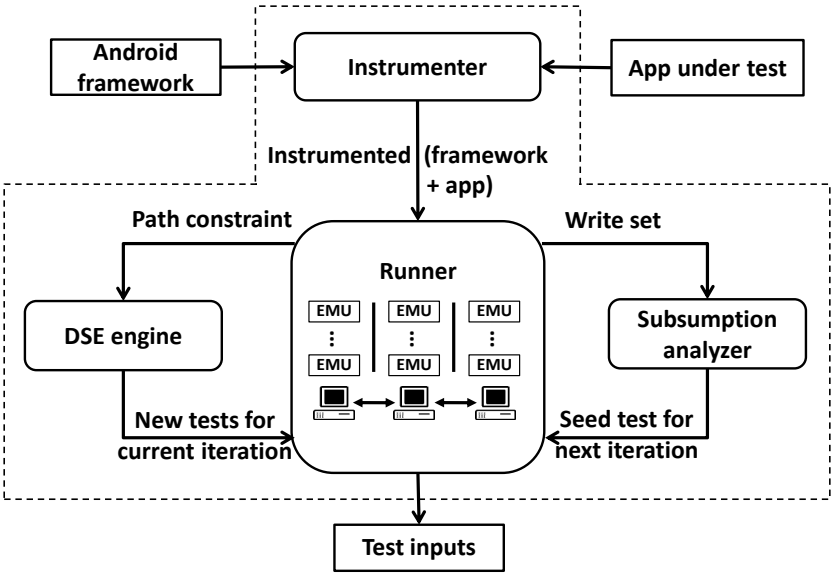


Figure 28: Dataflow diagram of our system.

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 during 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.

5.5.2 Studies

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 5.2. 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.

5.5.2.1 Study 1

The goal of this study is to measure the improvement in efficiency of CONTEST over CLASSIC. First, we performed DSE 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 29 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

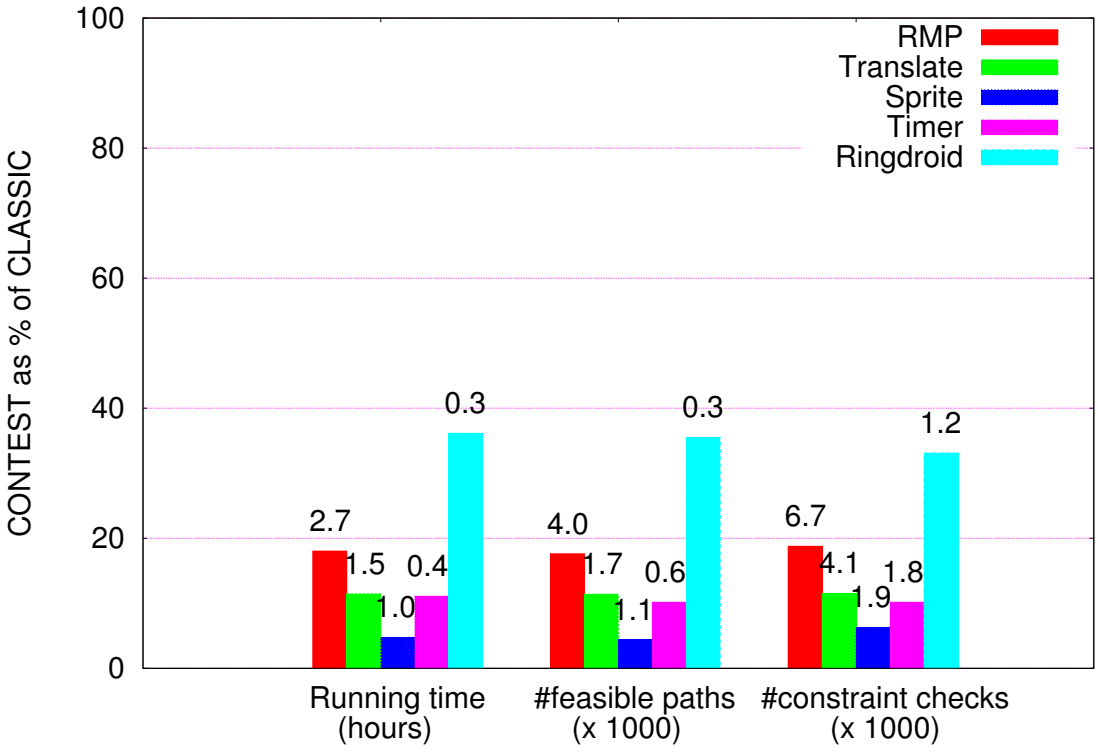


Figure 29: Results of Study 1: Running time, number of feasible paths explored, and number of constraint checks made by CONTEST normalized with respect to those metrics for CLASSIC.

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.

5.5.2.2 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 DSE 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 30 shows the results of the study. 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 30 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.

5.5.2.3 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 5.5.1 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.

5.6 Related Work

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

Memon presented the first framework for generating, running, and evaluating GUI tests. Several papers (e.g., [13, 57, 58, 84]) present components and extensions of this framework. Most existing GUI testing approaches either use capture-replay to infer the GUI model automatically [56, 74] or require users to provide the GUI model [82, 85]. An exception is the work of Ganov et al. [31], 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

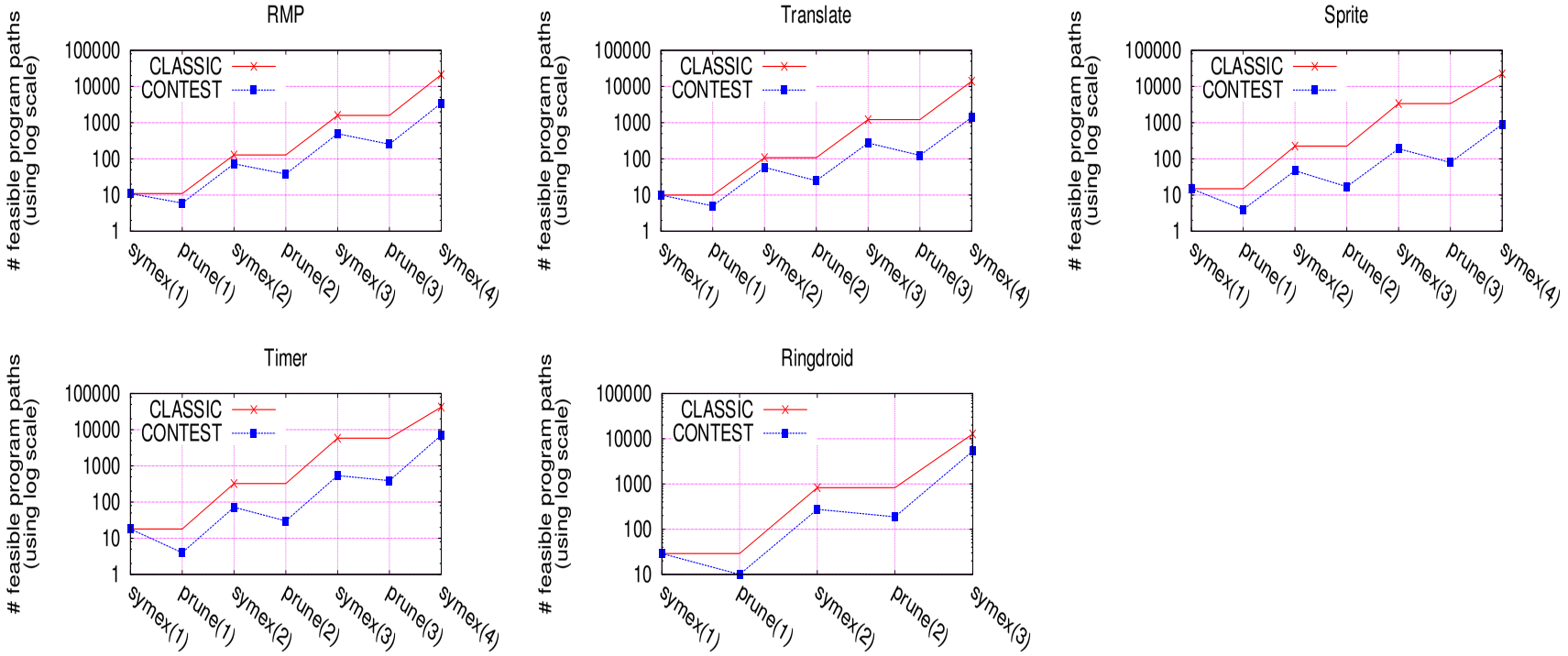


Figure 30: Results of Study 2: The number of paths (using a logarithmic scale) after symex and prune operations in each iteration. Because CLASSIC does not terminate for Timer and Ringdroid when k=4, the reported final numbers of paths for those two apps correspond to the time when the time-limit (12-hours) was met.

approaches.

Section 1.2.1 presents an overview of the large body of prior work exists that addresses the path-explosion problem. Our input subsumption idea is complementary to existing techniques for taming path explosion. Our system indeed leverages some of the existing 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.

5.7 Summary

This chapter presents a new technique, CONTEST that addresses the path-explosion problem that arises when symbolic execution is used for systematic testing of event-driven programs. As examples of event-driven programs, this work chose apps that run on mobile devices and receive various kinds of events as input. 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 CLASSIC.

There are 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). One direction for future work is to study other subsumption patterns besides the read-only pattern that we currently exploit to tame path explosion. 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, a more exhaustive empirical evaluation with more subjects can be conducted to further confirm CONTEST's improvement over CLASSIC.

CHAPTER VI

CONCLUSION

A symbolic-execution system that can be effectively used on real-world programs must be automatic and efficient. However, three important problems–path-explosion, path-divergence, and complex-constraint—arise in symbolic execution of real-world programs that make it challenging to build such a system. This dissertation presents three new techniques to facilitate symbolic execution of real-world programs. The new techniques partially solve those three problems.

First, due to reasons such as use of multiple programming languages and availability of parts of a program only in the binary format, it may not be possible to symbolically execute some parts of real-world programs. The user may have to examine those problematic parts, and either rewrite or specify models for them. The dissertation presents two complimentary techniques—type-dependence analysis and heap-cloning—that reduce this manual effort by identifying those parts of a program that a user must examine and manually address. Type-dependence analysis statically identifies problematic parts of a program. In contrast, heap cloning identifies problematic parts by performing dynamic symbolic execution. The strengths and weaknesses of the two techniques are complimentary.

In our empirical evaluation, type-dependence analysis determines that only 15% of all native methods that our subject programs use are potentially problematic (see Figure 12), and thus, a user has to examine and possibly provide models for only those methods, instead of all native methods. Similarly, for one of our subjects Lucene, heap cloning determines that only one out of 33 native methods that Lucene uses may introduce imprecision in path constraints (see Table 2), and thus need to be examined by the user.

Second, it may not be possible to solve constraints generated from symbolic execution because constraint solving is an intractable problem in general. Our solution to this problem is to statically identify problematic parts of a program that may generate complex

constraints, and require the user to rewrite or specify models for only those parts. Type-dependence analysis performs this identification. Finding all problematic parts at the same time provides an overall understanding of whether and how those parts are interrelated. If they are interrelated, then the user can develop a better solution to address those problems, compared to discovering one problem at a time as they are encountered dynamically. In our empirical evaluation, type-dependence analysis was able to pinpoint problematic parts in two large real-world programs.

Third, most real-world programs have an extremely large number of program paths. As a result, it is infeasible to symbolically executing all program paths. Two techniques—heap cloning and CONTEST—presented in this dissertation address this problem. First, by addressing the problems that arise in instrumenting Java's standard library classes, heap cloning enables implementation of precise and efficient dynamic-symbolic-execution systems for Java using the instrumentation-based approach. Such systems can symbolically execute more paths within a time budget compared to a system that is based on an alternative (i.e., custom-interpreter-based) approach. Second, CONTEST addresses this path-explosion problem for event-driven programs. CONTEST symbolically executes significantly fewer paths compared to the baseline technique, and thus, is more efficient, but at the same time, computes the same result as CLASSIC. We used CONTEST in a novel application of symbolic execution for systematic testing of specialized programs, called Apps, that run on modern mobile devices. Systematic testing for these programs is gaining importance because of their prevalent use.

Our empirical results show that a dynamic-symbolic-execution system for Java that leverages heap cloning and uses the instrumentation-based approach, can compute path constraints up to two orders of magnitude more efficiently than a system that uses the custom-interpreter-based approach. In our empirical evaluation of CONTEST, for one of our subjects RMP, CONTEST symbolically executes only 16% of 21,117 paths that CLASSIC explores (see Figure 29), in only 18% of 15 hours time that CLASSIC takes.

6.1 Merit of This Research

Symbolic execution is still viewed as a powerful technique that is used by researchers, but one that is not yet ready for common use in practice. One reason for this is several problems arise more commonly when the technique is applied to real-world programs instead of academic benchmark programs. By partially solving three such important problems (i.e., path-explosion, path-divergence, and complex-constraint), compared to existing works, this research enables application of symbolic execution to programs that are more similar to real-world programs with less manual effort and more efficiently. By extending the application scope of symbolic execution to new and more realistic types of programs, this research

- indirectly improves several techniques and tools that internal use this technique to solve various software engineering problems.
- may lead to discovery of both new research problems and insights to address known problems (e.g., the path-explosion problem) in a way that is highly effective in practice.

6.2 Future Work

We list four promising directions for future work.

6.2.1 Extending Heap Cloning

In this research, we used heap cloning for symbolic execution. However, the underlying idea of heap cloning is more generally applicable. Intuitively, heap cloning is a technique for determining side effects of some parts of a program, which cannot be analyzed or instrumented, during runtime. Thus, we believe that heap cloning has usefulness beyond symbolic execution. For example, heap cloning can be used to record program executions in the field, and then replay those executions at the developer's site for debugging or other purposes. In this example, it may be problematic to instrument all parts of a program because some parts may belong to third-parties, or they may be available only in the binary form, and thus, cannot be instrumented for recording executions. Heap cloning can be used to record and replay the side effects of uninstrumented code.

Furthermore, in our work, we implemented heap cloning for the Java programming language, and used Java's native methods as the only example of code that cannot be symbolically executed. However, the underlying idea of heap cloning is specific to neither Java nor native methods. Thus, one direction of future work is to extend the underlying idea of heap cloning to languages that are significantly different from Java (e.g., C), other types of code that cannot be analyzed or instrumented, and programs used in other domains (e.g., embedded systems).

6.2.2 Extending Contest

The Contest technique used a notion of subsumption condition between event sequences to reduce the number of paths that need to be symbolically executed. However, the subsumption condition is not specific to event sequences. It generalizes to program paths in any general program. However, checking such a generalized subsumption condition can be expensive. By focusing on event-driven programs we were able to efficiently check for subsumption of event sequences. One future direction is to discover efficient ways to check subsumption conditions for other types of programs. For example, it may be possible to define the subsumption condition differently so that it can be efficiently checked.

Another future direction can extend Contest for other types of event-driven programs such as embedded systems. For smartphone apps testing, Contest leverages specific patterns of program executions (e.g., read-only event) that commonly occur for apps. For other types of event-driven programs, the future work would involve studying whether the same kind of patterns also arise in those programs, and to discover new patterns that can be leveraged for efficient subsumption checking.

6.2.3 Testing and Analysis of Apps

Specialized programs, called Apps that run on mobile devices such as smart-phones and tablets are good targets of symbolic-execution-based analysis and testing techniques because (1) apps have many features that make static analysis challenging: use of a vast software framework, asynchrony, inter-process communication, databases, and GUIs; and (2) size of apps is typically smaller than size of other types of programs (e.g., desktop software). Their

small size can contain the notorious path-explosion problem to some extent.

Our work on testing apps (described in Chapter 5) lays a foundation for future symbolic-execution-based techniques in this field. Thus, one important direction of future work is to develop symbolic-execution-based techniques for problems that are arise in different stages of an app's life-cycle, including development, testing, auditing, and deployment. For example, one specific future work is to construct dynamic call graphs from executions of the test inputs that symbolic execution generates. Given that it is challenging to analyze apps and build call graphs statically, such dynamically-constructed call graphs can be useful.

6.2.4 Infrastructure for Real-world Software and Empirical Evaluations

Section 2.2 lists several symbolic-execution tools for Java and other programming languages. One common limitation among all existing (available or not) tools for Java is that they cannot apply to large real-world software. Thus, one important direction of future work is to develop an infrastructure that supports symbolic execution of real-world software. Specifically, the infrastructure should be able to apply the technique to software that use databases, file systems, networks, multiple large frameworks, multiple programming languages, and components that are available only in the binary format, etc. Such an infrastructure will require techniques similar to type-dependence analysis and heap cloning. Also, our dynamic-symbolic-execution system CINGER can provide a strong foundation for the infrastructure. We are planning to soon make CINGER publicly available.

Another direction of future work is to conduct an evaluation of the wide range of existing techniques that internally use symbolic execution to address various software-engineering problems, on a suite of real-world software. For example, it is known that the problem of exposing security vulnerabilities that arise in programs (e.g., web applications) that receive potentially untrusted data can be addressed using symbolic execution (e.g., [45]). However, it is still unknown how well the existing techniques for this problem work in practice.

The evaluation can produce three types of interesting results. First, the results will demonstrate whether the techniques presented in this dissertation and those developed by other researchers enable application of symbolic execution to a range of real-world software

without significant manual effort. Second, the results may show that a specific software-engineering problem can be solved highly effectively in practice. Such results will become examples of successful research, and may possibly open doors for technology transfer. Third, the results may uncover limitations of the existing techniques that future research can address.

APPENDIX A

PROOFS OF LEMMAS AND THEOREMS

In this part of the appendix, we provide proofs of Lemmas 1 and 2 stated in Section 5.4.4 and Lemma 3 and Theorem 1 stated in Section 5.4.5.

A.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

$$\langle s_{in}, n, \langle \gamma_0, \gamma_0, \epsilon\rangle\rangle \ \downarrow \ \langle \gamma_1, \Gamma, C\rangle \ \triangleright \ W. \tag{10}$$

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

$$\langle s_{in}, \pi', \gamma_{in}\rangle \ \Downarrow \ \gamma_0 \ \triangleright \ \alpha'. \tag{11}$$

From (11) and (10) 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. □

A.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

$$\gamma_0 \in \mathsf{final}(\alpha) \ \wedge \ \langle s_{in}, n, \langle\gamma_0, \gamma_0, \epsilon\rangle\rangle \Downarrow \langle\gamma_1, \Gamma_1, C\rangle \triangleright W. \tag{12}$$

Since $\alpha \sqsubseteq \alpha'$, γ_0 is also in $\mathsf{final}(\alpha')$. This and the second conjunct of (12) imply that $\alpha'\langle C, W\rangle$ is feasible. □

A.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* (9).

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 (9) 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. □

A.4 Proof of Lemma 4

Lemma 5. *The result of* iprune *is a subset of its input trace set, and it always satisfis the subsumption relationship in* (9)*.*

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 (9), 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. □

A.5 Proof of Theorem 1

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

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

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

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

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 lbranch on such trace sets by:

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

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 $\{0, \ldots, k-1\}$:

$$\bigcup_{i=0}^{j} \Delta_i \;\sqsubseteq\; \bigcup_{i=0}^{j} \Pi_i,$$

$$\bigcup_{i=0}^{j} F^i(\{\epsilon\}) \;\sqsubseteq\; \bigcup_{i=0}^{j} \Pi_i,$$

$$\mathsf{branch}(\bigcup_{i=0}^{j+1} F^i(\{\epsilon\})) \;=\; \mathsf{lbranch}(\bigcup_{i=1}^{j+1} F^i(\{\epsilon\})).$$

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:

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

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 (9) (Section 5.4.4), 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

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

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:

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

The two equalities here use the fact that $\mathsf{lbranch}$ preserves the union operator.

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

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

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 $\textsc{Contest}(s_{in}, \gamma_{in}, k)$. $\square$

REFERENCES

[1] Akers, S. B., "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 509–516, 1978.

[2] Anand, S., Godefroid, P., and Tillmann, N., "Demand-driven compositional symbolic execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 367–381, 2008.

[3] Anand, S., Naik, M., Yang, H., and Harrold, M. J., "Automated concolic testing of smartphone apps," No. GIT-CERCS-12-02, March 2012.

[4] Anand, S., Orso, A., and Harrold, M. J., "Type-dependence analysis and program transformation for symbolic execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 117–133, 2007.

[5] Anand, S., Pasareanu, C. S., and Visser, W., "JPF-SE: A symbolic execution extension to Java Pathfinder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 134–138, 2007.

[6] Berndl, M., Lhoták, O., Qian, F., Hendren, L., and Umanee, N., "Points-to analysis using BDDs," in *Conference on Programming Language Design and Implementation*, pp. 103–114, 2003.

[7] Binder, W., Hulaas, J., and Moret, P., "A quantitative evaluation of the contribution of native code to Java workloads," in *International Symposium on Workload Characterization*, pp. 201–209, 2006.

[8] Binder, W., Hulaas, J., and Moret, P., "Advanced Java bytecode instrumentation," in *International Conference on Principles and Practice of Programming in Java*, pp. 135–144, 2007.

[9] Bjørner, N., Tillmann, N., and Voronkov, A., "Path feasibility analysis for string-manipulating programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 307–321, 2009.

[10] Boonstoppel, P., Cadar, C., and Engler, D. R., "RWset: Attacking path explosion in constraint-based test generation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 351–366, 2008.

[11] Borges, M., d'Amorim, M., Anand, S., Bushnell, D., and Pasareanu, C., "Symbolic execution with interval constraint solving and meta-heuristic search," in *International Conference on Software Testing, Verification and Validation*, 2012. To appear.

[12] Brumley, D., Poosankam, P., Song, D. X., and 0002, J. Z., "Automatic patch-based exploit generation is possible: Techniques and implications," in *IEEE Symposium on Security and Privacy*, pp. 143–157, 2008.

[13] Bryce, R. C., Sampath, S., and Memon, A. M., "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 48–64, 2011.

[14] Burnim, J. and Sen, K., "CREST: Auomatic test generation tool for C." `http://code.google.com/p/crest/`.

[15] Cadar, C., Dunbar, D., and Engler, D. R., "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Symposium on Operating Systems Design and Implementation*, pp. 209–224, 2008.

[16] Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C. S., Sen, K., Tillmann, N., and Visser, W., "Symbolic execution for software testing in practice: preliminary assessment," in *International Conference on Software Engineering*, pp. 1066–1071, 2011.

[17] Castro, M., Costa, M., and Martin, J.-P., "Better bug reporting with better privacy," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 319–328, 2008.

[18] Chipounov, V., Kuznetsov, V., and Candea, G., "S2e: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, pp. 265–278, 2011.

[19] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A., "Clonecloud: elastic execution between mobile device and cloud," in *European Conference on Computer Systems*, pp. 301–314, 2011.

[20] Clause, J. A. and Orso, A., "Camouflage: automated anonymization of field data," in *International Conference on Software Engineering*, pp. 21–30, 2011.

[21] Dams, D., Hesse, W., and Holzmann, G. J., "Abstracting C with abC.," in *International Conference on Computer Aided Verification*, pp. 515–520, 2002.

[22] de Moura, L. M. and Bjørner, N., "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[23] Deng, X., Lee, J., and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *Internation Conference on Automated Software Engineering*, pp. 157–166, 2006.

[24] Dutertre, B. and de Moura, L., "A Fast Linear-Arithmetic Solver for DPLL(T)," in *International Conference on Computer Aided Verification*, pp. 81–94, 2006.

[25] Dwyer, M. B., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu, C. S., Robby, Zheng, H., and Visser, W., "Tool-supported program abstraction for finite-state verification.," in *International Conference on Software Engineering*, pp. 177–187, 2001.

[26] Enck, W., Gilbert, P., gon Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A., "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Symposium on Operating Systems Design and Implementation*, pp. 393–407, 2010.

[27] Factor, M., Schuster, A., and Shagin, K., "Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach," in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 288–300, 2004.

[28] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D., "Android permissions demystified," in *Conference on Computer and Communications Security*, pp. 627–638, 2011.

[29] Ganesh, V. and Dill, D. L., "A decision procedure for bit-vectors and arrays," in *International Conference on Computer Aided Verification*, pp. 519–531, 2007.

[30] Ganov, S. R., Killmar, C., Khurshid, S., and Perry, D. E., "Test generation for graphical user interfaces based on symbolic execution," in *Workshop on Automation of Software Test*, pp. 33–40, 2008.

[31] Ganov, S. R., Killmar, C., Khurshid, S., and Perry, D. E., "Event listener analysis and symbolic execution for testing GUI applications," in *International Conference on Formal Engineering Methods*, pp. 69–87, 2009.

[32] García, I., "Enabling symbolic execution of Java programs using bytecode instrumentation," Master's thesis, Univ. of Texas at Austin, 2005.

[33] Glover, F. and Kochenberger, G., eds., *Handbook of Metaheuristics*. Springer, 2003.

[34] Godefroid, P., "Compositional dynamic test generation," in *Principles of Programming Languages Symposium*, pp. 47–54, 2007.

[35] Godefroid, P., Kiezun, A., and Levin, M. Y., "Grammar-based whitebox fuzzing," in *Conference on Programming Language Design and Implementation*, pp. 206–215, 2008.

[36] Godefroid, P., Klarlund, N., and Sen, K., "DART: Directed automated random testing," in *Conference on Programming Language Design and Implementation*, pp. 213–223, 2005.

[37] Godefroid, P., Levin, M. Y., and Molnar, D. A., "Automated whitebox fuzz testing," in *Network and Distributed System Security Symposium*, 2008.

[38] Godefroid, P. and Luchaup, D., "Automatic partial loop summarization in dynamic test generation," in *International Symposium on Software Testing and Analysis*, pp. 23–33, 2011.

[39] Godefroid, P. and Taly, A., "Automated synthesis of symbolic instruction encodings from I/O samples," in *Conference on Programming Language Design and Implementation*, 2012. To appear.

[40] Grechanik, M., Csallner, C., Fu, C., and Xie, Q., "Is data privacy always good for software testing?," in *International Symposium on Software Reliability Engineering*, pp. 368–377, 2010.

[41] Igarashi, A., Pierce, B. C., and Wadler, P., "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, 2001.

[42] Jayaraman, K., Harvison, D., Ganeshan, V., and Kiezun, A., "A concolic white-box fuzzer for Java," in *NASA Formal Methods Symposium*, pp. 121–125, 2009.

[43] Khurshid, S., Pasareanu, C., and Visser, W., "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 553–568, 2003.

[44] Khurshid, S. and Suen, Y. L., "Generalizing symbolic execution to library classes," in *Workshop on Program Analysis for Software Tools and Engineering*, pp. 103–110, 2005.

[45] Kiezun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D., "Automatic creation of sql injection and cross-site scripting attacks," in *ICSE*, pp. 199–209, 2009.

[46] King, J. C., "A new approach to program testing," in *Programming Methodology*, pp. 278–290, 1974.

[47] Khknen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., and Niemel, I., "LCT: An open source concolic testing tool for Java programs," in *Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, pp. 75–80, 2011.

[48] Lakhotia, K., Tillmann, N., Harman, M., and de Halleux, J., "Flopsy - Search-based floating point constraint solving for symbolic execution," in *International Conference on Testing Software and Systems*, pp. 142–157, 2010.

[49] Lhoták, O. and Hendren, L. J., "Jedd: a BDD-based relational extension of Java.," in *Conference on Programming Language Design and Implementation*, pp. 158–169, 2004.

[50] Li, X., Shannon, D., Ghosh, I., Ogawa, M., Rajan, S. P., and Khurshid, S., "Context-sensitive relevancy analysis for efficient symbolic execution," in *Asian Symposium on Programming Languages and Systems*, pp. 36–52, 2008.

[51] Ma, K.-K., Khoo, Y. P., Foster, J. S., and Hicks, M., "Directed symbolic execution," in *International Static Analysis Symposium*, pp. 95–111, 2011.

[52] Majumdar, R. and Saha, I., "Symbolic robustness analysis," in *IEEE Real-Time Systems Symposium*, pp. 355–363, 2009.

[53] Majumdar, R. and Sen, K., "Hybrid concolic testing," in *International Conference on Software Engineering*, pp. 416–426, 2007.

[54] Majumdar, R. and Xu, R.-G., "Directed test generation using symbolic grammars," in *Internation Conference on Automated Software Engineering*, pp. 134–143, 2007.

[55] Majumdar, R. and Xu, R.-G., "Reducing test inputs using information partitions," in *International Conference on Computer Aided Verification*, pp. 555–569, 2009.

[56] Memon, A. M., Banerjee, I., and Nagarajan, A., "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Working Conference on Reverse Engineering*, pp. 260–269, 2003.

[57] Memon, A. M., Pollack, M. E., and Soffa, M. L., "Automated test oracles for GUIs," in *Internation Symposium on the Foundations of Software Engineering*, pp. 30–39, 2000.

[58] Memon, A. M. and Soffa, M. L., "Regression testing of GUIs," in *Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 118–127, 2003.

[59] Ngo, M. N. and Tan, H. B. K., "Detecting large number of infeasible paths through recognizing their patterns," in *Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 215–224, 2007.

[60] Pasareanu, C. S., Mehlitz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M. R., Person, S., and Pape, M., "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *International Symposium on Software Testing and Analysis*, pp. 15–26, 2008.

[61] Pasareanu, C. S. and Rungta, N., "Symbolic PathFinder: Symbolic execution of Java bytecode," in *Internation Conference on Automated Software Engineering*, pp. 179–180, 2010.

[62] Pasareanu, C. S., Rungta, N., and Visser, W., "Symbolic execution with mixed concrete-symbolic solving," in *International Symposium on Software Testing and Analysis*, pp. 34–44, 2011.

[63] Pasareanu, C. S. and Visser, W., "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[64] Qi, D., Roychoudhury, A., Liang, Z., and Vaswani, K., "Darwin: An approach for debugging evolving programs," in *Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 33–42, 2009.

[65] Saff, D., Artzi, S., Perkins, J. H., and Ernst, M. D., "Automatic test factoring for Java," in *Internation Conference on Automated Software Engineering*, pp. 114–123, 2005.

[66] Santelices, R. A., Chittimalli, P. K., Apiwattanapong, T., Orso, A., and Harrold, M. J., "Test-suite augmentation for evolving software," in *Internation Conference on Automated Software Engineering*, pp. 218–227, 2008.

[67] Santelices, R. A. and Harrold, M. J., "Exploiting program dependencies for scalable multiple-path symbolic execution," in *International Symposium on Software Testing and Analysis*, pp. 195–206, 2010.

[68] Saxena, P., Poosankam, P., McCamant, S., and Song, D., "Loop-extended symbolic execution on binary programs," in *International Symposium on Software Testing and Analysis*, pp. 225–236, 2009.

[69] Sen, K. and Agha, G., "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*, pp. 419–423, 2006.

[70] Sen, K., Marinov, D., and Agha, G., "CUTE: A concolic unit testing engine for C," in *Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 263–272, 2005.

[71] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D., "Detecting format-string vulnerabilities with type qualifiers.," in *USENIX Security Symposium*, 2001.

[72] Souza, M., Borges, M., d'Amorim, M., and Pasareanu, C. S., "Coral: Solving complex constraints for symbolic pathfinder," in *NASA Formal Methods*, pp. 359–374, 2011.

[73] Sridharan, M., Gopan, D., Shan, L., and Bodík, R., "Demand-driven points-to analysis for Java.," in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 59–76, 2005.

[74] Takala, T., Katara, M., and Harty, J., "Experiences of system-level model-based GUI testing of an Android application," in *International Conference on Software Testing, Verification and Validation*, pp. 377–386, 2011.

[75] Tilevich, E. and Smaragdakis, Y., "Transparent program transformationsin the presence of opaque code," in *International Conference on Generative Programming and Component Engineering*, pp. 89–94, 2006.

[76] Tillmann, N. and de Halleux, J., "Pex-White box test generation for .NET," in *International Conference on Tests and Proofs*, pp. 134–153, 2008.

[77] Tomb, A., Brat, G. P., and Visser, W., "Variably interprocedural program analysis for runtime error detection," in *International Symposium on Software Testing and Analysis*, pp. 97–107, 2007.

[78] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P., "Soot - A Java optimization framework.," in *Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 125–135, 1999.

[79] Veanes, M., de Halleux, P., and Tillmann, N., "Rex: Symbolic regular expression explorer," in *International Conference on Software Testing, Verification and Validation*, pp. 498–507, 2010.

[80] Volpano, D. M., Irvine, C. E., and Smith, G., "A sound type system for secure flow analysis.," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996.

[81] Whaley, J. and Lam, M. S., "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Conference on Programming Language Design and Implementation*, pp. 131–144, 2004.

[82] White, L. J. and Almezen, H., "Generating test cases for GUI responsibilities using complete interaction sequences," in *International Symposium on Software Reliability Engineering*, pp. 110–123, 2000.

[83] Xiao, X., Xie, T., Tillmann, N., and de Halleux, J., "Precise identification of problems for structural test generation," in *International Conference on Software Engineering*, pp. 611–620, 2011.

[84] Yuan, X., Cohen, M. B., and Memon, A. M., "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[85] Yuan, X. and Memon, A. M., "Generating event sequence-based test cases using GUI runtime state feedback," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, 2010.

[86] Zhang, P., Elbaum, S. G., and Dwyer, M. B., "Automatic generation of load tests," in *Internation Conference on Automated Software Engineering*, pp. 43–52, 2011.