University of California
Los Angeles

Symbolic Execution Algorithms for Test Generation

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Ru-Gang Xu

2009

The dissertation of Ru-Gang Xu is approved.

Jens Palsberg

Todd Millstein

Lei He

Rupak Majumdar, Committee Chair

University of California, Los Angeles

2009

For my Family.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank the many people who have encouraged my PhD studies and made my years at UCLA enjoyable.

First I would like thank my adviser Rupak Majumdar for his many years of exceptional guidance. I thank him for teaching me the art of research: how to find problems, solve them creatively and write about them precisely. I would also like to thank him for his patience and tolerance for my random escapades.

I would also like to thank the many UCLA professors whom I have enjoyed their classes and opinions: Eddie Kohler, for his systems courses and how to thoroughly critique a paper; Todd Millstein, for his classes in type systems; Jens Palsberg, for his insightful and entertaining commentary during our weekly reading groups.

I would also like to thank the many researchers that I have had the privilege of collaborating with: Ranjit Jhala, for his input on structural invariants, Patrice Godefroid for introducing me to directed testing and for hosting my stay at Bell Labs; Alex Groce for his expertise on how random testing is applied to real-world applications; Gerard Holzmann for hosting my stay at JPL; Andrey Rybalchenko for lessons in termination/non-termination.

Lastly, I would like to thank all the people in the our lab (TERTL) for making my day-to-day life entertaining: Lih Chen, Brian Chin, Petros Efstathopoulos, Mike Emmi, Jeff Fischer, Chris Frost, Pierre Ganty, Shant Hovsepian, Jacob Lacouture, Nikitas Liogkas, Mike Mammarella, Dan Marino, Shane Markstrum, Manav Mital, Rob Nelson, Milan Stanojevic, Steve VanDeBogart, and Alex Warth.

Vita

2001	B.S. (Electrical Computer Engineering), Carnegie Mellon, Pittsburgh, Pennsylvania.
2004	M.S. (Computer Science), University of California, Los Angeles, California.

Publications

R. Majumdar and R. Xu. *Reducing Test Inputs Using Information Partitions*, CAV, June 2009.

J. Andrews, A. Groce, M. Weston and R. Xu. *Directed Test Generation Using Symbolic Grammars*, ASE, September 2008.

R. Xu, P. Godefroid and R. Majumdar. *Testing for Buffer Overeflows with Length Abstraction*, ISSTA, July 2008.

A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko and R. Xu. *Proving Non-Termination*, POPL, January 2008.

R. Majumdar and R. Xu. *Directed Test Generation Using Symbolic Grammars*, ASE, November 2007.

R. Jhala, R. Majumdar and R. Xu. *State of the Union: Type Inference Via Craig Interpolation*, TACAS, March 2007.

R. Jhala, R. Majumdar and R. Xu. *Structural Invariants*, SAS, August 2006.

Abstract of the Dissertation

Symbolic Execution Algorithms for Test Generation

by

Ru-Gang Xu

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2009

Professor Rupak Majumdar, Chair

Correctness of software has become increasingly important and difficult as programs become more complicated and have more impact on our day-to-day lives. There are two approaches to ensure the correctness of software. Testing is the approach widely used in industry. Today, testing is tedious, expensive and prone to leave errors undetected. The other approach is to verify the correctness or guarantee the proper behavior of software through static analysis and model checking. However, this approach does not scale well, are restricted to simple properties or overwhelm the user with many false alarms. In the recent years, testing and verification have come closer together. *Directed testing or concolic testing* generates tests from constraints generated through both symbolic and real executions. However, the basic concolic execution algorithms do not scale to larger programs and cannot identify or seek out many types of bugs. This dissertation extends the basic concolic execution algorithm to scale to larger programs and more complex properties.

Specifically, this dissertation presents four symbolic execution algorithms that

automatically and systematically generate tests. These algorithms reduce the input space of automated testing and find different classes of errors. *Symbolic grammars* are introduced to generate orders of magnitude less input strings without sacrificing coverage. *Symmetry* reduces redundant tests by showing that some parts of the input are independent from other parts. Ideas in *Liveness* allow test generation to find errors leading to non-termination. *Abstraction* allows larger inputs to be generated that lead to memory safety violations and thus stop security holes before they happen.

This work has resulted in a tool that generates tests for C programs called SPLAT . SPLAT was used on a wide variety of open-source programs that compare these techniques to conventional industry-wide practices and-state of-the-art research. Preliminary studies show that these ideas are effective in finding new bugs quicker and can explore more of the program than other approaches.

CHAPTER 1

Introduction

The correctness of software has become extremely important. *Testing* has been the primary way that software is checked for correctness – costing billions of dollars from the software industry and accounting for about 50% the cost of software development [Mye79]. However, testing is never completely adequate thus leading to defects that cost the US economy about $60 billion every year [Nis02].

Two factors that contribute to the high cost of testing are the lack of automation and precise measurements of success. Although the current industry best practice is to attempt to test every aspect of a software system, testing requires substantial resources and can rarely check all possible execution scenarios. First, *testcases* need to be manually specified, that is the input and expected output must be defined. Also, test harnesses need to be created for software components. Then, tests must be repeatedly run as the software evolves. However, even when a large dedicated team of testers runs millions of tests, errors still remain in the final product [MSO06, CVE03]. It is difficult for the tester to know whether he is finished or measure how close he is to finishing. Because testing is unlikely to explore all the possible scenarios that the software may encounter upon release, the tester can only create tests based on the most likely usage of the software or on intuition where a bug may lie. In reality, testing is considered finished as soon as resources have run out, leaving no guarantee of correctness. This approach

often results in undetected errors.

On the other hand, *verification* gives a guarantee of correctness. Verification ensures the absence of certain property violations by proving such violations can never happen. Certain verification techniques such as model checking and static analysis are fully automatic, requiring little or no human intervention for a wide class of properties such as memory safety, the absence of runtime exceptions and termination. Unfortunately, verification techniques are difficult to use on complex software especially in the presence of dynamic memory allocation and data structures. Automatic techniques lead to *false warnings*, a warning that does not point to a real property violation. These false warnings must be manually filtered from the real bugs and can sometimes become overwhelming.

Currently the high cost of verification has made verification accessible to only the most safety critical software such embedded control systems or air traffic control and not to most commercial software. Relaxing correctness and focusing more on bug finding allow verification techniques to be applied on larger more complex software within a reasonable cost. This dissertation is a step toward the next generation of practical easy to use directed testing tools that can take the source code of a program and automatically generate test cases that cover a significant portion of the program and find bugs. Novel algorithms are introduced that balance the trade-off between cost and soundness by finding interesting inputs that explore a wider variety of behaviors instead of proving correctness. Because all algorithms presented will generate inputs, the results are easy to interpret for any programmer: a bug found can be examined by just running to program with the input.

One recently introduced approach toward this is *directed testing or concolic execution* where test inputs are generated from constraints derived from both

symbolic and real executions [GKS05, SMA05, CGP06]. Unfortunately, simply enumerating paths by solving constraints is not enough. Pure directed testing algorithms do not steer the execution toward bugs and can become lost – not finding any new bugs or improving code coverage after hundreds of hours.

This dissertation tries to remedy the two main short-comings of the standard concolic execution algorithm: state explosion and bug identification. This dissertation presents new symbolic execution algorithms for test generation that concolic execution more practical. All algorithms presented were implemented and applied to several open-source programs through a tool called Splat that automatically generates test inputs and systematically explores the input space while checking for certain property violations such as non-termination, runtime exceptions and unsafe memory accesses. Experiments show that these algorithms lead to greater code coverage and can find new bugs where the previous state-of-the-art could not.

Enumeration. First presented is an algorithm that combines exhaustive enumeration of test inputs from a structured domain with symbolic execution driven test generation, targeting programs whose valid inputs are determined by some context free grammar. The motivation is that concolic execution tools get stuck in the parser – endlessly generating malformed strings that just get thrown away. Instead of exploring paths in the program, the tool is just exploring the many paths that lead to a parse error. One possible solution is to enumerate the grammar of valid inputs; unfortunately, that also blows up because the number of valid strings greatly increase as the string length increases. To remedy this, the concrete input syntax is abstracted with *symbolic grammars*, where some original tokens are replaced with symbolic constants. This reduces the set of input strings that must be enumerated exhaustively. For each enumerated input string, which may

contain symbolic constants, symbolic execution based test generation instantiates the constants based on program execution paths. The "template" generated by enumerating valid strings reduces the burden on the symbolic execution to generate syntactically valid inputs while helping to exercise interesting code paths. Together, symbolic grammars provide a link between exhaustive enumeration of valid inputs and execution-directed symbolic test generation. Preliminary experiments with SPLAT show that the combination achieves better coverage than both pure enumerative test generation and pure directed symbolic test generation, in orders of magnitude less time and generated inputs.

Abstraction. Second, the dissertation discusses how to use abstraction to automatically generate inputs that lead to memory safety violations in C programs. Memory safety violations are notoriously difficult to find because they have a large input space. The solution is instead of representing the entire contents of an input buffer symbolically, SPLAT can track only a prefix of the buffer symbolically and a *symbolic length* that may exceed the size of the symbolic prefix. The use of symbolic buffer lengths makes it possible to compactly summarize the behavior of standard buffer manipulation functions, such as string library functions, leading to a more scalable search for possible memory errors. While reasoning only about prefixes of buffer contents may cause the search to be incomplete, experiments demonstrate that the symbolic length abstraction is both scalable and sufficient to uncover many real buffer overflows in C programs. On a set of benchmarks developed independently to evaluate buffer overflow checkers, SPLAT was able to detect buffer overflows quickly, sometimes several orders of magnitude faster than other approaches. Also, SPLAT was able to find two previously-unknown buffer overflows in a heavily-tested storage-system implementation.

Input partitions. Next, we investigate how to apply symmetry to reduce the number of paths explored for programs with independent inputs. Even though different combinations of inputs result in different paths, they do not result in new behaviors. Instead of tracking all bytes in the input buffer, the input is partitioned into "non-interfering" blocks such that symbolically solving for each input block while keeping all other blocks fixed to concrete values. Testing pieces in isolation is then shown to be able to find the same set of assertion violations as using the complete input buffer by proving there is no *information flow* between the input pieces. This can greatly reduce the number of paths to be solved (in the best case, from exponentially many to linearly many in the number of inputs). We present an algorithm that combines test input generation by concolic execution with dynamic computation and maintenance of information flow between inputs. Our algorithm iteratively constructs a partition of the inputs, starting with the finest (all inputs separate) and merging blocks if a dependency is detected between variables in distinct input blocks during test generation. Instead of exploring all paths of the program, our algorithm separately explores paths for each block (while fixing variables in other blocks to random values). In the end, the algorithm outputs an input partition and a set of test inputs such that (a) inputs in different blocks do not have any dependencies between them, and (b) the set of tests provides equivalent coverage with respect to finding assertion violations as full concolic execution.

Liveness. Lastly, we show how to use constraint solving techniques to find liveness errors, specifically inputs leading to infinite execution. These infinite executions are finitely represented with *lassos*, paths that end in a loop. We show how to find these lassos using concolic execution and how to prove that some of these lassos can lead to non-termination.

1.1 Contributions

Algorithms in this dissertation have been implemented in SPLAT – a tool that automatically generates inputs for C programs using symbolic execution and constraint solving [GKS05, SMA05, CGP06]. SPLAT scales to realistic programs and large input spaces using enumeration [MX07], abstraction [XMG08], and symmetry [MX09]. SPLAT also allows the detection of inputs leading to non-termination [GHM08] and a wide range of memory safety properties [XMG08]. Research contributions in SPLAT are:

1. *Symbolic grammars* describe input strings that satisfy the grammar describing valid inputs but with some symbolic variables [MX07]. Enumerating the symbolic grammars requires orders of magnitude less strings than the input grammar without missing bugs.

2. Describing an input string by a symbolic prefix and a *symbolic length* allows the generation of larger input strings that, although unsound, finds many memory safety bugs in real code [XMG08].

3. *Input partitions* reduces the number of paths needed to be tested by partitioning the input into non-interfering blocks and testing each block in isolation [MX09].

4. *Non-termination proofs* find that certain executions infinitely loop [GHM08]. Test generation finds candidate executions that may lead to non-termination. Some of these executions can be proven to lead to non-termination.

Each feature is evaluated by testing open-source C programs. The effectiveness of bug finding, code coverage and execution times are compared with other

state of the art techniques.

1.2 Outline

Chapter 2 provides some preliminary definitions that are used throughout the dissertation and an overview of concolic execution and its limitations. Chapter 3 introduces symbolic grammars and their use in generating structured inputs. Chapter 4 shows how abstraction can be used to find memory safety violations resulting from large strings. Chapter 5 shows how SPLAT can exploit non-interference to reduce the number of inputs explored. Chapter 6 shows how SPLAT can find bugs leading to non-termination.

CHAPTER 2

Background

Directed testing or *concolic execution* provides a systematic way of testing software where the only manual specification required is the size of the input [GKS05, SMA05, CGP06]. *concolic execution* is closely related to *symbolic execution* [Kin76, Cla76], where the program is executed on symbolic inputs, and satisfying assignments to constraints collected along a program path lead to new test inputs. Concolic execution combine symbolic execution with concrete random execution of the code. The concrete execution allows the symbolic execution to simplify constraints based on the concrete values along the run. Symbolic execution based test generation is *directed*: test inputs are generated by systematically exploring program paths at the symbolic level, and these inputs are then guaranteed to execute along pre-determined paths. Thus, the set of test inputs generated are not redundant: each leads to a different program path.

2.1 Example

Suppose the function `h` needs to be tested:

`h(int x, int y)` contains an error for some input as seen in line 04. Enumerating all 2^{64} or $1.8 * 10^{19}$ inputs is infeasible. However, enumerating all inputs is not necessary because many inputs result in the same execution path. In fact, there are only three unique paths in `h` and enumerating an input representing

```
00 int f(int x) {return 2 * x;}
01 int h(int x, int y) {
02   if (x != y)
03      if (f(x) == x + 10)
04         error();
05   return 0;
06 }
```

Figure 2.1: [Example] `h(int x, int y)`

each path would be sufficient to completely test `h`.

Concolic execution automatically finds each input that tests a new path. Suppose the first test is randomly chosen to be $(x = 3434, y = 2321)$. This results in an execution that enters the conditional in line 02 but does not enter the conditional in line 03. As the test is executed, predicates representing each conditional taken or not taken is recorded as the *path constraint*. For the first run, the path constraint is $\neg(x = y) \wedge \neg(2 * x = x + 10)$. To generate a new input, some predicate in the *path constraint* is negated. For a depth first search of all paths, the last predicate that had not been negated is negated and the solution of the constraint system with this negation provides the input going to that different path. In this example, the last predicate negated results in $\neg(x = y) \wedge (2 * x = x + 10)$. One solution to these constraints is $(x = 10, y = 2321)$, leading to the error in line 04. Suppose, the user decides `error()` is no longer an error and continues the search. The predicate $2 * x = x + 10$ is not negated because it already was. The next input is a solution for $(x = y)$, leading to a possible solution of $(x = 10, y = 10)$. Now the algorithm terminates because all predicates have already been negated.

2.2 Definitions

We introduce a simple imperative language with integer-valued variables that will be used to explain concolic execution and other algorithms presented in this dissertation. We represent programs as *control flow graphs (CFG)* $P = (X, X_0, \mathcal{L}, \ell_0, op, E)$ consisting of (1) a set of variables X, with a subset $X_0 \subseteq X$ of *input* variables, (2) a set of control locations (or program counters) $\mathcal{L}$ which include a special start location $\ell_0 \in \mathcal{L}$, (3) a function op labeling each location $\ell \in \mathcal{L}$ with one of the following basic operations:

1. a termination statement halt,
2. an assignment $x := e$, where $x \in X$ and e is an arithmetic expression over X,
3. a conditional $\mathsf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$, where $x \in X$ and ℓ', ℓ'' are locations in $\mathcal{L}$,

and (4) a set of directed edges $E \subseteq \mathcal{L} \times \mathcal{L}$ defined as follows. The set of edges E is the smallest set such that (1) every node ℓ where $op(\ell)$ is an assignment statement has exactly one node ℓ' with $(\ell, \ell') \in E$, and (2) every node ℓ such that $op(\ell)$ is $\mathsf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$ has two edges (ℓ, ℓ') and (ℓ, ℓ'') in E. For a location $\ell \in \mathcal{L}$ where $op(\ell)$ is an assignment operation, we write $N(\ell)$ for its unique neighbor.

Thus, the locations of a CFG correspond to program locations with associated commands, and edges correspond to control flow from one operation to the next. In the following, we assume that there is exactly one node ℓ_{halt} in the CFG with $op(\ell_{\mathsf{halt}}) = \mathsf{halt}$. A *path* is a sequence of locations $\ell^1, \ell^2 \ldots \ell^n$ in the CFG. A location $\ell \in \mathcal{L}$ is reachable from $\ell' \in \mathcal{L}$ if there is a path $\ell' \ldots \ell$ in the CFG. We

assume that every node in $\mathcal{L}$ is reachable from ℓ_0 and ℓ_{halt} is reachable from every node.

Semantics. The concrete semantics of the program is given using a *memory* that maps variables in X to values. For a memory M, we write $M[x \mapsto v]$ for the memory mapping x to v and every other variable $y \in X \setminus \{x\}$ to $M(y)$. For an expression e, we denote by $M(e)$ the value obtained by evaluating e where each variable x occurring in e is replaced by the value $M(x)$.

Execution starts from a memory M_0 containing initial values for input variables in X_0 and constant default values for variables in $X \setminus X_0$, at the entry location ℓ_0. Each operation updates the memory and the control location. Suppose the current location is ℓ and the current memory is M. If $op(\ell)$ is $x := e$, then the new location is $N(\ell)$ and the new memory is $M[x \mapsto M(e)]$. If $op(\ell)$ is $\mathsf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$ and $M(x) = 0$, then the new location is ℓ'' and the new memory is again M. On the other hand, if $M(x) \neq 0$ then the new location is ℓ' and the new memory remains M. If $op(\ell)$ is halt, the program terminates. Execution of the program starting from a memory M_0 defines a path in the CFG in a natural way. A path is executable if it is the path corresponding to program execution from some initial memory M_0.

2.3 Symbolic and Concolic Execution

We shall also evaluate programs *symbolically*. This is shown as Algorithm 2 and 1. Symbolic execution is performed using a *symbolic memory* μ, which maps variables in X to symbolic expressions over a set of symbolic constants, and a *path constraint* ξ, which collects predicates over symbolic constants along the execution path.

Execution proceeds as in the concrete case, starting at ℓ_0 with an initial symbolic memory μ which maps each variable x in X_0 to a fresh symbolic constant α_x and each variable $y \in X \setminus X_0$ to some default constant value, and the path constraint *true* (Algorithm 1 lines 1–4). To simply the algorithm, ξ is represented as a stack. For an assignment $x := e$, the symbolic memory μ is updated to $\mu[x \mapsto \mu(e)]$, where $\mu(e)$ denotes the symbolic expression obtained by evaluating e using μ and $\mu[x \mapsto v]$ denotes the symbolic memory that updates μ by setting x to v (lines 7–9). The control location is updated to $N(\ell)$. For a conditional $\mathsf{if}(x)\mathbf{then}\ \ell'\ \mathbf{else}\ \ell''$, there is a choice in updating the control location. If the new control location is chosen to be ℓ', the path constraint is updated to $\xi \wedge \mu(x) \neq 0$, and if the new control location is chosen to be ℓ'', the path constraint is updated to $\xi \wedge \mu(x) = 0$ (lines 10–16). In each case, the new symbolic memory is still μ. Symbolic execution terminates at halt (line 5).

For each execution path, every satisfying assignment to the path constraint ξ gives values to the input variables in X_0 that guarantee the concrete execution proceeds along this path. Given a path constraint ξ, we can have a new path if we can find a solution to a new path constraint where some predicate p is negated in ξ all predicates prior to p are removed. Algorithm 2 shows how to generate all paths by negating, solving and executing each predicate in depth first order.

Concolic execution [GKS05, SMA05] is a variant on symbolic execution in which the program is run simultaneously with concrete and symbolic values.

2.4 Limitations

Concolic execution offers a win over enumeration if one path can represent many inputs, in this example: 3 paths for $1.8 * 10^{19}$ inputs. However, concolic execution

Algorithm 1: **Execute**

Input: Program $P = (X, X_0, \mathcal{L}, \ell_0, op, E)$, Input *input*
Result: Path constraint ξ

```
1  for x ∈ X do
2  |  M(x) := input(x); if x ∈ X_0 then μ(x) := α_x;
3  end
4  ξ := emptyStack; ℓ := ℓ_0;
5  while op(ℓ) ≠ halt do
6  |  switch op(ℓ) do
7  |  |  case l := e
8  |  |  |  M := M[l ↦ M(e)]; μ := μ[l ↦ μ(e)]; ℓ := N(ℓ);
9  |  |  end
10 |  |  case if(x)then ℓ' else ℓ''
11 |  |  |  if M(x) = 0 then
12 |  |  |  |  ξ := push(μ(x) = 0, ξ); ℓ := ℓ'';
13 |  |  |  else
14 |  |  |  |  ξ := push(μ(e) ≠ 0, ξ); ℓ := ℓ';
15 |  |  |
16 |  |  end
17 |  end
18 end
19 return ξ;
```

has substantially more overhead then concretely executing the program due to the symbolic execution and constraint solving. It is clear that for programs where paths represent unique inputs, there is no gain in using concolic execution. In more complex programs, there exists a large number of paths that prevent directed testing to finish within any reasonable testing budget. Programs that take a `structured input`, an input that has to satisfy some specification, have parsing code that contains a unique path for each parse tree. Programs using inputs as a loop guard produce a new path at each iteration of the loop. For example, directed testing must enumerate all 2^{31} unique paths in function `g(int x)` in Figure 2.2.

As paths get longer and as the number of symbolic variables increase, con-

Algorithm 2: Generate

Input: Program P, partition Π, block $I \in \Pi$, flow map *flow*
Input: input *input*, last explored branch *last*

```
1  (ξ, flow) := Execute(P, Π, I, flow, input);
2  index := Length(ξ) − 1;
3  while not empty(ξ) ∧ index ≠ last do
4  |  p := pop(ξ);
5  |  if ξ ∧ ¬p is satisfiable then
6  |  |  input := Solve(ξ, ¬p);
7  |  |  flow := Generate(P, Π, I, flow, input, index);
8  |  index := index − 1;
9  end
10 return flow;
```

```
00 int g{int x} {
01   int ret = 0;
02   while (x > 0) {
03     ret += x;
04   }
05   return ret;
06 }
```

Figure 2.2: Path explosion: Paths get longer and longer

straint solving takes longer. Combined with the large number of paths in more complex programs, directed testing fails to scale.

2.5 Extensions

Few extensions of concolic execution have been proposed to tackle the path explosion problem. Concolic execution can be performed *compositionally* where *function summaries* — pre- and post-conditions describing the function call, are generated and used during concolic execution [God07]. *Demand-driven compositional testing* uses the least amount of intra-procedural paths to reach a specified

point in the program [AGT08]. RWSet removes paths that have the same side-effects as a previously explored path [BCE08].

CHAPTER 3

Symbolic Grammars

Automatic and comprehensive test input generation for large software programs where valid inputs to the system come from some structured domain is an important problem. Examples of such software systems are compilers or command processors, which accept inputs that form valid strings in some context free language, or business applications which process input described by some XML schema.

There are two predominant ways to automatically generate test inputs for such systems: enumerative and symbolic. In *enumerative* test generation, all inputs satisfying a certain input specification (for example, a grammar for a parser or an XML schema) are enumerated (up to some bounded size), and the program is executed on all the inputs [CL05, LS06]. As we have discussed in previous chapter, *symbolic* test generation [Cla76, Kin76, VPK04, BCH04, GKS05], the program is executed on symbolic rather than (or in addition to [GKS05, SMA05]) concrete inputs, and a set of constraints on the symbolic inputs is collected along an execution trace. A constraint solver is then used to generate test inputs that satisfy the symbolic constraints. The resulting test inputs are guaranteed to force the program execution along the path chosen by the symbolic execution.

Specification-based exhaustive enumeration is guaranteed to provide *valid inputs* to the program. This ensures that the application goes beyond the parsing and input sanitizing phase and executes along deeper paths. Unfortunately, ex-

haustive enumeration does not distinguish between different observable behaviors produced by the inputs. Thus, a large set of redundant tests may be generated, each of which has exactly the same execution behavior on the program. Also, for almost all nontrivial programs, the set of possible valid inputs is too large to completely enumerate, and in practice, one explores a random sampling of the input space, through some form of random or biased test input generation. This is not comprehensive: very often, the probability that random testing exercises program corner cases, where many bugs lurk, is astronomically small. In summary, specification-based exhaustive enumeration, while *selective*, in that it generates test inputs from the program's expected input domain, is *not directed*, in that the actual execution paths are not considered in the test generation.

In contrast, test generation based on symbolic or concolic execution is *directed*, exploiting path equivalences, and systematically exploring new paths. However, most symbolic execution implementations are *not selective*: they start with an unstructured buffer of symbolic variables, and hope to extract the structure of the input by looking at the tests executed along the path. While theoretically complete, symbolic techniques are expensive, and ultimately limited by the capacity of the symbolic engine. In practice, a symbolic test generator for a compiler stays "forever" within the many paths of the parser, generating incorrect inputs one after another, but exploring only novel parse error paths!

A test generation algorithm can combine the advantages of selective enumerative test generation and directed testing. However, there is a tension between the two techniques. For any single input, program execution is orders of magnitude faster than symbolic exploration, hence one should push as much work to enumerative testing as possible. On the other hand, the number of possible inputs to be enumerated is astronomical, so one should push as much work as possible to

the symbolic engine to explore only non-redundant computations. The solution is the use of *symbolic grammars* that balance the two competing requirements. Our test generation algorithm (1) transforms a grammar specifying input format into a symbolic grammar, (2) enumerates the set of valid strings in the symbolic grammar using enumerative techniques, and (3) runs directed testing on the symbolic strings enumerated.

Take a grammar G for arithmetic expressions (numbers, or the sum or difference of two arithmetic expressions):

$$\begin{aligned} \texttt{exp} &::= \texttt{num} \mid \texttt{exp} + \texttt{exp} \mid \texttt{exp} - \texttt{exp} \\ \texttt{num} &::= [0-9] \end{aligned}$$

For an input of size 3, enumerative techniques will generate all 210 valid strings of the form 0, ..., 9 (for integer constants), and $0+0$, $0-0$, $0+1$, $0-1$, ..., $9+9$, $9-9$. Symbolic techniques will start with three symbolic variables, and generate a large set of invalid inputs (e.g., "+00", "+–") to explore (the large number of) character-by-character comparisons in the lexer and error paths in the parser. In contrast, a *symbolic grammar* G' for G can replace the production of `num` with

$$\texttt{num} ::= \alpha$$

where α is a symbolic constant whose value is instantiated during symbolic exploration based on comparisons in the code. With this transformation, the number of possible valid strings of length 3 are α, $\alpha+\alpha$, and $\alpha-\alpha$. At this point, symbolic execution is run on the three inputs where symbolic constants are instantiated with respect to unique program paths.

Symbolic grammars enable several orders-of-magnitude decrease in the num-

ber of strings to be enumerated, and for each enumerated string, the symbolic constants generate enough non-determinism for the symbolic test generation to explore all paths of the program. Consequently, the use of symbolic grammars lets us profitably combine enumerative and symbolic test generation techniques to get a combined test generation algorithm whose performance should be much better than either alone. We have implemented CESE (Concolic Execution with Selective Enumeration), a tool that implements test generation using symbolic grammars for C programs that specify their input syntax using lex and yacc, on top of the YAGG string generator [CL05] and CUTE concolic execution [SMA05] tools.

We have applied our implementation to generate test inputs for a set of open source programs. In our initial experiments on a calculator for arithmetic expressions (used as an example application in many yacc tutorials), CESE outperformed both strictly enumerative and strictly symbolic test generation. The symbolic grammar had two orders of magnitude fewer strings to be enumerated. With symbolic grammar-based enumeration, CESE explored two orders of magnitude fewer inputs than CUTE for input buffers of size four, and could finish enumeration for larger buffers when CUTE could not finish within 5 hours. Similar trends were borne out in other experiments. Overall, CESE was able to achieve an average 10% more branch coverage than CUTE in a 30 minute testing budget. Further, limit experiments where CUTE was run for 5 hours showed that the branch coverage obtained by CUTE saturated (*i.e.,* , did not significantly improve over the coverage obtained in 30 minutes), and remained approximately 9% less than CESE running for 30 minutes. Further, for the programs in our suite that came with manual testcases, we saw that branch coverage obtained by CESE was within 10% of coverage with manual tests. This difference could be attributed to program behaviors that only manifest with larger input buffers. We

find this impressive: in spite of enumerating very small input buffers, CESE was able to come within the same ballpark as carefully crafted manual tests. In comparison to pure enumerative (grammar-based) input generation, CESE generated several orders of magnitude fewer inputs, and achieved slightly better (6% better) coverage under the same testing budget. Since generated tests are often added to regression suites, the many fewer tests generated by CESE (and consequently, the much lower test execution time) indicates a win for CESE . We also used CESE to check for buffer overflows, in particular, to check if a known buffer overflow in the path resolution function of the `wuftpd` FTP server can be detected. CESE found the bug in four minutes, whereas CUTE timed out without finding the bug in 13 hours. The specific configuration that leads to this bug requires a buffer of over 1000 bytes, making it outside the scope of exhaustive enumeration, and making the odds against random testing astronomically high. These initial results are clearly indicative that CESE is a scalable and useful technique for automated comprehensive test generation, and can match or outperform several known test input generation algorithms.

3.1 Example

We introduce and motivate our technique by testing a calculator example *SimpleCalc* that is seen in many tutorials for yacc [Joh75] and lex [LS75]. The *SimpleCalc* implementation consists of 1826 lines of generated C code. The grammar for *SimpleCalc* inputs is shown below.

$$
\begin{array}{lll}
\text{Expressions } e & ::= & (e) \mid e * e \mid e/e \mid e \ \% \ e \mid e + e \mid e - e \\
 & & \mid e \vee e \mid e \wedge e \mid -e \mid l \mid n \\
\text{Letters } l & ::= & [a - zA - Z] \\
\text{Numbers } n & ::= & [0 - 9]
\end{array}
$$

The program takes an arithmetic expression with letters as variables, various numerical operators, parentheses for precedence, and logical operators. The calculator implementation replaces letters with numbers that have been recorded in an array. Numerical and logical operators are directly applied, and precedence is handled during parsing. This implementation contains bugs: the *SimpleCalc* implementation forgets to check for division or modulus by zero.

We test *SimpleCalc* with a fixed input buffer of four bytes called *input*. We compare and contrast random testing, test generation using concolic execution using the tool CUTE , and concolic testing with selective enumeration using CESE . We restrict the size of our buffer to four so we can exhaustively test all program paths using both CUTE and CESE . Although it is generally infeasible to run either CUTE or CESE to completeness for large inputs or large programs, this small example clearly highlights the differences between naive concolic execution, concolic execution with selective enumeration, random testing, and specification-guided testing using concrete grammars. We compare the branch coverage obtained for both CUTE and CESE , where *branch coverage* is the percentage of branches executed, and whether the bugs can be found. We also examine the effect of increasing the input buffer size on these techniques.

3.1.1 Random Testing

With an input size of four bytes, there are $(2^8)^4 = 2^{32}$ unique inputs. The input space is too large for exhaustive testing all inputs. An automatic way of tackling this problem is to randomly choose inputs. However, we claim that random testing is not effective for this examples because the chances of hitting bugs are very low.

Based on the *SimpleCalc* grammar, there are 80,910 valid strings of size four,

27,032 of size three, 62 of size two and 62 of size one. To calculate the number of valid input buffers, we take account of the string terminator. With an input of size one, the string terminator must be at $input[1]$. The contents at $input[2]$ and $input[3]$ do not matter. Therefore, there is a total of $2^8 \cdot 2^8 \cdot 62 = 4{,}187{,}046$ inputs representing valid strings of size one. Following the same calculation, we have 15,872 inputs representing valid strings of size two, 27,032 of size three and 80,910 of size four, totaling close to 4.2 million valid inputs. Therefore, every input has around a 0.1% chance of being a syntactically correct input and the majority of these inputs will be only of size one, thus unlikely to exercise any interesting paths.

Generating an input that will show buggy behavior in this calculator is smaller. This implementation does not check for divide by zero errors, therefore operations dividing by zero or modulo by zero result in runtime exceptions. Valid strings containing "/0" or "%0" result in this error. For valid inputs of size four or less, there are only 372 inputs that demonstrate the error. Thus, random testing has a 0.000009% chance of hitting bugs. In fact, even if *SimpleCalc* is tested with 8 million random inputs, there is only a 50% chance that a bug causing string would have been generated. The problem, as is well-known, is that random testing is neither selective nor directed.

3.1.2 Constrained Exhaustive Enumeration

Specification-based test generation improves the pitfalls of random testing by generating inputs that are guaranteed to satisfy certain well-formedness specifications [CL05, GG75, BKM02, KM04]. In particular, there are test input generators that take as input a grammar (written, *e.g.,* in yacc) describing valid inputs, and generates test cases that satisfy the grammar [CL05, Mau90, LS06]. Usually,

these techniques exhaustively enumerate all inputs satisfying the specification, and test the program on all such inputs. Unfortunately, even for simple input specifications such as our grammar for *SimpleCalc*, the space of valid inputs is very big. As Table 3.1 demonstrates, for the *SimpleCalc* example, the number of valid strings for an input buffer of size six is already $187,765,078$. Enumerating and testing this large space of inputs is expensive. Moreover, certain errors may only be exhibited when the input buffer is much larger. Exhaustive enumeration can generate many equivalent test cases, *i.e.,* tests that have the same observable behavior on the program. In this example, the grammar-based input generator YAGG [CL05] generates the tests $0+0$, $0+1$, $0+2$, etc., all of which exercise the identical program path. Thus, while the enumerative strategy for specification-based testing is *selective*, it is not *directed*.

	Number of Inputs by Technique					**Time**		**Cov**
Len	**Grammar**	**Exh**	**Cute**	**SymStr**	**Cese**	**Cute**	**Cese**	
1	62	2^8	21	1	21	0.5s	4s	36%
2	124	2^{16}	247	2	40	2s	3s	52%
3	27K	2^{24}	2.5K	11	1.7K	20s	24s	54%
4	108K	2^{32}	248K	35	6.6K	30m	3m	56%
5	47MM	2^{40}	`n/a`	201	261K	`n/a`	1h	58%
6	187MM	2^{48}	`n/a`	652	1.5MM	`n/a`	3h	58%

Table 3.1: **Effect of input size. Length** is the maximum size of the input buffer. In the **Number of Inputs by Technique** column: **Grammar** denotes the number of syntactically valid strings, **Exh** denotes the number of unique buffers by exhaustive enumeration, **Cute** gives the number of inputs generated by CUTE , **SymStr** gives the number of strings in the symbolic grammar, **Cese** gives the number of inputs generated by CESE . **Time** gives the execution time for CESE denoted by **Cese** and the execution time for CUTE denoted by **Cute** in seconds (s), minutes (m) or hours (h). **Cov** shows the branch coverage for both CUTE and CESE in the first 4 rows, but just for CESE in the last two. CUTE did not terminate within 5 hours for those tests so such entries are marked as `n/a`.

3.1.3 Concolic Execution

We know compare how concolic execution performs on this example. Symbolic execution based test generation is *directed*: test inputs are generated by systematically exploring program paths at the symbolic level, and these inputs are then guaranteed to execute along pre-determined paths. Thus, the set of test inputs generated are not redundant: each leads to a different program path. Unfortunately, current implementations of concolic execution based test generation are not selective: test inputs are generated randomly, and iteratively refined using symbolic constraints. While theoretically complete in the limit, in practice, the lack of selectivity is a serious problem, and a very large number of inputs must be generated to reach the part of the code not related to input error handling. This leads to poor coverage for most realistic testing budgets.

We test the capability of symbolic execution based test generation on the calculator example, using CUTE , an implementation of concolic execution [SMA05]. To test *SimpleCalc* with CUTE , we created a symbolic input buffer of size four. CUTE then exhaustively generates all paths in the program by iteratively finding satisfying assignments to constraints that lead to paths that have not yet been covered. Unfortunately, the code for parsing examines all possible values for its input characters. For lex, this operation is represented by a table lookup. Figure 3.1 shows branches that are equivalent to this table lookup. From just these 10 branches, CUTE can derive 10^4 unique paths for a size four buffer. Coupled with the other branches in the code, CUTE needed to explore a total of 248,523 inputs, taking 30 minutes. This worsens as the input size increases. As Table 3.1 shows, we could not finish exhaustive testing of program paths for buffers greater than four characters even after 5 hours.

3.1.4 Cese

The main idea of CESE is to combine the selectiveness of specification-guided test generation with the directedness of symbolic or concolic test generation. To do this, we introduce *symbolic grammars*. It will be convenient for us to consider context free grammars where the terminal symbols are regular expressions rather than individual characters from an alphabet. A symbolic grammar for a (concrete) grammar replaces some terminals of the grammar with a symbolic constant. Each string in the symbolic grammar represents a set of strings, where each symbolic constant is substituted with a string in the regular expression which it represents.

For example, a symbolic grammar G'_{calc} for *SimpleCalc* replaces the concrete productions for letters and numbers with symbolic placeholders:

$$\begin{aligned} \text{Letters } l &::= \alpha \\ \text{Numbers } n &::= \beta \end{aligned}$$

where α and β are symbolic constants. With this change, the number of strings of a certain length that can be generated by the grammar reduces significantly. For example, instead of the 100 different strings "0/0", "0/1", ..., "9/9" representing division, we now have just one symbolic string "β_1/β_2" representing all these concrete strings. Note that we use subscripts for the different occurrences of the symbolic variables, each occurrence of a symbolic constant is instantiated separately.

The original program does not know about symbolic constants so our test generation algorithm must instantiate symbolic constants with actual constants. This instantiation can be performed in a *directed* way by treating symbolic constants as unconstrained symbolic values to be filled in by concolic execution.

Think of a string generated by a symbolic grammar as a string with "holes" for certain terminals. These holes are filled in by a concolic execution, depending on branches executed within the code. Together, the reduction in the number of possible strings in the language enables exhaustive enumeration to scale — thus providing selectivity— and concolic execution with the symbolic constants enables exploration of non-redundant strings — thus providing directedness.

This is the basic idea of CESE . We convert the concrete grammar to a symbolic grammar by replacing certain lexical tokens with symbolic constants. What tokens to replace is decided by a simple heuristic. If the token represents one concrete string (*e.g.,* lexical tokens corresponding to program keywords or operators), it is not replaced. On the other hand, if the lexical token corresponds to an unbounded set of concrete strings (*e.g.,* variable names, numbers), we replace it with a symbolic constant. Second, we exhaustively enumerate all symbolic strings from the symbolic grammar (G'_{calc} in the example) up to a certain size. Third, for each (symbolic) string, we use concolic execution to perform directed testing, where each symbolic constant is considered to be an unconstrained input to be solved for.

For example, "$\alpha_1 + \alpha_2$" is run by forcing only the second byte to be '+' and allowing concolic execution to generate values for α_1 and α_2 that exercise different paths. For this particular symbolic input, concolic execution exercised 188 unique paths. By working on the symbolic grammar, CESE has replaced 3,844 possible runs (corresponding to the valid grammar strings) with 188 concolic executions. Compared to CUTE , CESE gets the same coverage for substantially fewer inputs (6,611 versus 248,532) and an order of magnitude less time (3 minutes, versus 30 minutes). Table 3.1 shows the number of symbolic strings generated and also the number of concrete inputs generated from all those symbolic strings.

```
if (*yy_cp >= 0 && *yy_cp <= 0) yy_c = 0;
if (*yy_cp >= 1 && *yy_cp <= 7) yy_c = 1;
if (*yy_cp >= 8 && *yy_cp <= 8) yy_c = 2;
if (*yy_cp >= 9 && *yy_cp <= 31) yy_c = 1;
if (*yy_cp >= 32 && *yy_cp <= 32) yy_c = 3;
if (*yy_cp >= 33 && *yy_cp <= 47) yy_c = 1;
if (*yy_cp >= 48 && *yy_cp <= 57) yy_c = 4;
if (*yy_cp >= 58 && *yy_cp <= 96) yy_c = 1;
if (*yy_cp >= 97 && *yy_cp <= 122) yy_c = 5;
if (*yy_cp >= 123 && *yy_cp <= 255) yy_c = 1;
```

Figure 3.1: [Example] Calculator Lexer.

While the number of inputs still grows as the input buffer size increases, the significant reduction in the input space by moving to a symbolic grammar allows us to exhaustively search larger inputs. In further experiments detailed in Section 3.3, we have found that this combination of selective symbolic test input generation together with directed search is essential in scaling concolic test generation to real examples.

3.2 The CESE Approach

3.2.1 Symbolic Grammars

Let Σ be a finite alphabet. A *terminal* is a regular expression over Σ. We define a *grammar* $G = (V_t, V_n, R, S)$ where V_t is a finite set of terminals, V_n is a finite set of *variables*, $R \subseteq V_n \times (V_n \cup V_t)^*$ is a finite set of production rules, and $S \in V_n$ is a distinguished start variable. The language $L(G) \subseteq \Sigma^*$ of the grammar G is defined in the usual way [Sip97]. The language $L_h(G) \subseteq \Sigma^*$ of the grammar $G = (V_t, V_n, R, S)$ is defined as all strings derived from h applications of any of the productions rules R from the start variable S. A word $w \in L_h(G)$ has a *height* of h.

Let $\alpha_1, \ldots, \alpha_k$ be k symbolic names not in Σ. We assume each α_i stands for the regular language $\{\alpha_i\}$. A *symbolic grammar* G' for a grammar G w.r.t. terminals $T = \{t_1, \ldots, t_k\} \subseteq V_t$ is the grammar $(V_t \setminus T \cup \{\alpha_1, \ldots, \alpha_k\}, V_n, R[\alpha_i/t_i], S)$ where $R[\alpha_i/t_i]$ substitutes α_i for each occurrence of t_i for $i \in \{1, \ldots, k\}$. The language of the symbolic grammar G' is a subset of $(\Sigma \cup \{\alpha_1, \ldots, \alpha_k\})^*$. Notice that a string can now contain symbolic constants.

A symbolic grammar G' *abstracts* a concrete grammar G in the following sense. For any string $w \in L(G)$, there exists $w'(\beta_1, \ldots, \beta_k) \in L(G')$ with symbolic constants β_1, ..., β_k replacing terminals t_1, ..., t_k such that there exist strings $a_1 \in L(t_1)$, ..., $a_k \in L(t_k)$ such that $w = w'[\beta_1/a_1, \ldots, \beta_k/a_k]$.

Given a (concrete or symbolic) grammar G and a height h, all possible strings in $L_h(G)$ can be enumerated by dynamic programming [CL05].

3.2.2 The Cese Algorithm

The Cese algorithm has four phases: symbolic grammar construction, exhaustive enumeration, program instrumentation, and concolic execution. Let $P = (X, X_0, \mathcal{L}, \ell_0, op, E)$ be a program where $X_0 = \{x_i\}$ and $0 \leq i < k$ define some string $s = x_0, x_1 ... x_{k-1}$ of length k such that all valid inputs must satisfy some grammar G, $s \subseteq G$.

For the first phase, we construct a symbolic grammar G'_i for each concrete grammar G_i. The symbolic grammar construction uses the following heuristic. If a lexical token is a constant string (equivalently, if the regular expression defines a singleton language), then no symbolic constants are generated. Otherwise, we distinguish between finite regular languages and infinite regular languages. This distinction can be checked by looking for cycles in the derived automaton. For a finite regular language with bound k on the length of strings, we introduce

k symbolic variables $\alpha_1, \ldots, \alpha_k$ and replace the token with the k sequences α_1, $\alpha_1\alpha_2$, ..., $\alpha_1 \ldots \alpha_k$. For an infinite regular language, we replace the token with the symbolic regular language α^* denoting any number of symbolic constants.

In our experiments, the tokens either defined regular languages with one letter strings (*e.g.,* tokens for single-letter variable names in *SimpleCalc*), or infinite regular languages (*e.g.,* tokens for numbers).

However any subroutine that converts a concrete grammar to an abstracting symbolic grammar can be used. There is a trade-off between the number of symbolic strings with the number of symbolic variables in each string. As the number of symbolic variables increases, the number of valid symbolic strings decreases. For example, the coarsest abstraction is an unbounded number of symbolic letters. This symbolic grammar only contains four strings for an input of size four: α_1, $\alpha_1\alpha_2$, $\alpha_1\alpha_2\alpha_3$ and $\alpha_1\alpha_2\alpha_3\alpha_4$. However using this abstraction is equivalent to just using concolic execution.

Once a symbolic grammar is constructed, we use exhaustive enumeration techniques [CL05, LS06, Mau90] to generate strings from the grammar G'_i up to height h_i. The generated strings have both constant symbols and symbolic constants. For each choice $w \in L_{h_i}(G'_i)$ and the length of w is less than k_i, we introduce a loop in the beginning of our program: [1]

$$\textbf{for } j = 0 \textbf{ to } k_i - 1 \textbf{ do } x_j := \gamma[j]$$

where $\gamma[j] = w[j]$ if $w[j]$ is a constant symbol, and $\gamma[j] = x_j$ if $w[j]$ is a symbolic constant. The effect of the loop is to only retain the symbolic constants in the string as inputs, while instantiating all constant symbols.

[1] Our basic imperative language does not have a **for** loop. However, we write this for loop for readability. This can easily be converted to more basic control flow in our language.

Finally, we perform concolic execution on this instrumented program.

The correctness of the CESE algorithm is defined relative to the CUTE algorithm and the algorithm that enumerates all valid strings and executes the program on each string. Specifically, for any program P (that generates exclusively constraints within the capability of the underlying constraint solver), the set of paths explored by CUTE on *valid* inputs (an input is valid if the k characters do form a string in $L(G)$) is exactly the same as the set of paths explored by CESE . Further, this set is exactly the set of paths explored by exhaustive enumeration of all strings from G and executing the program on each string.

3.3 Experiments

CESE was implemented for programs that use yacc and lex to describe their inputs. Both symbolic grammar generation and symbolic string generation were automatic. We used YAGG [CL05] to automatically generate symbolic strings from our symbolic yacc and lex grammars and CUTE [SMA05] as our concolic testing engine. CUTE was modified to handle reasoning about statically allocated arrays by replacing those array accesses by branches, as seen in Figure 3.1. We used `lp_solve` [BDE07] as our underlying linear constraint solver. For real applications, there are rarely resources to explore all symbolic strings, therefore, we can choose which symbolic inputs to use. In the implementation, we sorted the inputs by the number of symbolic constants in the input. This optimization on the average increases coverage by 3% in our 30 minute experiments.

We ran two sets of experiments to test coverage and bug finding. Section 3.3.1 compares the effectiveness of CESE , naive concolic testing, random testing and specification based testing in branch coverage. Section 3.3.2 describes how

concolic testing and CESE can find a deep buffer overflow bug. All experiments were performed on a MacBook Pro 2.33 Ghz Intel Core 2 Duo with 2GB RAM running Mac OS X 10.4.8.

3.3.1 Coverage

Our first set of experiments measured branch coverage. We can distinguish a branch statically (*i.e.,* location in the code) or dynamically (*i.e.,* location on an executed path). Branch coverage is statically unique branches executed over all runs divide by the total number of branches in the program. For all experiments, CESE distinguished each branch dynamically to explore program paths, but then measured the number of statically unique branches covered. CESE can also explore paths based on distinguishing static branches only but the measured branch coverage is usually significantly reduced in that case[God07].

We tested five programs `bc`, `lua`, `logictree`, `cuetools`, and `wuftpd`. `bc` is the popular UNIX calculator. `lua` is an interpreter. `logictree` is a logical formula solver. `cuetools` is an API for playlists. `wuftpd` is a popular FTP server. These programs were modified to take a buffer as an input and were linked with a concolic execution aware string library.

Each program has several command line and configuration options. We restrict the programs to have their default configurations and do not explore the alternate configurations. We ran CESE on each program for 30 minutes. For each program, CESE generated words with up to h applications of the production rules of the symbolic grammar, where the parameter h was chosen so that all words that can be derived with $h-1$ applications, would be explored within 30 minutes. Table 3.5 shows the values of h in the **Height** column. Also note that the generated inputs can be used independently of CESE as part of a regression

suite. All inputs generated by CESE for all experiments can be rerun without the symbolic execution and constraint solving in less than 5 minutes. We focus on measuring test generation for the default configuration so we may compare against other approaches.

Manual Testing. Each program has various command line or configuration options, but for all experiments only the default configuration was used. Since we do not exercise all configuration options, full branch coverage is not possible. To estimate the maximum possible branch coverage based on the default configuration, we measure the branch coverage of test cases created by the program developers. We found testcases for all programs except for `wuftpd`. All manual testcases were relatively complex and required substantial knowledge of the program to create. These testcases included mathematical algorithms, sorting algorithms, and a large CD playlist.

Table 3.2 shows how CESE running for 30 minutes compares to manually created test cases. CESE 's automatically generated inputs have 10% less total coverage than the manual tests. We found this quite remarkable considering that two of these programs were language interpreters that included large sets of library functions that were unspecified in the grammar. Also, to our surprise, CESE performed slightly better in the `cuetools` testcase, because the developer only considered certain types of music lists and did not utilize the complete grammar. In the other programs, CESE was not as effective as manually created tests because CESE did not use a large enough input buffer or did not have time to enumeratively explore all symbolic constants.

Naive Concolic Testing. We used CUTE [SMA05] for the comparison. For all our CUTE experiments, we chose the input size to provide the best coverage for each 30 minute run. Input size was 10 for all CUTE experiments. Table

3.3 shows the results. On average, CESE had a 10% improvement over CUTE . Usually, CESE generates fewer inputs in the allotted time, because CESE explores deeper paths resulting in longer execution times while runs generated by CUTE are short runs resulting from parse errors. However in `wuftpd`, CUTE generated less inputs, because the number of symbolic constants and their constraints overwhelmed the constraint solver, causing it to timeout. As seen in Table 3.5, CESE could generate longer input strings with significantly less symbolic variables. CUTE required all of the input to be symbolic therefore creating a burden in the constraint solving and limiting the input size.

We also investigated how long it would take naive concolic test generation to achieve the same amount of coverage as running CESE for 30 minutes. CUTE cannot get close to the same coverage as CESE even when given ten times as much time. We ran CUTE for five hours on each program. There was only a slight increase (1%) in average coverage – still 9% less coverage than running CESE for 30 minutes. Note that for programs requiring larger valid strings such as cuetools, there was no improvement. CUTE is stuck in the parsing code – doing a search that is exponential in size of the input. CESE on the other hand, essentially skips a large part of the parsing code and its performance instead depends on the number of production rules and the number of symbolic variables in the symbolic grammar.

Specification-Based Testing. We used the concrete grammar to exhaustively generate inputs with up to h applications of the production rules where h was chosen such that we could explore all inputs with $h-1$ applications. Table 3.5 shows h per program in the **Height** column. Table 3.4 shows the coverage for 30 minute runs of grammar based testing. Grammar based testing generates more inputs than CESE in the same amount of time, but CESE explores inputs

with more height, therefore, more complex and longer paths. The introduction of symbolic letters in CESE reduces the input space without sacrificing coverage. Normally, this reduction is so significant that the cost of symbolic execution is worth it (*i.e.,* for `bc`, there were 790 CESE inputs of height 3 but 257074 concrete words of height 3). However, grammar based testing was slightly better than CESE for `logictree`. `logictree`'s grammar allowed grammar based testing to explore words of higher height than CESE in 30 minutes. However, as we explore words with increasing height, there is a combinatorial blowup in the number of concrete words. If testing increased to one hour, CESE will explore words of greater height than traditional grammar testing.

Overall, CESE only performed 6% better than enumeration-based testing in the same budget. However, the number of inputs generated by CESE is usually an order of magnitude fewer than the number of inputs generated by exhaustive enumeration. Moreover, without test generation, the total run time of CESE inputs was 5 minutes, in contrast to 2.5 hours for exhaustive enumeration. Given that generated inputs are often added to the regression suite, this clearly indicates the superiority of CESE with respect to test suite quality.

Random Testing. We also applied random testing for 30 minutes. We fixed the input length to be 10 for each program because that value gave the best coverage results. As seen in Table 3.4, random testing explored the most inputs but was the least effective, because most random inputs were invalid and only exercised the syntax error handling code of the test programs. These results reaffirm that random testing is ineffective for programs requiring structured inputs.

Discussion and Limitations. As described in Section 3.1, both concolic execution and specification based testing have limitations. The combination of the two, as shown in our experiments, lessen these limitations, allowing CESE to

have better performance and scale to larger programs.

Concolic testing is limited to the number and types of constraints generated by the program. If the constraints are beyond the theory of the constraint solver, concolic testing resorts to random testing. If the number of constraint becomes large, the constraint solver will become very slow. Although in our experiments all constraints were within the theory of `lp_solve`, both `lua` and `wuftpd` generated constraints that caused `lp_solve` to timeout when using naive concolic testing. Specifically, uses of `switch` statements and the `strlen` function introduces many inequalities in our constraints resulting in an exponential increase in the number of constraints to be solved. CESE greatly reduces the number of symbolic constants per input. Instead of testing the program with one large symbolic input, CESE divides the search space using knowledge of the grammar, thus allowing CESE to run all experiments without causing the underlying constraint solver to timeout. However, these limitations still affect performance when CESE is used to generate large inputs.

With larger inputs, lex and yacc style symbolic grammars do not give us enough constraints. Consider the following program that calculates the 10th factorial in the `bc` language:

```
define f (x) {
  if (x<=1) return(1)
  return (f(x-1)*x)
}
f (10)
```

This 60 character input contains 30 tokens and 9 symbolic constants in our symbolic grammar for `bc`. Generating interesting inputs of this size is still infeasible.

Enumerating all possible symbolic strings and executing them with a large number of symbolic constants would take far too many resources.

Also, the grammar does not capture semantic properties of the input. For example in `bc` and `lua`, we must rely solely on concolic execution to ensure only defined functions are being used, assigned variables are being read, input is type correct, etc. Other properties are not captured by either the grammar nor can be found by concolic execution. For example in `lua`, there is a large set of library functions such as "print" that can be called. These functions do not appear in the grammar, and calls to these functions are sufficiently deep making it hard if not impossible for concolic execution to realize them. To remedy this, one can either use a more descriptive grammar such as one that captures semantic properties *i.e.,* `f (10)` is a valid expression only if `f` has been defined, or to focus on specific classes of bugs.

In summary, our preliminary data suggests that CESE is highly effective in quickly generating a small test suite that can match or outperform many other test generation algorithms, and can get close to coverage obtained by manual testing while investing in a relatively short testing budget. However, more expressive specifications are required in order to explore deeper parts of the program state space.

3.3.2 Bug Finding

Next we investigate the effectiveness of using CESE to find a specific class of memory access bugs. We use CESE to find a known buffer overflow in `wuftpd` that is difficult to find with CUTE . We show that CESE allows concolic testing to scale so it can find interesting bugs that are beyond conventional techniques.

In the path lookup code in `wuftpd`, the `fb_realpath()` function has an off-by-

Program	LOC	Cese (30 min) Coverage	Inputs	Manual Testing Coverage
`bc`	12K	1010/2500 = 40%	133996	1235/2500 = 49%
`logictree`	8K	599/1376 = 43%	16827	740/1376 = 53%
`cuetools`	10K	572/1876 = 31%	99367	514/1876 = 27%
`lua`	32K	704/2422 = 29%	1061	1300/2422 = 54%
`wuftpd`	36K	552/1285 = 43%	10168	`n/a`
Total		3437/9459 = 36%		3789/8174 = 46%

Table 3.2: **Coverage**: Cese and Manual Testing. **LOC** is lines of code. **Coverage** is the branch coverage – executed branches divided by total branches. **Inputs** is the number of inputs generated. **Manual Testing Coverage** denotes the branch coverage for the developers' testcases. `n/a` denotes we could not find the developers' testcases.

one error that can be used as a buffer overflow with specifically crafted instructions. Figure 3.2 shows the bug. If `resolved` is equal to a non-root directory, then an extra "/" is added. Therefore, the `MAXPATHLEN` check is incorrect because `rootd` should be `!rootd` in line 07. Although this function is called by any command that uses pathnames, finding this bug is difficult because one needs to call this function with a pathname containing directory symlinks that results in a resolved pathname of exactly `MAXPATHLEN` size.

However even if we avoid this difficulty by restricting pathnames to contain a specific directory symlink that can exercise this error, finding the other pieces is still difficult. Suppose `MAXPATHLEN` is 1024 bytes and the directory link expands from a single letter directory link to a 23 letter directory name. Then the size of the string acting as the buffer must be exactly 1000 characters long. Also, there is the requirement of generating the right command. Random testing fails because the chance of both the directory and command string being generated is infinitesimally small. Although grammar-based testing will find the right commands to execute, grammar-based testing also fails because the number of valid

Program	LOC	Cute (30 min) Coverage	Inputs	Cute (5 hour) Coverage	Inputs
`bc`	12K	865/2500 = 35%	148868	883/2500 = 35%	949948
`logictree`	8K	298/1376 = 22%	225103	341/1376 = 25%	2133323
`cuetools`	10K	456/1876 = 24%	147915	456/1876 = 24%	720384
`lua`	32K	584/2422 = 24%	2734	668/2422 = 28%	22939
`wuftpd`	36K	238/1285 = 19%	1139	238/1285 = 19%	11195
Total		2441/9459 = 26%		2586/9459 = 27%	

Table 3.3: **Coverage**: CUTE . **LOC** is lines of code. **Coverage** is the branch coverage – executed branches divided by total branches, and **Inputs** is the number of inputs generated, for 30 minute CUTE runs and 5 hour CUTE runs.

Program	LOC	Grammar (30 min) Coverage	Inputs	Random (30 min) Coverage	Inputs
`bc`	12K	779/2500 = 31%	262773	626/2500 = 25%	401673
`logictree`	8K	620/1376 = 45%	272851	526/1376 = 38%	461524
`cuetools`	10K	425/1876 = 23%	256748	452/1876 = 25%	428672
`lua`	32K	650/2422 = 26%	245130	541/2422 = 22%	390404
`wuftpd`	36K	377/1285 = 29%	290356	68/1285 = 5%	489904
Total		2851/9459 = 30%		2213/9459 = 23%	

Table 3.4: **Coverage**: Grammar Based Testing and Random Testing. **LOC** is lines of code. **Coverage** is the branch coverage – executed branches divided by total branches, and **Inputs** is the number of inputs generated.

strings smaller than the bug causing input is huge. For example, there are 27^{1000} strings of lower case letters and the character '/' with length 1000. If we use a combination of random and grammar-based testing, we still are likely to fail to generate such a large string of that specific size.

Concolic testing can theoretically find this bug by using symbolic execution on string lengths. We experimentally compare pure concolic testing against CESE by using both techniques on `wuftpd`. The directory symlink is incorporated into both techniques as extra constraints on the input. The goal of the test is to generate

Program	Cese			Grammar	
	Height	Len	Sym	Height	Len
bc	3	10	2	3	10
logictree	6	5	3	7	6
cuetools	11	40	3	11	40
lua	11	20	5	7	10
wuftpd	15	21	4	13	16

Table 3.5: **Inputs**: CESE and Grammar Based Testing. **Height** is the maximum applications of production rules and **Len** is the maximum generated input length for grammar based testing and CESE . **Sym** is the maximum number of symbolic constants in CESE inputs.

any of the eight commands that can hit the bug and a string which, combined with the resolution of a known directory symlink, has a length of exactly `MAXPATHLEN`.

Length Abstraction. Although generating an input that exercises this buffer overflow requires a large pathname and therefore a large input buffer, both CESE and CUTE can use a *length abstraction* for the strings while generating inputs [DRS03]. The concolic execution input is changed to include both the original input buffer and a symbolic length. Concolic execution can track the length constraints on the inputs by by instrumenting the various string and memory manipulation library functions with the appropriate symbolic operations.

Using this length abstraction, CESE finds the overflow within four minutes while CUTE does not find it within thirteen hours. Symbolic grammars allowed concolic execution to scale by reducing the number of constraints needed to be solved and the total number of inputs needed to be explored. Specifically, CUTE is stuck tracking constraints in the parsing of commands. Even after 13 hours of execution and over 29,116 generated inputs, CUTE still does not increase its coverage of `wuftpd` and does not find the bug. On the other hand, CESE gets the command from the grammar and thus has fewer symbolic values to solve and

```
/*
 * Join the two strings together, ensuring that the
 * right thing happens if the last component is
 * empty, or the dirname is root.
 */
00    if (resolved[0] == '/' && resolved[1] == '\0')
01        rootd = 1;
02    else
03        rootd = 0;
04
05    if (*wbuf) {
06        if (strlen(resolved) + strlen(wbuf) +
07           rootd + 1 > MAXPATHLEN) {
08            errno = ENAMETOOLONG;
09            goto err1;
10        }
11        if (rootd == 0)
12            (void) strcat(resolved, "/");
13        (void) strcat(resolved, wbuf);
14    }
```

Figure 3.2: [Example] `wuftpd` buffer overflow bug. In line 07, `rootd` in the comparison with `MAXPATHLEN` should be `!rootd`

less inputs to generate. Therefore, CESE finds the bug in 4 minutes and achieves 43% branch coverage in 30 minutes while CUTE gets only 19% coverage and does not find the bug within 13 hours.

CHAPTER 4

Length Abstractions

Memory safety violations can lead to severe security vulnerabilities. Software containing these errors can lead to denial of service or loss of control to an attacker, costing billions of dollars in damage [FOB05]. Although many techniques and tools have been developed for finding such errors, none have been shown to be 100% effective on realistic code [ZLL04]. Proving the absence of these errors using static analyzes usually lead to many false warnings due to the lack of precise reasoning about bit operations, pointer arithmetic and arithmetic overflow. Finding inputs leading to such errors using random or systematic testing is also difficult due to the typically large input space.

Although concolic execution can be applied, the large input size and the large number of paths make full path coverage problematic for very large applications within a limited search period, say, one night. A lighter-weight approach to buffer overflow detection using concolic execution is needed. The insight is that tracking all symbolic values contained in input buffers is too precise and often unnecessary for detecting these types of errors, thus resulting in a large input space that cannot be searched completely in a reasonable amount of time. The idea is to track and symbolically reason about *lengths* of input buffers and strings. This is done by extending symbolic execution with respect to buffer contents to also include buffer lengths. On the other hand, in order to speed up the search and make it more tractable, symbolic execution only tracks the influence of data values stored in

prefixes of input buffers, instead of full buffers. This underapproximation has the effect of pruning the search space in a rather *uniform* manner. Preliminary experimental evidence of this is shown by code coverage data.

Because this technique explores an underapproximation of the input space, some errors may be missed. However, this underapproximation finds a wide class of common memory safety violations, because in many errors of this type, the length of the input is important while the contents are not. Furthermore, the underapproximation may be tuned to be more precise by making the symbolic input prefix longer. The user can initially use a short prefix and gradually increase the prefix as their testing budget allows. In the limit, all input could be made symbolic, allowing for completeness, but at a higher test generation cost.

Figure 4.1 shows a possible buffer overflow in procedure `t1` resulting from an off-by-one error when accessing array `A`. The input is a string s where the string length i is an index to an array of size 5. There is a check to see if the input i is within the bounds of the array but the check does not consider that i is incremented before the array access. If the input is 4, the bounds check passes, but $A[4+1]$ is set to 0 resulting in a memory safety violation.

Assuming code for the function `strlen` is available, a concolic execution testing tool such as DART [GKS05] will attempt to exercise all feasible program paths and find $n+1$ unique explicit paths for inputs s of up to length n. Indeed, covering all possible execution paths of the `strlen` function for an input size bounded by n requires $n+1$ input tests of different length. In this case, if n is greater than 4, the off-by-one error in accessing the array is found simply because all string lengths are enumerated up to n.

Note that this off-by-one error only depends on the size of the input string s, not on its content. To eliminate the redundant inputs due to the unfoldings of the

loop in `strlen`, we can track only the abstract length of the input string s and instrument the string library, including functions like `strlen`, with additional code that updates abstract lengths. `strlen` would thus be replaced by a function that returns a symbolic length. Then, program variable i would be assigned to this symbolic length. Using directed testing, this would result in two paths to be covered, each satisfying $(i > 4)$ or $(i \leq 4)$ respectively.

However, either path does not lead to a bug. To remedy this, we can include implicit paths by providing instrumentation that tracks allocated memory and adds the appropriate checks before each memory dereference [GLM07]. Function `t2` in Figure 4.1 shows these checks in the form of assertions. At each such assertion, we then try to solve for an input that violates the assertion. In this case, the path constraint for the assertion is $(i \leq 4) \wedge (i + 1 \geq 5)$ resulting in a symbolic input length $(i = 4)$.

This combination of techniques have been implemented in SPLAT , a tool for finding memory safety violations in C programs. In the next section, we illustrate further the key features of SPLAT with a more realistic example. In Section 4.2, we recall basic notions of directed test generation. In Section 4.3, I describe the implementation of SPLAT , and specify how to carry out symbolic execution with symbolic lengths. Section 4.4 discusses results of experiments with a large set of benchmarks. Experiments validate the choice of length abstractions, showing that SPLAT can efficiently find buffer overflows in many programs for which complete symbolic searches do not.

```
unsigned int strlen
             (char *s) {
    char *ptr = s;
    unsigned int cnt = 0;
    while (*ptr != '\0') {
        ++ptr;
        ++cnt;
    }
    return cnt;
}
void t2(char *s) {
     unsigned int i, tmp;
     int A[5];
     i = strlen(s);
     if(i > 4)
          return;
     tmp = i+1;
     assert(tmp>=0 && tmp<5);
     A[tmp] = 0;
}
```

```
void t1(char *s) {
    unsigned int i;
    int A[5];
    i = strlen(s);
    if(i > 4)
         return;
    A[i+1] = 0;
}
```

Figure 4.1: [Example] Buffer overflow due to off-by-one error in `t1`; additional instrumentation in `t2` with an assert

4.1 Example

Although Splat can find general memory safety violations, we introduce and motivate our technique by examining how we can find buffer overflows in C programs. Figure 4.2 illustrates a buffer overflow that was present in WuFTP, an ftp server. In this example, the string functions are the standard `string.h` functions. A string is stored in some fixed sized buffer as an array of non-zero 8-bit characters followed by a string terminator represented by a 0 byte. A buffer overflow occurs when a string is copied into some buffer that is too small. This is detected by Splat because copying a character beyond the end of the buffer results in an illegal write. Even though this example is small, it challenges most static analysis and automated test generation tools. Specifically, the path sen-

sitivity and pointer arithmetic causes static tools to report many false alarms, while the large buffers create scalability problems for test generation tools. We demonstrate how SPLAT overcomes these problems.

Testing Algorithm. Our algorithm for detecting buffer overflows implemented in SPLAT combines two ideas: first, systematically searching for test inputs using concolic execution [GKS05, SMA05], and second, tracking buffers and strings *partially symbolically*.

The systematic search for test inputs runs the program on symbolic values representing the input in addition to concrete inputs. The program is instrumented to additionally maintain a *valid range* for each pointer. The valid range denotes the set of addresses that can be safely accessed by the pointer. For example, for a pointer into a buffer, the valid range is between the start and end of the buffer.

Like the concolic execution algorithm discussed previously, SPLAT maintains a *symbolic state* that maps concrete addresses to symbolic expressions, and a *path constraint* that stores the sequence of conditionals executed, as well as a sequence of symbolic constraints representing the predicates in the conditionals. However, in addition to the standard concolic execution algorithm, at each memory dereference, if the dereferenced address is a symbolic expression, SPLAT constructs a symbolic constraint such that any satisfying assignment to this constraint will ensure that after executing the current path, the address being dereferenced will point outside the valid range. Thus, finding a satisfying assignment indicates a memory safety violation, and this satisfying assignment provides an input to the program that demonstrates the bug. This test is then generated and run to confirm the bug. If there is no satisfying assignment, the systematic search continues by generating a new input by modifying the path constraint and finding a

```
01: void lookup(char *resolved) {
02:  char *wbuf = "blah";
03:  if (resolved[0] == '/' &&
04:        resolved[1] == '\0')
05:         rootd = 1;
06:  else rootd = 0;
07:
08:  if (strlen(resolved) + strlen(wbuf) +
09:      rootd + 1 > 1024) return;
10:  if (rootd == 0)
11:      strcat(resolved, "/");
12:  strcat(resolved, wbuf);
13:
14: }
15:
16: void test() {
17:  char resolved[1024];
18:  input(resolved,5);
19:  lookup(resolved);
20:  exit(0);
21: }
```

Figure 4.2: [Example] Buffer overflow due to off-by-one error

satisfying assignment for this modified constraint. The new input is guaranteed to have a different execution path from all previous runs. SPLAT terminates when no new execution path can be found or when a bug is found.

Fully and Partially Symbolic Representations. In Figure 4.2, there is an off-by-one error that causes a buffer overflow in the `strcat` function on line 12. If `resolved` is equal to a non-root directory, then an extra "/" is added (lines 10–11). The length check on line 8–9 is incorrect, and `rootd` should be `!rootd`. The bug is exposed when `lookup` is called with a pathname that results in a resolved pathname length of exactly 1024. Since most static analyzers treat buffers and pointer arithmetic conservatively, they are likely to generate many false positives for any code of this form, and identifying this particular bug within this large set of false positives may be difficult.

Normally, concolic execution tools would track 1024 symbolic variables: one

for each character in the input (call this the *fully symbolic* representation). Unfortunately, introducing such a large number of symbolic values results in a large number of paths and a large set of symbolic constraints that stresses the capacity of the underlying constraint solver. Thus, this bug is difficult to find for directed testing tools using a fully symbolic representation. For example, running CUTE [SMA05] on the program of Figure 4.2 took 2 hours and generated 1019 paths before finding the error. Excess paths are created when running through the various string manipulation functions (such as `strlen`) because those functions are not summarized.

Our algorithm for buffer overflows is based on the following observations. First, for many buffer overflows (including this one), most of the actual content of the buffer is not relevant, what is relevant is the *length* of the string stored in the buffer, and *some small prefix* of that string. Therefore, instead of the exact fully symbolic representation, we use a *partially symbolic* representation that tracks a few characters of the stored string and its length symbolically while filling the rest of the buffer randomly.

Second, many strings are manipulated as an abstract datatype using the standard `string.h` header functions. Once we introduce this partially symbolic representation, we can precisely abstract the behavior of many header functions instead of stepping through them. For example, the `strlen` function can be abstracted to simply return the symbolic length of the string. This can drastically reduce the number of paths to be explored in the directed search.

With these two optimizations, our algorithm can reach all branches and detect all memory safety violations in `lookup` while exploring only a few paths.

In general, of course, partial symbolic representations can miss paths, but our approach allows the tester to iteratively increment the size of the symbolic prefix.

Running Splat . We demonstrate our technique step by step. The `test` function is the test harness and the starting point for SPLAT . For C programs, we introduce a function `input(p,k)` which specifies the input, where `p` is the address of the buffer storing the input and `k` is the number of symbolic entries in the input (*i.e.,* the symbolic prefix). In this example, we chose the symbolic prefix to be 5 characters. In general, the user can set a short prefix length and gradually increase the length as their test budget allows. To allow large input strings to be generated, a symbolic string length is associated with the input. If the symbolic length exceeds the symbolic prefix, characters beyond the prefix are randomly generated.

Thus, the test harness constructs the following inputs for Figure 4.2. First, it fills the character buffer `resolved` with random characters (each of size 8 bits) followed by the string terminator character. Of these random characters, the first 5 are tracked. At the same time, it constructs an integer representing the length of the string that is also tracked symbolically. Finally, it calls the `lookup` function with the buffer `resolved` (line 19). Such a test harness could be automatically constructed by static analysis of the C code [GKS05] with some default symbolic prefix length parameter.

During the first run, the input string will be of length 5 with randomly chosen non-zero characters in the first five entries and a string terminator in the last entry. Let's say for this run, we randomly generate `resolved =` "a1weq". We introduce 5 symbolic values representing the first 5 elements of `resolved`: $\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4$ and a symbolic value β representing the string length. We instrument the `strlen` function to return the symbolic length of a string. Thus, when called with `resolved`, `strlen` returns the symbolic value β as the length. When the program is executed with this input, it does not take the then

branch at line 3 nor the then branch at line 8. Executing to line 9, we have $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge \neg(\beta + 5 > 1024)$ as the path constraint.

Notice that there is no predicate representing the branch at line 10. This is because the variable `rootd` is not symbolic, hence the conditional `rootd == 0` evaluates directly to `true` and so is not included in the path constraint (see [GKS05]). At line 11, we update the symbolic state of `resolved` by updating the string length to $\beta + 1$. At this point, we have to check whether the call to `strcat` at line 11 can cause a buffer overflow. To check this, we ask whether there is a satisfying assignment for the extended path constraint $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge \neg(\beta + 5 > 1024) \wedge (\beta + 1 \geq 1024)$. Since the above constraint is unsatisfiable, there is no buffer overflow (yet).

At line 12, we update the string length of `resolved` to be $\beta+5$ and again check for a possible overflow. This time, we check if there is a satisfying assignment for the extended path constraint $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge \neg(\beta + 5 > 1024) \wedge (\beta + 5 \geq 1024)$. This constraint is satisfiable and a solution is $\beta = 1019$ and $\alpha_0\alpha_1\alpha_2\alpha_3\alpha_4 =$ "a1weq". This indicates a potential buffer overflow. Next we generate an input string with a prefix of "a1weq" but of length 1019 by filling the non-symbolic suffix with random non-zero characters. This new test case causes the `resolved` array to overflow.

Suppose now we fix the bug by replacing `rootd` in line 9 with `!rootd` and rerun SPLAT . The first run with "a1weq" passes all memory violation checks. We create a new test case by negating the last branch predicate, proceeding in a depth-first order. Our path constraint currently is $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta + 6 \leq 1024)$. We solve for a new test case satisfying $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta + 6 > 1024)$, getting a 1019 length string with "a1weq" prefix. After the next run, we get the path constraint $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta + 6 > 1024)$. We recognize that

both branches of the last conditional statement have already been explored so we negate the first condition and solve for the path constraint ($\alpha_0 = '/' \wedge \alpha_1 = 0$), getting the string "/" as the next input. We dropped the (β+6 > 1024) constraint because by negating an early branch predicate we can no longer guarantee that we will hit the later branch. The third run with "/" as input has ($\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta + 6 \leq 1024)$ as its path constraint. Again we search alternative path constraints depth first and negate the last branch, getting the new path constraint $(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta + 6 > 1024)$. However, β represents the symbolic length of the input and the second character α_1 is the string terminator so there is no satisfying assignment for this constraint. Then, SPLAT terminates after exploring all three paths in `lookup`.

Summary. SPLAT is composed of three main ingredients: (1) instrumentation at every memory access to detect memory safety violations (buffer overflows), (2) concolic execution, and (3) partially symbolic representations with symbolic tracking of string length and symbolic summaries of string library functions. Memory safety violations are found by tracking (de)allocations of memory and insuring all dereferences stay within their respected bounds. The systematic search of directed testing insures all explicit paths will be explored. Combining (1) and (2) checks each memory dereference along all explicit paths, that is all implicit paths leading to possible memory safety violations are explored. In addition, (3) scales SPLAT to realistic programs that rely on the C string library by reducing the burden on the symbolic path exploration. In the experimental section, we validate our choice of partially symbolic representations by showing that, despite the lightweight nature of the abstraction, it is sufficient to find many buffer overflows in real benchmarks.

4.2 Memory Violation Checking

We shall extend our previous language that we used to present concolic execution with memory allocation of arrays. That is, for some program $P = (X, X_0, \mathcal{L}, \ell_0, op, E)$, $x \in X$ can be either an integer or an array of integers. We define an array A of size k as a mapping from $0 \leq i < k - 1$ to integers. We introduce array indexing and the sizeof function as new subexpressions. The i-th entry of the some array stored in x is accessed by $x[i]$, this access does not result in an error if x is an array of length k and $0 \leq i < k$. For some array x, $\mathsf{sizeof}(x)$ returns the length of x. Arrays are allocated using $x := \mathsf{malloc}(k)$ where after the statement x points to an array of size k and array deletion $\mathsf{free}(x)$ where after the statement, array pointed by x is removed. These operations allow us to model dynamic memory allocation on the heap and limited pointer arithmetic.

Semantics. Semantics remain the same for other operations except M is now a mapping from variables to arrays or integers. For a memory M, we write $M[x \mapsto A]$ for the memory mapping x to some array A and every other variable $y \in X \setminus \{x\}$ to $M(y)$. If x maps to some array A of length more than k, the operation $x[k] = e$ updates $A[k]$ to be $M(e)$, error other wise. After an allocation operation $x := \mathsf{malloc}(k)$, a new array of size k is created with default values of zero, that is $A[i] = 0$ for all $0 \leq i < k$ and $M[x \mapsto A]$. After a deletion operation $\mathsf{free}(x)$, we return an error if x does not map to an array, otherwise, $M[x \mapsto 0]$.

Symbolic Execution. SPLAT performs symbolic execution of the program together with concrete execution, similar to previous discussions. However there are additional changes and checks to handle arrays. The *symbolic memory map* μ may map variables to symbolic arrays. A *symbolic array* $A' = \langle \varphi, \varphi_l \rangle$. φ_i represents the symbolic length of the array. $\varphi = \langle \varphi_0, \ldots, \varphi_{k-1} \rangle$ represent the

first k symbolic values of the array which we call the *symbolic prefix*. For input arrays, *i.e.,* $x \in X_0$ and $M_0(x) = A$, we have $\mu(x) = A'$ where $A' = \langle \varphi, \varphi_l \rangle$ and $\varphi = \langle \alpha_0, \ldots, \alpha_{k-1} \rangle$ where $alpha_i$ is a fresh symbolic constant that represents the first k symbolic values of the array.

After an allocation operation $x := \mathsf{malloc}(e)$, we have $\mu[x \mapsto A']$ where $A' = \langle \varphi, \mu(e) \rangle$ and $\varphi = \langle 0, ..., 0 \rangle$. After an free operation $\mathsf{free}(x)$, we return an error if x does not map to an array in M, otherwise, $\mu[x \mapsto 0]$. If the $\mathsf{sizeof}(x)$ is in an symbolic expression, we check if $\mu(x)$ maps to some array $A' = \langle \varphi, \varphi_l \rangle$ and replace $\mathsf{sizeof}(x)$ with φ_l, otherwise we throw an error.

Constraint Solving. Like the standard concolic execution algorithm, the path constraint ξ is initially *true*. At every conditional statement ℓ : $\mathsf{if}(x)\mathsf{goto}\ \ell'$, if the execution takes the then branch, the symbolic constraint ξ is updated to $\xi \wedge \neg(\mu(x) = 0)$ and if the execution takes the else branch, the symbolic constraint ξ is updated to $\xi \wedge (\mu(e) = 0)$. For each array access $x[e]$, we check if x is an array, that is if $\mu(x)$ maps to some $A' = \langle \varphi, \varphi_l \rangle$, otherwise we report an error. If $\mu(x)$ does map to some A'. We check if $\xi \wedge \mu(e < 0)$ is satisfiable and if $\xi \wedge \mu(e > \varphi_l)$. If we find solutions to those contraints, we have found a negative index or an array out-of-bound access and thus we terminate and report an error.

Solving constraints lead to new paths, as discussed in the standard concolic execution algorithm, or lead to previously explored paths that result in a memory safety violation. Solving occurs as before except the length solved for an input can exceed the *symbolic prefix*. In that case, random values are placed in array slots that exceed the prefix.

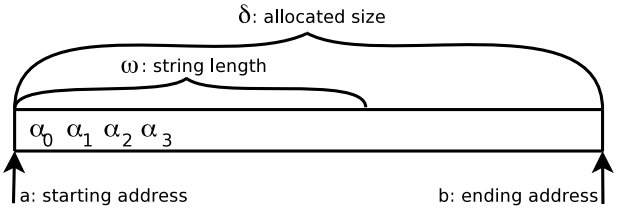

Figure 4.3: Memory nodes contain possibly symbolic representations of string length and size of allocated memory.

$$
\begin{array}{lcl}
\text{p = malloc(s)} & \rightarrow & create(p, p+s, \mu(s)) \\
\text{int i} & \rightarrow & create(\&i, \&i + sz(i), null) \\
\text{*p} & \rightarrow & mn = find(p) \\
 & & assert(p \leq mn.\delta + mn.a) \\
 & & assert(p \geq mn.a) \\
\text{free(p)} & \rightarrow & mn = find(p) \\
 & & assert(p \leq mn.\delta + mn.a) \\
 & & assert(p \geq mn.a) \\
 & & delete(mn)
\end{array}
$$

Figure 4.4: Tracking memory for heap and stack allocations, and checking pointer dereferences.

4.3 Implementation

We implemented Splat for testing C programs. Splat consists of three parts: a source-to-source instrumenter, a library for tracking memory allocations, deallocations, and accesses, and a library for symbolic execution, constraint solving, and coverage tracking.

The instrumenter takes the source code of the program and adds calls to the runtime library that tracks memory allocation and memory dereferences. It also adds the calls that run the symbolic execution in parallel with the concrete execution.

4.3.1 Symbolic State

Aside from the concrete state (*i.e.,* the heap and stack), SPLAT tracks symbolic values, allocated memory sizes, and string lengths. Tracking concrete allocated memory sizes allows the detection of memory safety violations, while tracking symbolic values and string lengths allow the generation of new inputs that result in memory safety violations.

For each allocated buffer, SPLAT internally maintains a *memory node* that represents the state of the buffer. The structure of a memory node is shown in Figure 4.3. A memory node tracks the starting address a and ending address b of the buffer, the symbolic size δ of the buffer, the symbolic contents $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{k-1}$ (with k being the preset constant for the length of the symbolic prefix), and (in case the node refers to a string) the symbolic string length ω. Because there exists only one symbolic length ω per memory node, we can only track one string that starts at the beginning of the buffer. In our experiments, we have found no need to track multiple strings per buffer. Memory nodes are kept sorted by address ranges using a splay tree data structure [ST85]. As an optimization, if the symbolic content stored in a buffer is a constant, it is not stored and we rely on the concrete value.

Memory nodes are created whenever a local variable is allocated on the stack (e.g., when a function is called) or when memory is allocated in the heap. Figure 4.4 shows the instrumentation that is added to the program to create memory nodes and to check memory dereferences for possible errors. The function $create(a, b, sz)$ creates a new memory node that starts at the concrete address a and ends at the concrete address b with a (possibly symbolic) size sz. Notice that for allocations, the compiler may choose to allocate more memory for alignment but we track the most conservative allocation. In case the allocated memory

is a string, the field ω of the memory node is set to a fresh symbolic constant. Recall that μ is the symbolic memory map, and $\mu(s)$ returns a possibly symbolic expression by evaluating s in the symbolic state. The function $find(p)$ finds the memory node associated with an address p in the splay tree, if one exists. The function $delete(mn)$ deletes the memory node mn from the splay tree. Memory nodes are removed from the splay tree when memory is freed on the heap or when stack variables go out of scope on a function return.

Symbolic Execution. Symbolic expressions arise from inputs. Following our description of test generation in section 4.2, we define an input through the `input(ptr, k)` function where `ptr` points to the beginning of an input buffer and `k` is the number of elements in the input. This adds mappings to fresh symbolic values in the contents of the memory node that `ptr` points to. As these elements are accessed and modified, other memory nodes are updated with symbolic expressions.

The symbolic updates occur as described in Section 4.2. However we need to take into account some specific features of C: memory allocation and string manipulations. We replace the functions in `string.h` with our string library that is aware of symbolic lengths and memory nodes. The `string.h` functions are modified to update the symbolic length of strings. This is similar to how CSSV [DRS03] symbolically executes string manipulation functions. Figure 4.5 show the updates for some widely used string manipulation functions.

The `strlen(s)` function makes the contents of the return value to be the symbolic string length. The symbolic length is not simply the field ω of the memory node that holds the string, because `s` may not point to the beginning of the memory node a. The function `strcpy(d,s)` copies a string `s` to `d`, so the symbolic length of string `d` is updated with the length of string `s`. Again, offsets

$$
\begin{array}{lcl}
\text{l = strlen(s)} & \rightarrow & mn = find(s) \\
 & & \mu[\&l \mapsto mn.\omega - mn.a + s] \\
 & & \\
\text{strcpy(d,s)} & \rightarrow & mn_d = find(d); mn_s = find(s) \\
 & & mn_d.\omega = d - mn_d.a + mn_s.\omega \\
 & & \qquad -s + mn_s.a \\
 & & assert(mn_d.\omega < mn_d.\delta) \\
 & & \\
\text{strcat(d,s)} & \rightarrow & mn_d = find(d); mn_s = find(s) \\
 & & mn_d.\omega = mn_d.\omega + mn_s.\omega \\
 & & \qquad -s + mn_s.a \\
 & & assert(mn_d.\omega < mn_d.\delta) \\
 & & \\
\text{sprintf}(d, c, s_1 ... s_n) & \rightarrow & \forall 1 \leq i \leq n.\ mn_i = find(s_i) \\
 & & mn_d = find(d) \\
 & & mn_d.\omega = d - mn_d.a + strlen(c) \\
 & & \qquad + \sum_{i=1}^{n} mn_i.\delta - i + mn_i.a \\
 & & assert(mn_d.\omega < mn_d.\delta)
\end{array}
$$

Figure 4.5: Tracking memory for string operations.

with relation to the starting address of the memory nodes are added to take account of d or s not being at the starting addresses of their respective memory nodes. The function `strcat(d,s)` appends the string `s` to the end of the string `d` so after the operation the symbolic length of string `d` is the sum of the length of `d` and the length of `s`.

The function `sprintf(d,c,`$s_1 ... s_n$`)` creates a string at `d` from a format string c and some string parameters $s_1 ... s_n$. We do not write all cases for the update of the length of d, because there are many cases for calculating the length of a C format string. Instead, we show a simplified update where length of d is the sum of the string length of c plus the lengths of the parameters.

4.3.2 Test Generation

As discussed in Section 4.2, SPLAT explores all paths by iteratively finding satisfying assignments for new path constraints representing unexplored paths. In the implementation, each path is a sequence of integers representing branches taken. Each branch id is mapped to a symbolic expression (if is not a constant) representing the predicate associated with taking or not taking the branch. A trie [Knu97] stores all previously explored paths and whether negation of a branch is unsatisfiable or has already been explored. This allows SPLAT to use different search strategies. For example, a depth-first search only requires storing one path in the trie. The search terminates when all paths are either unsatisfiable or have been explored.

Memory Safety. The splay tree of memory nodes track all allocated memory in the heap and on the stack. For each pointer dereference `*p`, we should be able to find, by calling *find*(`p`), the memory node that contains the pointer in the splay tree. If we do not find a memory node, we have a memory safety violation. This approach is similar to Valgrind's Memcheck [NS07]. However, unlike Memcheck, since the return address on the stack is not part of any memory node, we can always detect buffer overflows that overwrite the return address. Note that if we do not track symbolic state, SPLAT can be used as a runtime checker for memory violations.

In addition, we explicitly add assertions about well-formedness of memory in the code using the function *assert*(e). If during test generation, we can find a satisfying assignment for the conjunction of the path constraints with the symbolic expression $\neg e$, we have found a potential error. Since symbolic execution can be imprecise, such a satisfying assignment is subsequently run as a new input to confirm the bug and exhibit a concrete execution trace to the user.

For example in Figure 4.4, when we dereference a pointer, we generate the assertion $assert(p \leq mn.a + mn.\delta)$, where mn is the memory node for p. Failure of this assertion indicates an input for which the pointer p points beyond its memory node (and hence a memory error). This is a stricter approach to memory safety by insuring the dereference occurs in the memory node pointed to by the referent [JK97]. The *referent* refers to the valid address in the expression to be dereferenced. For example in `*(ptr + 5)`, `ptr` is the referent. If `ptr` points to a character buffer of size 3, `*(ptr + 5)` will always be a violation of the referent notion of memory safety. However, `ptr + 5` may still be a valid address; if the buffer is on the stack, `ptr + 5` can point to some other variable allocated on the stack.

Similar checks are performed for string operations as seen in Figure 4.5. Whenever a string is copied into another buffer, a check is made to see if the string length will exceed the size of the buffer. Whenever `sprintf` is called, the length of the generated string is checked to see if it fits in the destination buffer.

4.3.3 Constraint Solving

We generate new inputs by finding satisfying assignments for path constraints and constraints representing memory violations. We use STP, a bit accurate SAT based decision procedure [GD07a]. This allows us to deal with widely-used bit operators and arithmetic overflow. In our experience, arithmetic overflow has been crucial in generating many memory safety violations.

A satisfying assignment for a symbolic length may go beyond the symbolic prefix. Concrete buffer entries beyond the symbolic prefix are randomly chosen characters (excluding the string terminator). Further we add additional constraints that make the symbolic length consistent with the symbolic prefix. The

```
01 GETSHORT (s, cp) { \
02    register u_char *t_cp = (u_char *)(cp); \
03    (s) = ((u_int16_t)t_cp[0] << 8) \
04     | ((u_int16_t)t_cp[1]) \
05     ; \
06    (cp) += INT16SZ; \
07 }
08
09 GETLONG (l, cp) { \
10    register u_char *t_cp = (u_char *)(cp); \
11    (l) = ((u_int32_t)t_cp[0] << 24) \
12     | ((u_int32_t)t_cp[1] << 16) \
13     | ((u_int32_t)t_cp[2] << 8) \
14     | ((u_int32_t)t_cp[3]) \
15     ; \
16    (cp) += INT32SZ; \
17 }
```

Figure 4.6: [Example] Buffer overflow due to arithmetic overflow

occurrence of the string terminator in the symbolic prefix affects the symbolic length. Given a symbolic string length α_k for the symbolic prefix $\alpha_0\alpha_1 \ldots \alpha_{k-1}$, we have the added constraints $\alpha_i = 0 \Rightarrow \alpha_k = i$ for $0 \leq i < k$.

Figures 4.7 and 4.6 show a buffer overflow bug originating in the Bind DNS server that demonstrates the need for a bit-accurate constraint solver. The bug is caused by an arithmetic overflow in line 34. If `dlen - n` < 0, a huge amount would be copied. This example requires the analysis to understand bit operators, pointer arithmetic, and fixed-sized integers. To test this example, we create a symbolic buffer with 100 symbolic values of size 1 byte each. Since we fill the `msg` buffer entirely with symbolic values, the symbolic string length is not tracked. Note that we could have made the two strings within the message have symbolic lengths and saved many extra executions, but that would require knowledge of the internals of `rrextract`.

Instead of listing all runs, we examine a run that reaches line 34. At line 34, suppose for the given run $n = 13$, `cp` is 0x40232504, `eom` is 0x40232568,

```
18 void rrextract(char *msg, int msglen) {
19  int len, n;
20  short type, dlen;
21  char *eom, *cp, expanded;
22  char data[MAXDATA*2];
23  eom = msg + msglen; cp = msg;
24  n = strlen (cp); if (n > 15) return;
25  cp += n;  len += n;
26  GETSHORT(dlen, cp);
27  cp += 2;  len += 2;
28  if (cp + dlen > eom) return;
29  GETSHORT(type, cp);
30  cp += 2;  len += 2;
31  if (type !=  T_NXT) return;
32  n = strlen(cp); if (n > 15) return;
33  cp += n;  cp1 = data;
34  memcpy(cp1, cp, dlen - n);  // overflow
35  cp += (dlen - n);  cp1 += (dlen - n);
36 }
```

Figure 4.7: [Example] Buffer overflow due to arithmetic overflow

$\mathsf{dlen} = \alpha_i << 8 \mid \alpha_{i+1}$ and $\mathsf{type} = \alpha_i << 8 \mid \alpha_{i+1}$. We want to find a satisfying assignment to $(\alpha_i << 8 \mid \alpha_{i+1} - 13 >_{unsigned} 1024)$ in conjunction with the path constraint $(\alpha_j << 8 \mid \alpha_{j+1} = T\_NXT) \wedge (0x40232504 + \alpha_i << 8 \mid \alpha_{i+1} \leq_{unsigned} 0x40232568)$. Note that we must distinguish between the unsigned less-than and the signed less-than operators. Given a decision procedure for bit-vectors, a satisfying assignment can be found: for example $\mathsf{dlen} = 12$ results in a 2GB `memcpy`. In this example, if the underlying constraint solver was not bit accurate, the error would be missed.

4.4 Evaluation

We demonstrate Splat on several programs. We ran Splat on benchmarks representing real exploits [ZLL04] and found all bugs except two. Because the benchmarks were not full programs, we also ran Splat on a module in the Snort

	Program	LOC	Prefix	Size	Buggy	Fixed
Real Exploit Benchmarks	Bind 1	2.9K	100	100	0.02s	0.5s
	Bind 2	3.1K	0	2048	0.1s	1.1s
	Bind 3	2.5K	100	100	0.05s	0.1s
	Bind 4	2.7K	0	2048	0.6s	0.6s
	sendmail 1	1.9K	100	100	t/o	t/o
	sendmail 2	2.4K	0	2048	6.8s	t/o
	sendmail 3	2.0K	100	100	18.6s	1m16s
	sendmail 4	2.4K	100	100	0.16s	1m49s
	sendmail 5	2.1K	100	100	t/o	t/o
	sendmail 6	2.2K	100	100	0.04s	1m38s
	sendmail 7	2.8K	100	100	0.05s	18.2s
	WuFTP 1	2.4K	0	2048	0.3s	1.8s
	WuFTP 2	2.6K	0	2048	0.03s	0.03s
	WuFTP 3	2.3K	0	2048	0.4s	0.4s
Programs	snort	1.7K	100	100	13s	33s
	WuFTP	36K	10	2048	4m	4m43s
	nvds	12K	80	80	2m5s	2hr1m52s

Table 4.1: **Experimental Results:** SPLAT bug finding effectiveness. **LOC** is lines of code. **Prefix** is the length of the symbolic prefix in bytes. **Size** is the maximum length of the input string in bytes. **Buggy** is time spend finding the bug. **Fixed** is time spent rerunning the test after fixing the error. t/o means timeout after 2 hours.

intrusion detection system, the WuFTP server, and NVDS, a well-tested flash-based memory system. All tested programs except NVDS had known memory safety bugs. For the case studies, SPLAT found all known bugs and 2 unknown bugs in NVDS.

Table 4.1 shows the results. All experiments were performed on a 2.33GHz Intel Core 2 Duo with 2GB of RAM. Each experiment contains both the program containing the bug and the program with the bug fixed. The numbering of each benchmark corresponds to the same numbering as the paper [ZLL04] describing

	Symbolic Length				
Program	**Prefix**	**Size**	**Buggy**	**Fixed**	**Cov**
`Bind 2`	0	2048	0.1s	1.1s	40%
`Bind 4`	0	2048	0.6s	0.6s	55%
`sendmail 2`	0	2048	6.8s	t/o	59%
`WuFTP 1`	0	2048	0.3s	1.8s	44%
`WuFTP 2`	0	2048	0.03s	0.03s	23%
`WuFTP 3`	0	2048	0.4s	0.4s	39%
`WuFTP`	10	2048	240s	283s	43%

Table 4.2: **Experimental Results:** Comparing the length abstraction with a fully symbolic input string on programs with string manipulations: experiments with length abstraction. **Prefix** is the length of the symbolic prefix in bytes. **Size** is the maximum length of the input string in bytes. t/o means timeout after 2 hours for the benchmarks or 24 hours for the case study. **Cov** is the branch coverage for the testing fixed programs run until completion or timeout.

the benchmarks. We ran SPLAT on both buggy and fixed versions, because enumeration of the buggy program stops when the bug is found and depending on the location of the bug, only a fraction of paths are enumerated. Table 4.1 the time spent to find the bugs in the buggy program, and the time spent enumerating paths in the fixed program.

For each program, the representation of the input string was chosen with the shortest possible symbolic prefix that can find the bug. The maximum size of the input and the size of the symbolic prefix are shown in Table 4.1. If the size of the symbolic prefix equals the maximum size, then the input was fully symbolic. A symbolic prefix of size zero was successful in finding bugs in 6 out of the 14 benchmarks. A symbolic prefix of 10 characters was successful in finding bugs for the WuFTP case study. The other benchmarks did not use the `string.h` library thus requiring the input to be fully symbolic.

For programs that utilized the `string.h` library, we demonstrated how the length abstraction allows directed testing to scale to larger more complex pro-

	Fully Symbolic				
Program	**Prefix**	**Size**	**Buggy**	**Fixed**	**Cov**
`Bind 2`	100	2048	0.61s	32.5s	40%
`Bind 4`	100	2048	t/o	t/o	55%
`sendmail 2`	100	2048	3.4s	t/o	59%
`WuFTP 1`	100	2048	2.9s	22.5s	44%
`WuFTP 2`	100	2048	t/o	t/o	38%
`WuFTP 3`	100	2048	0.7s	1.16s	39%
`WuFTP`	1024	2048	t/o	t/o	29%

Table 4.3: **Experimental Results:** Comparing the length abstraction with a fully symbolic input string on programs with string manipulations: fully symbolic. **Prefix** is the length of the symbolic prefix in bytes. **Size** is the maximum length of the input string in bytes. t/o means timeout after 2 hours for the benchmarks or 24 hours for the case study. **Cov** is the branch coverage for the testing fixed programs run until completion or timeout.

grams. We show that the length abstraction allows directed testing to find errors in Bind 4, WuFTP 2, and the WuFTP case study that could not be found with a fully symbolic input within our given testing budget. We also show that SPLAT with the length abstraction enumerates fewer paths without sacrificing branch coverage.

4.4.1 Finding Memory Safety Violations

Real Exploit Benchmarks. The first 14 rows of Table 4.1 are real exploit benchmarks [ZLL04]. These benchmarks are small stripped down versions of Bind, Sendmail, and WuFTP, specifically designed to test buffer overflow detection tools. These benchmarks were independently developed to be small but realistic and representative of known buffer overflows. They have been shown to substantially challenge dynamic and static buffer overflow detection tools [ZLL04, ZLL05]. In these thorough evaluations, four static detection tools

were no better than randomly guessing buffer overflow warnings for programs with or without such errors. Only one static tool (Polyspace) was marginally better but produced 1 warning for every 12 lines of code. SPLAT successfully found errors in all benchmarks except two, without reporting any false warnings.

The original benchmarks contained inputs that would result in finding the exploit. These inputs were removed and replaced with symbolic inputs. For the 6 benchmarks that exclusively used the `string.h` library, we represented the whole input with only a symbolic length. For all other benchmarks, we represented all characters of the input with 100 symbolic characters.

The Bind programs represent several buffer overflows in the Bind DNS Server. Bind 1 contained a memcpy that had an arithmetic expression in the size argument that could overflow. Bind 2 and 3 had memcpy size arguments that were improperly bound-checked. Bind 4 contained a sprintf without a bounds check. Sendmail represent bugs in the Sendmail email server. Sendmail 1 did not increment a counter as it processed the "¡" character. Sendmail 2 contains a copy to a fixed sized buffer without a bounds check. Sendmail 3 has an index that is not reset after reading a return character. Sendmail 4 does not check the size if a return character is read. Sendmail 5 contains an improper bounds check for sequences of "/". Sendmail 6 contains an arithmetic underflow. Sendmail 7 allows an arbitrary size to be passed as a bound for `strncpy`. WuFTP represent bugs in the WuFTP ftp server. WuFTP 1 and WuFTP 3 contain unchecked strcpy or strcat functions. WuFTP 2 contains an incorrect bounds calculation as shown in Figure 4.2.

All benchmarks finish quickly except Sendmail 1 and 5 which timeout. To reach the bug, Sendmail 5 requires a long string of "/" characters of some particular length while Sendmail 1 requires repeated occurrences of the pattern "¡¿".

Therefore, Sendmail 1 and 5 require the input to be fully symbolic. In either case, the buggy input was difficult to find because both benchmarks require finding a particular long input from millions of inputs that all lead to different paths.

Snort. We tested the "Back Orifice" rootkit detector module in the Snort intrusion system that had a known buffer overflow. Snort modules have well-defined inputs that describe a packet. SPLAT can model this packet as a symbolic buffer. The bug occurs because the length of the packet field is not checked and later used as a bound on a while-loop that reads the contents of the packet. SPLAT quickly finds this buffer overflow.

WuFTP. We tested a version of the WuFTP server with a buffer overflow in the pathname normalization function. The example in Section 4.1 is a simplified version of that bug. Although WuFTP processes packets, the contents of the packets are strings that are interpreted as FTP commands. We test WuFTP by replacing the packet with a symbolic string with a 10 character symbolic prefix and a symbolic length. To skip the parser, we make several keywords in the string concrete and others symbolic. These keywords are defined by the underlying grammar. The details of construction were presented without the memory and string lengths fully tracked in a previous study[MX07]. Running SPLAT on each of these strings finds the error after 240 seconds.

NVDS. NVDS is a non-volatile storage system for flash memory that had been a target of substantial random differential testing [GHJ07]. We tested NVDS on an emulated system in RAM. Testing was different from the previous experiments, because NVDS did not just accept a string as an input. To test NVDS, we formatted the emulated flash, wrote to it three times and read from it once. The parameters for the writes and reads were symbolic. We found overflows in the memory emulating the flash that resulted from an arithmetic overflow in the

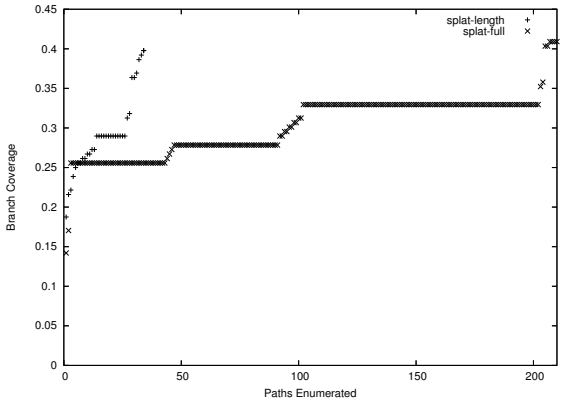

Figure 4.8: Coverage for Bind 2: Splat-length enumerates 34 unique paths in Bind 2 with a random string and a symbolic length. Splat-full must enumerate 210 paths with a symbolic input of 100 bytes.

bound checking in both the write and read functions.

4.4.2 Length Abstractions

For the 6 benchmarks and the 1 case study where string operations were restricted to the C string library, we ran SPLAT with the input string being represented by a small prefix (10 characters for the WuFTP case study and no prefix for the benchmarks) and a symbolic string length. We compared how SPLAT performs when the input string was symbolically represented by its length (Splat-length) with how SPLAT performed with the input string represented by all symbolic characters (Splat-full). Splat-full represents previous work in test generation tools that tries to completely search all paths up to some size input. Splat-full required 100 symbolic characters as the input for the benchmarks and 1024 characters as the input for WuFTP. These sizes were chosen to be slightly greater than the minimum string length that could set off the error, thus giving Splat-full the best chance possible in finding the error. In our experiments, Splat-length performed faster than Splat-full in finding errors as seen in Table 4.3 and Table 4.2— showing

the effectiveness of using a string length abstraction.

In Bind 4 and WuFTP 2, the fully symbolic string technique could not find the bug within 2 hours while Splat-length found both errors in less than a second. In the WuFTP case study, Splat-full could not find the error within 24 hours. In all 3 tests, Splat-full was stuck in generating input not relevant to reaching the error.

For example, Bind 4 contains many `sprintf` of the form, where `buf` is a fixed 1000 byte buffer and all other arguments are inputs:

```
sprintf(buf, "%s: query(%s) %s (%s:%s)",
        sysloginfo, queryname, complaint, dname,
        inet_ntoa(data_inaddr(a_rr->d_data)));
```

Suppose `sysloginfo`, `queryname`, `complaint`, and `dname` are all string inputs with maximum length of 250 while `(data_inaddr(a_rr->d_data))` was a 32-bit integer input. To check for an overflow, Splat-length solves the constraint:

$$\begin{aligned} &strlen(sysloginfo) + strlen(queryname) \\ &+strlen(complaint) + strlen(dname) \\ &+15 + 15 + 1 > 1000 \end{aligned}$$

The constraint has been simplified to include the string length of the address representing the 32-bit integer as 15, the length of the format string as 15, string terminator as 1 and the size of `buf` as 1000.

While Splat-length can solve the string lengths and generate an input that will cause the memory violation, a tool without the length abstraction must rely on a complete search. If we fix the integer input, Splat-full requires placing the string terminator at almost all locations in each of the four strings in the

```
01 char pathspace[ MAXPATHLEN ];
02 char old_mapped_path[ MAXPATHLEN ];
03 char mapped_path[ MAXPATHLEN ] = "/";
04 int mapping_chdir(char *orig_path) {
05   char *path = &pathspace[0];
06   strcpy( old_mapped_path, mapped_path );
07   strcpy( path, orig_path );
08      . . .
09 }
```

Figure 4.9: [Example] WuFTP 1: strcpy at line 07 can overflow

worst case. If each string can have a length of 250, there are around 250^4 or approximately 4 billion strings to enumerate. Furthermore, instead of tracking just 4 symbolic constants representing each string length, 1000 symbolic constants must be tracked — leading to a substantial cost increase in constraint solving. In our Bind 4 experiments, we tried to give Splat-full a better chance of finding the error by reducing the input string sizes such that four input strings of length 24 would result in a buffer overflow, but this also led to a time-out as seen in Table 4.3.

The other experiments involve copying into a buffer without doing a proper check. Figure 4.9 shows a representative memory violation in WuFTP 1 where the `strcpy` on line 14 may overflow because the input is not bound checked before. Splat-length quickly finds these errors by representing the input with just a symbolic length. However, Splat-full can also find the error by enumerating all string lengths up to the maximum length of the input. Because the 6 benchmarks are small snippets of the real exploit, the Splat-full can terminate and find the error. In real programs, directed testing with a fully symbolic input will unlikely find the bug, because it would be stuck enumerating many irrelevant paths of inputs with varying string lengths.

Since a fully symbolic search may get lost in a large search space, Splat-

length also gets better branch coverage, i.e., number of branches explored / total branches, given a limited test budget. As seen in Table 4.2 and Table 4.3, in the WuFTP case study, Splat-length's 283 second search had better coverage than the fully symbolic search timing out after 24 hours. Splat-length reaches higher branch coverage faster than Splat-full. Figure 4.8 shows how branch coverage increases when Splat-length and Splat-full are run on Bind 2. Splat-length only enumerating 34 unique paths results in the same branch coverage as Splat-full enumerating 210 unique paths. As Splat-full is unrolling loops to generate new paths, no new branches are covered. Although it is expected that given substantial resources that Splat-full would get better branch coverage, only in the case of WuFTP 2 did the fully symbolic string approach get better coverage (46/120 versus 28/120). Note that all such experiments are not close to full branch coverage because the benchmarks represent snippets of code from the full program where many paths are unreachable and we do not model all inputs such as networking or configuration options in our WuFTP case study.

4.5 Related Work

Many approaches to detect memory safety violations statically or dynamically have been proposed. Splat combines ideas from several of those with test input generation.

Runtime Checking. Runtime checkers detect memory safety violations of specific execution runs but require a test input to trigger the violation. For such approaches, there is a trade-off between being able to detect the violation and performance. Valgrind [SN05] uses one bit for each address to represent if it is allocated or not. If an invalid address is accessed, a memory safety violation is reported. This only detects accesses to unallocated memory, so the return address

of a function on the stack can still be overwritten and undetected. Jones and Kelly [JK97] implemented a memory safety checker in `gcc` that adds instrumentation to check whether an address evaluated from some expression containing an address p still points to the same buffer as p. CRED [RL04] extends Jones and Kelly by checking bounds only before a memory dereference and focuses only on string operations. If SPLAT was run without input generation, SPLAT would be similar to the CRED approach with additional instrumented libraries.

Larson and Austin [LA03] extends runtime checking by finding errors that may occur along the same control path of the supplied input but with a different input. Their memory model [LA03] associates each buffer index with a range, and each buffer with a string length and buffer size that are updated symbolically. SPLAT tracks similar constraints but is more precise because [LA03] conservatively represents each range and size with an integer instead of a symbolic expression. Also [LA03] does not perform test generation and may generate false alarms when symbolic execution is imprecise.

Static Analysis. In contrast to dynamic analysis, static analysis runs on all paths of a program and does not require any test inputs. However, they typically generate (many) false positives. CSSV [DRS03] converts string manipulation to integer operations and performs an integer analysis to insure string operations remain within proper bounds. Although false positives were reported to be few, manual summaries are needed for functions and the integer analysis was expensive. Archer [XCE03] tracks linear relationships between variables and automatically generates function summaries by inferring relationships between function parameters by various heuristics. Boon [WFB00] uses a flow insensitive analysis for string manipulations errors which is fast but very imprecise. A comparison between various static tools showed that none were very effective in finding real

buffer overflows, either not finding the errors or generating too many false positives [ZLL04]. SPLAT 's symbolic execution of string lengths was inspired by static analyses that track only lengths [DRS03, XCE03, WFB00].

Directed Testing. The idea of path exploration using both symbolic and concrete execution is from directed testing tools such as DART [GKS05], CUTE [SMA05], EXE [CGP06] and SAGE [GLM08]. Recent papers [GLM07, JSS07, CGP06] also suggest to systematically inject assertions in programs during directed test generation in order to detect memory safety violations and other standard programming errors, such as division by zero and integer overflows. Our contribution is to use symbolic length abstraction and symbolic prefixes of input buffers to improve scalability of automatic test generation for buffer overruns.

Underapproximation. To allow SPLAT to effectively find bugs and finish within a reasonable amount of time, SPLAT uses an underapproximation that leaves some input suffixes random but tracks the length of the string input symbolically. In the context of test generation, different underapproximations involving heap shapes have been explored in Java Pathfinder [VPP06]. Our experiments with SPLAT show the new underapproximation using symbolic string lengths and buffer sizes seems to be effective in finding buffer overflows in C programs.

CHAPTER 5

Reducing Test Inputs with Control and Data Dependencies

In this chapter, we explore how to reduce the number of paths explored in concolic execution without sacrificing completeness by usng control and data dependencies. In many applications, the inputs can be partitioned into "non-interfering" blocks such that symbolically solving for each input block while keeping all other blocks fixed to concrete values can find the same set of assertion violations as symbolically solving for the entire input. This can greatly reduce the number of paths to be solved (in the best case, from exponentially many to linearly many in the number of inputs). We present an algorithm that combines test input generation by concolic execution with dynamic computation and maintenance of information flow between inputs. Our algorithm iteratively constructs a partition of the inputs, starting with the finest (all inputs separate) and merging blocks if a dependency is detected between variables in distinct input blocks during test generation. Instead of exploring all paths of the program, our algorithm separately explores paths for each block (while fixing variables in other blocks to random values). In the end, the algorithm outputs an input partition and a set of test inputs such that (a) inputs in different blocks do not have any dependencies between them, and (b) the set of tests provides equivalent coverage with respect to finding assertion violations as full concolic execution.

```
void test(int a_1, int a_2,
          ...,
          int a_n) {
1:      if (a_1 = 0) a_1 := 1;
2:      if (a_2 = 0) a_2 := 1;
        ...
n:      if (a_n = 0) a_n := 1;

n+1: if (a_1 = 0) error();
n+2: if (a_2 = 0) error();
        ...
2n:     if (a_n = 0) error();
}
```

```
00 void free(int A[], int count[]) {
01   for (int i = 0; i < N; i++) {
02     old_count[i] = count[i];
03   }
04   for (int i = 0; i < N; i++) {
05     if (A[i] != 0)
06        count[i]++;
07   }
08   for (int i = 0; i < N; i++) {
09     if (A[i] != 0)
10        assert(count[i]==old_count[i]+1);
11   }
12 }
```

Figure 5.1: [Example] Many independent inputs: (a) Example `test` (b) Example `free`

We develop a technique that exploits the *independence* between different parts of the program input to reduce the number of paths needed to be explored during test generation.

As a simple example, consider the function `test` shown in Figure 5.1(a). While there are 2^n syntactic paths through the code, we quickly recognize that it is sufficient to check only $2 \cdot n$ paths: two each for each of the inputs a_1, a_2, ..., a_n being zero or non-zero. In particular, we conclude that `error()` is not reachable based only on these paths. The additional paths through the program do not add any more "interesting" behaviors, as the inputs are *non-interfering*: there is no data or control dependency between any two distinct inputs a_i, a_j for $i \neq j$. This indicates that by generating tests one independent input at a time (while holding the other inputs to fixed values), we can eliminate the combinational explosion of testing every arrangement of inputs, in this case, from an exponential number of paths (2^n for n inputs) to a linear number ($2 \cdot n$), while retaining all interesting program behaviors (e.g., behaviors that can cause assertion violations or behaviors that lead to an error). While the above example is artificial, there are many interesting examples where the input space can be

partitioned into independent and non-interfering components, either through the application semantics (e.g., blocks in a file system, packet headers in network processors, permission-table entries in memory protection schemes) or due to security and privacy reasons (e.g., independent requests to a server).

We present an automatic test generation algorithm FlowTest that formalizes and exploits the independence among inputs. The main idea of FlowTest is to compute control and data dependencies among variables dynamically while performing concolic execution, and to use these dependencies to keep independent variables separated during test generation.

FlowTest maintains a partitioning of the program inputs (where two variables in different blocks are assumed not to interfere), and generates tests by symbolically treating variables in each block in isolation while holding variables in other blocks to fixed values.

In case the partition does denote non-interfering sets of variables, and all program executions terminate, the test generation is relatively sound: any assertion violation detected by basic concolic execution is detected. To check for data or control dependencies between variables in separate blocks, FlowTest maintains a *flow map* during test generation which associates each variable with the set of input blocks in the current partition which can potentially influence the value of the variable. If there is some entry in the flow map which contains more than one input block, this indicates "information flow" between these input blocks. In this case, these input blocks are merged and test generation is repeated by tracking this larger block of inputs together.

The algorithm terminates when the input partitions do not change (and tests have been generated relative to this input partition). For example `test`, starting with the initial optimistic partition in which each variable is in a separate par-

tition, FlowTest will deduce that this partition is non-interfering, and generate test cases that explore the $2n$ interesting paths. In contrast, concolic execution explores 2^n paths.

We have implemented FlowTest on top of the SPLAT directed testing implementation [XMG08] to test and check information flow in C programs. The benefit of FlowTest is demonstrated on a memory allocator, a memory protection scheme, an intrusion detector module and a packet printer. FlowTest dramatically reduces the number of paths explored for all case studies without increasing much overhead per path due to flow-set generation and dependency checking. In all cases, FlowTest reduced the overall time for input generation and the number of paths generated. In two cases, FlowTest cut input generation in half. In one case, FlowTest terminated in less than ten minutes when the basic concolic execution algorithm failed to terminate even after a day.

Related Work. Test generation using concolic execution has been successfully applied to several large programs [GKS05, SMA05, CGP06, XMG08, CDE08]. However, path explosion has been a fundamental barrier. Several optimizations have been proposed to prune redundant paths, such as function summarization [God07, AGT08] and the pruning of paths that have the same side-effects of some previously explored path through read-write sets (RWSets) [BCE08]. FlowTest is an optimization orthogonal to both function summarization and RWSets.

Program slicing has been used to improve the effectiveness of testing and static analysis by removing irrelevant parts of the program [Wei79, Tip95, KL88, KY94]. One way to view FlowTest is as simultaneous path exploration by concolic execution and dynamic slicing across test runs: for each input block, the algorithm creates dynamic slices over every run, and merges input blocks that have common data or control dependencies. In contrast to running test generation on

statically computed slices, our technique, by computing slices dynamically and on executable traces, can be more precise.

Our optimization based on control and data dependencies is similar to checking information flow [Den76, MPL04, CLO07]. For example, dynamic information flow checkers [MPL04, CLO07] are based on similar dependency analyzes.

5.1 Definitions

We illustrate our algorithm by extending our concolic execution algorithm presented in Chapter 2.

Partitions. A *partition* $\Pi(X)$ of a set X is a set of pairwise disjoint subsets of X such that $X = \bigcup_{Y \in \Pi(X)} Y$. We call each subset in a partition a *block* of the partition. For a variable $x \in X$, we denote by $\Pi(X)[x]$ the block of $\Pi(X)$ that contains x.

Given a partition $\Pi(X)$ and a subset $Y \subseteq \Pi(X)$ of blocks in $\Pi(X)$, the partition $\divideontimes(\Pi(X), Y)$ obtained by merging blocks in Y is defined as $(\Pi(X) \setminus Y) \cup \{\cup_{b \in Y} b\}$.

A partition $\Pi(X)$ *refines* a partition $\Pi'(X)$ if every block in $\Pi'(X)$ is a union of blocks in $\Pi(X)$. In this case we say $\Pi'(X)$ is *as coarse as* $\Pi(X)$. If $\Pi'(X)$ is as coarse as $\Pi(X)$ but $\Pi'(X) \neq \Pi(X)$, we say $\Pi'(X)$ is coarser than $\Pi(X)$. When the set X is clear from the context, we simply write Π.

Control and Data Dependence. For two locations $\ell, \ell' \in \mathcal{L}$ we say ℓ' *post-dominates* ℓ if every path from ℓ to ℓ_{halt} contains ℓ'. We say ℓ' is the *immediate* post-dominator of ℓ, written $\ell' = \mathit{idom}(\ell)$, if (1) $\ell' \neq \ell$, (2) ℓ' post-dominates ℓ, and (3) every ℓ'' that post-dominates ℓ is also a post-dominator of ℓ'. It is known that every location has a unique immediate post-dominator [Muc97], and hence

Algorithm 3: FlowTest

```
Input: Program P, initial partition Π_0(X)
1  local partitions Π and Π_old of X ;
2  local flow map flow;
3  Π := Π_0(X) ;
4  Π_old := ∅ ;
5  flow(x) := {Π[x]} for x ∈ X;
6  while Π_old ≠ Π do
7  |  Π_old := Π;
8  |  for I ∈ Π_old do
9  |  |  input := λx ∈ X_0.random();
10 |  |  flow := Generate(P, Π, I, flow, input, −1);
11 |  end
12 |  for each x ∈ X do
13 |  |  Π := 𝕄(Π, flow(x));
14 |  end
15 end
```

the function $idom(\ell)$ is well-defined for every $\ell \neq \ell_{\mathsf{halt}}$.

A node ℓ is *control dependent* on ℓ' if there exists some executable path $\ell_0 \ldots \ell' \ldots \ell$ such that $idom(\ell')$ does not appear between ℓ' and ℓ in the path.

For an expression e, we write $\mathsf{Use}(e)$ for the set of variables in X occurring in e. For variables $x, y \in X$, we say x is *data dependent* on y if there is some executable path to a location ℓ such that $op(\ell)$ is $x := e$ and $y \in \mathsf{Use}(e)$.

5.2 The FlowTest Algorithm

Overall Algorithm.

We illustrate our algorithm by extending our concolic execution algorithm presented in Chapter 2. The algorithm is described on the same imperative language from Chapter 2.

Algorithm 3 shows the overall FlowTest algorithm. It takes as input a pro-

Algorithm 4: Generate

```
   Input: Program P, partition Π, block I ∈ Π, flow map flow
   Input: input input, last explored branch last
 1 (ξ, flow) := Execute(P, Π, I, flow, input);
 2 index := Length(ξ) − 1;
 3 while not empty(ξ) ∧ index ≠ last do
 4 |   p := pop(ξ);
 5 |   if ξ ∧ ¬p is satisfiable then
 6 |   |   input := Solve(ξ, ¬p);
 7 |   |   flow := Generate(P, Π, I, flow, input, index);
 8 |   index := index − 1;
 9 end
10 return flow;
```

gram P and an initial partition $\Pi_0(X)$ of the set X of variables, and applies test generation with iterative merging of partitions. It maintains a "current" partition Π of the inputs, which is updated based on control and data dependence information accrued by test generation. The "old" partition Π_{old} is used to check when a round of test generation does not change the partition. Initially, Π is the input partition $\Pi_0(X)$, and Π_{old} is the empty-set (lines 3, 4).

The main data structure to store dependency information is called a *flow map*, $flow : X \to 2^{\Pi}$, a function mapping each variable $x \in X$ to a set of blocks of the current partition Π. Intuitively, $flow(x)$ denotes the set of input blocks that are known to influence (through data or control dependence) the value of x. Initially, we set $flow(x) = \{\Pi[x]\}$ for each $x \in X$ (line 5).

The main loop (lines 6–14) implements test generation and iterative merging of partitions. The procedure Generate (described next) implements a standard path exploration algorithm using concolic execution, but additionally updates the flow map. Generate is called to generate tests for each block I of the current partition Π (lines 8–11). In each call, the variables in the block I are treated sym-

bolically, and every other variable is given a fixed, random initial value. Generate returns an updated flow map which is used to merge blocks in Π to get an updated partition (lines 12–14). For every $x \in X$ such that $|\mathit{flow}(x)| > 1$, the blocks in $\mathit{flow}(x)$ are merged into one block to get a new coarser partition. The main loop is repeated with this coarser partition until there is no change.

Algorithm Generate**.** Algorithm 4 describes the path enumeration algorithm, and is similar to the path enumeration schemes in [GKS05, SMA05, CGP06]. It takes as input the program P, a partition Π of the inputs of P, a block I of the partition, an input *input* mapping input variables to initial values, and an index *last* tracking the last visited branch. It performs test case generation using a depth first traversal of the program paths using concolic execution. In the concolic execution, only inputs from I are treated symbolically and the rest of the inputs are set to fixed concrete values (chosen randomly). The procedure returns an updated flow map.

The main loop of Algorithm Generate implements a recursive traversal of program paths. In each call to Generate, the function Execute (described next) is used to perform concolic execution along a single path and update the flow map. The returned path constraint ξ is used to generate a sequence of new inputs in the loop (lines 3–9). This is done by popping and negating the last constraint in the path constraint and generating a new input using procedure Solve. The new input causes the program to explore a new path: one that differs from the last one in the direction of the last conditional branch. The function Generate is recursively invoked to explore paths using this new input (line 7).

Notice that the pure concolic execution algorithm is captured by the call Generate$(P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1)$.

Algorithm Execute**.** Algorithm 5 adds computing data and control dependencies

Algorithm 5: **Execute**

Input: Program P, partition Π, block $I \in \Pi$, flow map *flow*, input i
Result: Path constraint ξ and updated flow map *flow*

```
1  for x ∈ X do
2  |  M(x) := i(x); if x ∈ I then μ(x) := α_x;
3  end
4  ξ := emptyStack; ℓ := ℓ_0;
5  Ctrl(ℓ_0) := ∅;
6  while op(ℓ) ≠ halt do
7  |  switch op(ℓ) do
8  |  |  case l := e
9  |  |  |  M := M[l ↦ M(e)]; μ := μ[l ↦ μ(e)]; ℓ := N(ℓ);
10 |  |  |  Ctrl(N(ℓ)) := Ctrl(ℓ) \ RmCtrl(Ctrl(ℓ), N(ℓ));
11 |  |  |  flow(l) := flow(l) ∪ ⋃_{x∈Use(e)} flow(x) ∪ ⋃_{⟨ℓ',x'⟩∈Ctrl(N(ℓ))} flow(x');
12 |  |  end
13 |  |  case if(x)then ℓ' else ℓ''
14 |  |  |  if M(x) = 0 then
15 |  |  |  |  ξ := push(μ(x) = 0, ξ); ℓ := ℓ'';
16 |  |  |  |  Ctrl(ℓ'') := (Ctrl(ℓ) ∪ {⟨ℓ, x⟩}) \ RmCtrl(Ctrl(ℓ) ∪ {⟨ℓ, x⟩}, ℓ'');
17 |  |  |  |  flow(x) := flow(x) ∪ ⋃_{⟨ℓ̂,y⟩∈Ctrl(ℓ'')} flow(y);
18 |  |  |  else
19 |  |  |  |  ξ := push(μ(x) ≠ 0, ξ); ℓ := ℓ';
20 |  |  |  |  Ctrl(ℓ') := (Ctrl(ℓ) ∪ {⟨ℓ, x⟩}) \ RmCtrl(Ctrl(ℓ) ∪ {⟨ℓ, x⟩}, ℓ');
21 |  |  |  |  flow(x) := flow(x) ∪ ⋃_{⟨ℓ̂,y⟩∈Ctrl(ℓ')} flow(y);
22 |  |  |  |
23 |  |  end
24 |  end
25 end
26 return(ξ, flow);
```

along a single program path to concolic execution. It takes as input the program P, a concrete input i, and a partition Π. It returns a path constraint ξ and an updated flow map *flow*. Notice that the path constraint is maintained as a stack of predicates (instead of as a conjunction of predicates). This helps in simplifying the backtracking search in Generate.

Algorithm Execute, ignoring lines 10, 11, 16, 17, 20 and 21, is identical to

the concolic execution algorithm discussed in Chapter 2 It executes the program using both the concrete memory M and the symbolic memory μ. The extra lines update the flow map.

We now describe the working of Algorithm Execute. The concrete memory is initialized with the concrete input i, and the symbolic memory is initialized with a fresh symbolic constant α_x for each $x \in I$ (lines 1–3). The path constraint is initialized to the empty stack and the initial control location is ℓ_0 (line 4).

The main loop of Execute (lines 6–25) performs concrete and symbolic evaluation of the program while updating the flow map. The loop executes while the program has not terminated (or, in practice, until some resource bound such as the number of steps has been exhausted).

We first ignore the update of the flow map and describe how the concrete and symbolic memories and the path constraint are updated in each iteration.

If the current location is ℓ and the current operation is an assignment $l := e$ (lines 8–12), the concrete memory updates the value of l to $M(e)$ and the symbolic memory updates it to $\mu(e)$ (line 9). Finally, the control location is updated to be $N(\ell)$.

If the current location is ℓ and the current operation is $\mathsf{if}(x)$ **then** ℓ' **else** ℓ'', the updates are performed as follows (lines 13–23). The concrete memory M and symbolic memory μ remain unchanged. If $M(x) \neq 0$, then the constraint $\mu(x) \neq 0$ is pushed on to the path constraint stack ξ, and the new control location is ℓ' (line 19). If $M(x) = 0$, then the constraint $\mu(x) = 0$ is pushed on to the path constraint stack ξ, and the new control location is ℓ'' (line 15).

We now describe how the control dependencies and the flow maps are updated. We use a helper data structure Ctrl mapping each location to a set of pairs of locations and expressions. This data structure is used to main-

tain the set of conditionals on which a location is control dependent along the current execution. At the initial location ℓ_0, we set $\mathsf{Ctrl}(\ell_0) = \emptyset$ (line 5). Each statement updates the set Ctrl. We use the following definition. Let $L \subseteq \mathcal{L} \times X$ be a set of pairs of locations and variables. Let $\ell \in \mathcal{L}$. We define $\mathsf{RmCtrl}(L, \ell) = \{\langle \ell', x\rangle \in L \mid \ell \text{ is the immediate post-dominator of } \ell'\}$. Intuitively, these are the set of conditionals that on which ℓ is no longer control dependent.

Suppose the current location is ℓ and $op(\ell)$ is the assignment $l := e$. The set $\mathsf{Ctrl}(N(\ell))$ consists of the set $\mathsf{Ctrl}(\ell)$ minus the set of all locations which are immediate post-dominated by $N(\ell)$ (line 10). The flow map for the variable l is updated as follows (see line 11). There are three components in the update. The first component is $\mathit{flow}(l)$, the flow computed so far. The second component $\bigcup_{x \in \mathsf{Use}(e)} \mathit{flow}(x)$ captures data dependencies on l due to the assignment $l := e$: for each variable x appearing in e, it adds every input block known to influence x (the set $\mathit{flow}(x)$) to $\mathit{flow}(l)$. The third component captures dependencies from controlling conditionals. The controlling conditionals for $N(\ell)$ and their conditional variables are stored in $\mathsf{Ctrl}(N(\ell))$. For every $\langle \ell', x'\rangle \in \mathsf{Ctrl}(N(\ell))$, we add the set $\mathit{flow}(x')$ of inputs known to influence x' to $\mathit{flow}(l)$.

Now suppose the current location is ℓ and $op(\ell)$ is $\mathsf{if}(x)$ **then** ℓ' **else** ℓ''. In this case, depending on the evaluation of the conditional x, the execution goes to ℓ' or ℓ'' and the corresponding data structure $\mathsf{Ctrl}(\ell')$ or $\mathsf{Ctrl}(\ell'')$ is updated to reflect dependence on the conditional x (lines 16, 20). The pair $\langle \ell, x\rangle$ is added to the set of controllers to indicate that the conditional may control execution to ℓ' and ℓ'', and as before, the set of conditionals post-dominated by ℓ' (respectively, ℓ'') are removed. Finally, for each $\langle \hat{\ell}, y\rangle$ in $\mathsf{Ctrl}(\ell')$ (respectively, $\mathsf{Ctrl}(\ell'')$), the set of input blocks $\mathit{flow}(y)$ is added to the flow map for x.

The updated flow map is returned at the end of the loop.

Algorithm solve**.** Finally, procedure solve takes as input a stack of constraints ξ and a predicate p, and returns a satisfying assignment of the formula

$$\bigwedge_{\phi \in \xi} \phi \wedge p$$

using a decision procedure for the constraint language. In the following, we assume that the decision procedure is *complete*: it always finds a satisfying assignment if the formula is satisfiable.

Relative Soundness. As we have seen, FlowTest can end up exploring many fewer paths than pure concolic execution (i.e., the call Generate$(P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1)$). However, under the assumption that all program executions terminate, we can show that FlowTest is relatively sound: for every location ℓ, if concolic execution finds a feasible path to ℓ, then so does FlowTest. In particular, if all program executions terminate, then FlowTest does not miss any assertion violation detected by concolic execution.

We say a location $\hat{\ell}$ is *reachable* in FlowTest(P, Π_0) or Generate$(P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1)$ if the execution reaches line 7 of Execute with $\ell = \hat{\ell}$. Clearly, a location reachable in either algorithm is reachable in the CFG by an executable path.

Theorem 1. *Let $P = (X, X_0, \mathcal{L}, \ell_0, op, E)$ be a program and Π_0 an initial partition of the inputs of P. Assume P terminates on all inputs. If* FlowTest(P, Π_0) *terminates then every location $\ell \in \mathcal{L}$ reachable in* Generate$(P, \{X\}, X, \lambda x.\{X\}, \lambda x.0, -1)$ *is also reachable in* FlowTest(P, Π_0)*.*

We sketch a proof of the theorem. We prove the theorem by contradiction. Suppose that there is a location ℓ that is reachable in concolic execution but not

in FlowTest. Fix a path $\pi = \ell_0 \to \ell_1 \to \ldots \to \ell_k$ executed by concolic execution such that ℓ_k is not reachable in FlowTest but each location $\ell_0, \ldots, \ell_{k-1}$ is reachable by FlowTest. (If there are several such paths, choose one arbitrarily.) Notice that since π is executed by concolic execution, the path constraints resulting from executing π is satisfiable. Also, $op(\ell_{k-1})$ must be a conditional, since if it were an assignment and ℓ_{k-1} is reachable in FlowTest, ℓ_k would also be reachable in FlowTest.

Since every program execution terminates, we can construct a *path slice* [JM05] of π, i.e., a subsequence π' of operations of π, with the following properties: (1) every initial memory M_0 that can execute π can also execute π', and (2) every initial memory M_0 that can execute π' is such that there is a (possibly different) program path π'' from ℓ_0 to ℓ_k such that M_0 can execute π''. Such a slice can be computed using the algorithm from [JM05]. Since the path constraint for π is satisfiable, so is the path constraint for π'. Let $\mathsf{V}(\pi')$ be the set of variables appearing in π', i.e., $\mathsf{V}(\pi')$ is the smallest set such that for every operation $l := e$ in π' we have $\{l\} \cup \mathsf{Use}(e) \subseteq \mathsf{V}(\pi')$ and for every operation $\mathsf{if}(x)$ **then** ℓ' **else** ℓ'' in π', we have $x \in \mathsf{V}(\pi')$.

We show that each variable in $\mathsf{V}(\pi')$ is eventually merged into the same input block. We note that every conditional operation in π' controls the next operation in π' and every assignment operation $l := e$ in π' either uses l in a subsequent assignment or in a conditional. Now, since every location along π' is reachable, we can show (by induction on the length of π') that all variables in $\mathsf{V}(\pi')$ are merged into the same input block. Call this block I.

Consider the call to Generate made by FlowTest with the input block I. This call is made by FlowTest in the iteration after I is created. Since π' is a path slice of π, and π is executable, we know that the sequence of operations in π' can be

executed by suitably setting values of variables in I, no matter how the rest of the variables are set. Moreover, since the path constraint for π is satisfiable, so is the path constraint for π'. Since every execution terminates, the backtracking search implemented in Generate is complete (modulo completentess of the decision procedure), so the call to Generate using input block I must eventually hit ℓ_k. This is a contradiction, since this shows ℓ_k is reachable in FlowTest.

While the theorem makes the strong assumption that all program paths terminate, in practice this is ensured by setting a limit on the length of paths simulated in concolic execution. In fact, if the program has an infinite execution, then the concolic execution algorithm does not terminate: it either finds inputs that show non-termination (and gets stuck in Execute), or finds an infinite sequence of longer and longer program execution paths.

5.3 Example

We illustrate the working of FlowTest on a simplified version of a virtual memory page-free routine in an OS kernel (the actual code was tested in our experiments). Figure 5.1(b) shows the `free` function which takes as input two arrays of integers `A` and `count` each of size N, a fixed constant. For this example, let us set $N = 2$. For readability, we use C-like syntax and an array notation as a shorthand for declaring and using several variables (our simple language does not have arrays, but our implementations deals with arrays).

Notice that the loop in lines 4–7 of the function `free` in has 2^N paths, because the conditional on line 5 could be true or false for $0 \leq i < N$. So, even for small N, concolic execution becomes infeasible. For example, our implementation of concolic execution in the tool SPLAT [XMG08] already takes one day on an Intel

Core 2 Duo 2.33 Ghz machine when $N = 20$.

Now we show how FlowTest can reduce the cost of testing this function. Our intuition is that the behavior of one "page" in the system is independent of all other pages, so we should test the code one page at a time. Concretely, in the code, there is no control or data dependency between two separate indices of the arrays. For example, the value $A[0]$ is not control or data dependent on $A[1]$, and $count[0]$ is not control or data dependent on $count[1]$. However, there is dependence between $A[i]$ and $count[i]$ (through the conditional in lines 5–6).

Initially, we start with the optimistic partition

$$\{\ \{A[0]\},\ \{A[1]\},\ \{count[0]\},\ \{count[1]\}\}$$

in which each variable is in its own partition. Consider the run of Generate when $A[0]$ is treated symbolically, but all other variables are fixed to constant values. The concolic execution generates two executions: one in which $A[0]$ is 0 and a second in which $A[0]$ is not zero. For the run in which $A[0]$ is not zero, moreover, the flow map is updated with the entry:

$$\mathit{flow}(count[0]) = \{\{A[0]\}, \{count[0]\}\}$$

since the assignment to $count[0]$ is control dependent on the predicate $A[0] \neq 0$. Thus, the blocks $\{A[0]\}$ and $\{count[0]\}$ are merged. In a similar way, the blocks $\{A[1]\}$ and $\{count[1]\}$ are merged.

In the next iteration of the main loop of FlowTest, the function Generate generates tests individually for the blocks $\{A[0], count[0]\}$ and $\{A[1], count[1]\}$. This time, there is no additional merging of input blocks. Hence, the algorithm terminates. The assertion holds on all paths of the resulting test set. The relative

soundness result implies that the assertions hold for every execution of `free`.

5.4 Evaluation

5.4.1 Implementation

We have implemented a subset of the FlowTest algorithm to generate inputs for C programs on top of the Splat concolic execution tool [XMG08]. In our tool, we take as input a program and a manual partition of the inputs, and implement the test generation and dependence checking part of the algorithm (lines 8–11 in Algorithm 3). However, instead of merging input blocks and iterating, our tool stops if two different input blocks must be merged (and assumes that the merging is performed manually). Our experiments (described below) confirm the following hypotheses:

- The performance impact of the added checks for dependencies is small, and more than compensated by the reduction in the number of paths explored.
- A tester can find non-interfering input partitions by superficially examining the source code.

Our implementation is divided in four main components: control flow generation, tracking flow, symbolic execution, and test generation. We use CIL [NMR02] to instrument C code and to statically build the post-dominators (assuming every loop terminates) using the standard backwards dataflow algorithm [Muc97]. We use the concolic execution and test generation loop of Splat to additionally track information flow between input partitions. Our implementation handle function calls and dynamic memory allocation in the standard way [GKS05, SMA05]. In our experiments, we did not put a priori bounds on the lengths of executions,

but ran each function to completion.

Functions. Statement labels are made to be unique across all functions, therefore, entering and exiting a function during an execution is similar to in-lining the function at each call site.

Pointers and Memory. Given a branch label ℓ, we track all writes from that branch label to the label that post-dominates ℓ. Because our algorithm executes the code, address resolution is done during runtime. This implies we do not require a static alias analysis or suffer from the imprecision associated with static alias analysis, especially in the presence of address arithmetic or complex data structures.

We merge all dynamically allocated memory by allocation site when computing data dependencies, but distinguish each memory location individually when performing concolic execution. This can merge different potentially independent locations into the same block, but does not introduce imprecision in concolic execution. For statically- and stack-allocated variables, each byte is distinguished individually. For multiple writes to the same address from the branch label ℓ to the immediate post-dominator of ℓ, we take the union of all blocks that flow into that address. This limits the memory tracked to be (stack size $\times$ static allocation points) per branch node and is conservative because writes to one allocation point flow into all dynamic memory allocated from that point. Experientially, we found that this trade off did not lead to false alarms and was sufficient to check that blocks did not interfere with each other – mostly because each field in a data structure is distinguished.

Performance. Experimentally, we found that the extra information stored and checked along each path results in doubling the time taken to generate and explore a path. However, this increase in the per path time was more than compensated

by the reduction in the number of paths explored. The static analysis cost (to compute post-dominators) is negligible (< 1 second for each of our case studies). While we did not experiment with automatic merging of input blocks, we expect there would not be a significant performance hit because finer grain blocks would have few paths and merging would occur quickly.

5.4.2 Case Studies

We have compared our algorithm with SPLAT on the following case studies. `pmap` is the code managing creation and freeing of a new processes in a virtual memory system for an educational OS kernel (1.6 KLOC), `mondrian` is the insertion module in a memory protection scheme (2.6 KLOC), `snort` is a module in an intrusion detection system (1.7 KLOC), and `tcpdump` consists of eight printers in a packet sniffer (12 KLOC).

In all programs selected, it was possible to come up with a good partition of the input with only a cursory examination of the source code (even though we did not write the code ourselves). In `pmap`, each header representing a physical page is a separate block. In `mondrian`, the protection bits could be partitioned from all other inputs. In both of the packet processing programs `snort` and `tcpdump`, various parts of the packet header could be partitioned. For `tcpdump`, we looked at 10 different packet types that could be printed by `tcpdump`. The partition for all those packet types were destination address, source address and everything else.

The implementation automatically generated inputs to check if there was interference between the manually chosen partitions for each program. Table 5.1 shows the results. All experiments were performed on a 2.33 GHz Intel Core 2 Duo with 2 GB of RAM. The two columns compare partitioning with concolic

				FlowTest		Splat	
Program	**Input**	**Blocks**	**Cov**	**Paths**	**Time**	**Paths**	**Time**
`pmap`	1024	512	42%	1536	8m29s	6238	>1hr
`mondrian`	12	2	30%	733	23m20s	2916	36m35s
`snort`	128	2	85%	45	4m44s	73	4m55s
`tcpdump`	128	3	18%	1247	34m1s	4974	53m46s

Table 5.1: **Experimental Results:** Comparing FlowTest and Splat . **Input** is the size of the symbolic input buffer in bytes. **Blocks** is the number of blocks into which the input was (manually) partitioned. **Cov** is branch coverage for both FlowTest and Splat . **Paths** is the number of unique paths explored. **Time** is the time taken up to a maximum of one hour.

execution. The **Input Size** is the size of the symbolic input buffer used for both implementations. **Blocks** is the number of input blocks found manually and used for FlowTest. **Paths** is the number of unique paths found. **Coverage** is the number of branches covered. For all experiments, we could not achieve 100% coverage because we were only testing a specific part of the program. While these partitions may not be the best partition, FlowTest finished in half the time of Splat in two cases. For `pmap`, FlowTest finished in less than 10 minutes. In contrast, Splat did not finish in one day (Table 5.1 shows the progress made by Splat after one hour (6238 paths).

We did not compare our algorithm with combinations of other optimizations such as function summaries [God07, AGT08] or RWSets [BCE08], as these techniques are orthogonal and can be combined to give better performance.

Limitations. In several examples, we found artificial dependencies between inputs caused by error handling code. For example, the `snort` module first checks if a packet is an UDP packet, and returns if not:

```
if(!PacketIsUDP(p)) return;
```

This introduces control dependencies between every field checked in `PacketIsUDP` and the rest of the code. However, these fields can be separated into their own blocks if the `PacketIsUDP(p)` predicate holds. A second limitation is our requiring that inputs are *partitioned*. In many programs, there is a small set of inputs that have dependencies with all other inputs, but the rest of the inputs are independent. Thus, it makes sense to divide the inputs into subsets whose intersections may not be empty.

CHAPTER 6

Non-Termination

Although there has been much work in proving programs always terminate [CS02, BMS05, Cou05, CPR06], there has been little work that find non-terminating executions. One way of finding non-terminating program executions is to find *feasible lassos*. A *lasso* is a finite program path called *stem* followed by a finite program path called a *loop*. A lasso is *feasible* if an execution of the stem can be followed by infinitely many executions of the loop. In general, the method is incomplete, as not all non-terminating program executions are lassos because the infinite behavior may be non-periodic. However, lassos describe the most common non-termination bugs.

The algorithm proceeds in two phases. The first phase generates candidate lassos, and the second checks each lasso for possible non-termination. Candidate lassos can be generated exhaustively by systematically executing all paths of the concrete program until a control location repeats. Backtracking is controlled by collecting symbolic constraints during concrete program execution. This method has two advantages. First, the stem followed by one execution of the loop is guaranteed to be feasible. Second, every such lasso is guaranteed to be generated. Alternatively, a termination prover could be used to supply a candidate lasso whenever the proof fails. The second phase proves the feasibility of a given lasso. A lasso is feasible if and only if there exists a *recurrent* set of states, a set of states that is visited infinitely often along the infinite path that results from unrolling

the lasso. The existence of a recurrent state is formulated as a template-based constraint satisfaction problem. The constraint satisfaction problem for recurrent sets turns out to be equivalent to constraint systems for invariant generation [CSS03, SSM04], therefore, techniques for constraint-based invariant generation apply directly to the problem of checking non-termination. The precision of the analysis can be adjusted by choosing an appropriate constraint theory. We can either apply a more precise bit-level analysis that takes into account many of the machine-dependent characteristics of programs such as overflow, or a less precise integer-level (arithmetic) analysis that is geared towards more algorithmic reasons for non-termination.

6.1 Example

Non-Terminating Binary Search. Joshua Bloch recently pointed out that many implementations of binary search, for example, the version that used to be in Java's standard library, can produce `ArrayOutOfBounds` exceptions because it ignores arithmetic overflows [Blo06]. Specifically, the statement:

$$\texttt{mid} = (\texttt{lo} + \texttt{hi})/2$$

that computes the midpoint of the range can overflow for large values of `lo` and `hi`, producing a negative value that is used as an array index.

The version of binary search shown in Figure 6.1 is similar to the original, except that we have replaced the signed integers of the original version with *unsigned* integers. (This is similar to the signature of binary search in C's stdlib which uses the unsigned type `size_t` for the indices.) This time, while the array

index will remain within bounds, the arithmetic overflow can lead to an infinite loop.

We illustrate the non-termination by considering an execution of a path through the `while` loop that follows the first branch of the conditional statement:

$$[\texttt{lo} \leq \texttt{hi}]; \texttt{mid} := (\texttt{lo} + \texttt{hi})/2; [a[\texttt{mid}] < k]; \texttt{lo} := \texttt{mid} + 1.$$

We maliciously choose the following initial values:

$$\texttt{lo} = 1, \qquad \texttt{hi} = MAXINT, \qquad \texttt{a}[0] < \texttt{k},$$

where $MAXINT$ is the maximum value of an unsigned integer. The first statement of the path is the evaluation of the loop condition $\texttt{lo} \leq \texttt{hi}$, which returns true for the chosen inputs. Then, we update the value for the variable `mid`. Then $\texttt{mid} = 0$ (by arithmetic overflow), and we come back to the head of the loop with $\texttt{lo} = 1$ and $\texttt{hi} = MAXINT$, never terminating. Thus, if an adversary can choose the input values to the program, she can force the program to enter an infinite loop.

One sufficient condition to check for such non-terminating loops is to find a *lasso-shaped* execution: a feasible program execution that reaches the head of the loop (the conditional $\texttt{lo} < \texttt{hi}$ on line `3`) with some state s, then executes the body of the loop (lines `4-11`) and goes back to the same state s (modulo the live variables). In this case, we can unwind the execution of the loop arbitrarily many times, starting at s, executing the loop and returning to s.

We can search for such paths by exploring the program *symbolically*. In symbolic execution, the program is executed on symbolic instead of, or in addition to, concrete inputs, and symbolic constraints are gathered along the execution. Any

```
 1: int bsearch(int a[], int k,
                unsigned int lo,
                unsigned int hi) {
 2:     unsigned int mid;
 3:     while (lo < hi) {
 4:             mid = (lo + hi)/2;
 5:             if (a[mid] < k) {
 6:                     lo = mid + 1;
 7:             } else if (a[mid] > k) {
 8:                     hi = mid - 1;
 9:             } else {
10:                     return mid;
11:             }
12:     }
13:     return -1;
14: }
```

Figure 6.1: [Example] Broken binary search

satisfying assignment to the symbolic constraints is guaranteed to execute the program along the current path. Let X be the set of program variables (ignoring the heap for the moment). Then, for a program path p, we write $\rho_p(X, X')$ to denote the symbolic constraint generated by symbolic execution when traversing p. It constrains the value X' of the program variables at the end of the path in terms of their original values X.

We assume that symbolic execution completes a loop in the control flow graph of the program, say along the path $\ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$ where ℓ_0 is the entry point of the program, ℓ is the head of a loop, and *stem* and *loop* are respectively the executed path up to the loop and along the loop body. To detect an infinite loop, we can ask for a satisfying assignment to the query:

$$\rho_{stem}(X_0, X) \wedge \rho_{loop}(X, X') \wedge X = X' \tag{6.1}$$

While we write $X = X'$ to enforce the search for the same state, we only require equality of all *live* variables (i.e., variables whose values will be read in the future). For the program `bsearch`, and the path that takes the first branch of the conditional on line `5`, the generated query is:

$\mathsf{true} \wedge$	constraints up to line 3
$\texttt{lo} < \texttt{hi} \wedge \texttt{mid} = (\texttt{lo} + \texttt{hi})/2 \wedge$	
$a[\texttt{mid}] < k \wedge \texttt{lo}' = \texttt{mid} + 1$	constraints for the loop
$\wedge \texttt{lo}' = \texttt{lo} \wedge \texttt{hi}' = \texttt{hi}$	set live variables equal

This query can be resolved by a decision procedure that treats variables and arithmetic in a bit-accurate way [CKS05, GD07b, XA05], and a satisfying assignment to the variables will provide an input that causes non-termination.

Unbounded Ranges and Recurrent Sets. Unfortunately, modeling the bit-accurate semantics leads to too many cases of non-termination, as a significant portion of C code is written without considering overflow, or with implicit assumptions about the size of integer inputs to the program. For example, under the bit-accurate semantics, the following loop does not terminate:

```
n := input();
for (unsigned i = 0; i <= n; i++) body;
```

An offending input is when n is $MAXINT$, the maximum representable number in the machine. When i is incremented after it reaches n, the value rolls back to 0, and the loop check $i \leq n$ continues to hold. Such loops occur in library functions that sort inputs. However, there is always an implicit assumption that the arrays we shall sort in practice are not that large (or the memory allocator will fail to allocate the array even before the sorting routine is called).

While these non-terminating programs represent an important class of bugs, especially for security and denial-of-service related vulnerabilities where an attacker can exploit the overflow, we also want to find non-terminating executions in the still important *abstracted* semantics, where numbers are modeled as unbounded, mathematical integers. These non-terminating executions can point out *algorithmic* problems in code.

In the abstracted semantics, the condition in Equation (6.1) is sufficient for non-termination, but clearly not necessary. The *same* state may not be reached in any iteration of the loop, but the execution can nevertheless be non-terminating. For example, consider the loop:

```
function loop() {
  i = input(); y = input();
  if (y!=0)
    while (i >= 0) { i = i - y; }
}
```

The loop is non-terminating if the initial value of i is non-negative, and y is negative. In this case, every execution of the loop body produces a new state where the new value of i is the old value of i minus the value of y. Thus, for no unrolling of the loop body can we satisfy the requirement from Equation (6.1). Instead, to show this loop is non-terminating, we check for a *recurrent set*.

A recurrent set R is a set of states at the head of the loop that satisfies the following properties:

(1) R satisfies the loop predicate p,

(2) some reachable state s satisfies R, and

(3) for any state s satisfying R, the successor of s after executing the loop body is again in R.

Clearly, if there is such an R, then we can find a non-terminating execution of the program. In case of the loop in `loop`, the set $i \geq 0 \wedge y \leq 0$ forms a recurrent set. The first two conditions are easily checked, and for condition (3), we notice that $i \geq 0 \wedge y \leq 0 \wedge i' = i - y \wedge y' = y \Rightarrow i' \geq 0 \wedge y' \leq 0$. Thus, we can conclude that there is a non-terminating execution of the loop. We have reduced the search for non-terminating executions to the search for recurrent sets.

Reduction to Constraint Solving. We shall use a template-based approach to searching for recurrent sets, reducing the search for recurrent sets to constraint solving over a suitable domain. We restrict attention to the important class of *linear* programs, where the transition relation is a linear function of the variables (and their updated values). Consider the lasso-shaped execution:

$$\mathtt{i} := \mathtt{input}(); \mathtt{y} := \mathtt{input}(); [\mathtt{y} \neq 0]; [\mathtt{i} \geq 0]; \mathtt{i} := \mathtt{i} - \mathtt{y};$$

Let us assume that the recurrent set is defined by a parametric linear inequality together with the loop guard. That is, we assume the template

$$\mathtt{i} \geq 0 \wedge a\mathtt{i} + b\mathtt{y} \geq c,$$

where a, b, and c are unknown parameters. They define a recurrent set if an execution through the loop starting from a state satisfying this constraint returns

to this set:

$$\mathtt{i} \geq 0 \wedge a\mathtt{i} + b\mathtt{y} \geq c \Rightarrow$$
$$\mathtt{i} \geq 0 \wedge \mathtt{i}' = \mathtt{i} - \mathtt{y} \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{i}' \geq 0 \wedge a\mathtt{i}' + b\mathtt{y}' \geq c.$$

These constraints are similar to constraints generated in template-based invariant generation [CSS03], and similar non-linear constraint solving techniques can be used to solve them, we provide the formal details in Section 6.4. After solving these constraints for a, b, and c, we get a recurrent set

$$\mathtt{i} \geq 0 \wedge \mathtt{y} \leq 0.$$

This set is reachable for the initial condition $\mathtt{i} \geq 0 \wedge \mathtt{y} < 0$. Any solution to these constraints demonstrates an input that causes non-termination.

Summary. Our non-termination checker has two components: one component performs a reachability computation on the state space of the program to enumerate all possible lasso shaped executions, and the second component attempts to infer a recurrent set for each lasso-shaped execution. In our implementation, the enumeration of lassos is performed using directed test generation. Existence of a recurrent set implies that the current lasso induces a non-terminating execution. If no recurrent set is found (either because the loop terminates or because the template is too weak), the generation of lassos continues.

6.2 Definitions

We develop our algorithm for an abstract imperative programming language. For ease of exposition, this language ignore features such as memory and references, or function calls.

A *program* $P = (X, \mathcal{L}, \mathcal{L}^{\circlearrowleft}, \ell_0, \mathcal{T})$ consists of a set X of variables, a set $\mathcal{L}$ of control locations, a set of cutpoint locations $\mathcal{L}^{\circlearrowleft} \subseteq \mathcal{L}$, an initial location $\ell_0 \in \mathcal{L}$, and a set $\mathcal{T}$ of transitions. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in \mathcal{L}$ are control locations, and ρ is a *transition relation* over free variables from $X \cup X'$. The variables from X denote values at control location ℓ, and the variables from X' denote the (updated) values of the variables from X at control location ℓ'. The sets of locations and transitions naturally define a directed graph, called the *control-flow graph* (CFG) of the program [ASU86]. Note though that unlike [ASU86] we put the transition constraints at the edges of the graph. We assume that for any infinite path through the control-flow graph, there exists at least one control location in $\mathcal{L}^{\circlearrowleft}$ that is visited by the path infinitely many times. In our examples, we shall write programs using a C-like syntax, but these can be easily processed into (abstract) programs.

A *state* of the program P is a valuation of the variables from X. The set of all states is denoted $\mathsf{V}.X$. We shall represent sets of states using formulas over X. We write $s \models \psi$ if the state $s \in \mathsf{V}.X$ satisfies the formula ψ. A formula ψ over X represents the set $\{s \in \mathsf{V}.X \mid s \models \psi\}$. For a formula ρ over $X \cup X'$ and a valuation $(s, s') \in \mathsf{V}.X \times \mathsf{V}.X'$, we write $(s, s') \models \rho$ if the valuation satisfies the constraint ρ. An *execution* of the program P is a sequence $\langle \ell_0, s_0 \rangle, \langle \ell_1, s_1 \rangle, \ldots, \langle \ell_k, s_k \rangle \in (\mathcal{L} \times \mathsf{V}.X)^*$, where ℓ_0 is the initial location and for each $i \in \{0, \ldots, k-1\}$, there is a transition $(\ell_i, \rho, \ell_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A *path* of the program P is a sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_k)$ of transitions, where ℓ_0

is the initial location and the sequence of transitions form a path in the CFG. The path π is *feasible* if there is an execution $\langle \ell_0, s_0 \rangle, \ldots, \langle \ell_k, s_k \rangle$ such that for each $i \in \{0, \ldots, k-1\}$, we have $(s_i, s_{i+1}) \models \rho_i$. A location ℓ is *reachable* if some feasible path ends in ℓ. A state s is reachable at location ℓ if $\langle \ell, s \rangle$ appears in some computation. Feasibility is extended in a natural way to paths that are *infinite*. That is, an infinite path $\pi = (\ell_0, \rho_0, \ell_1), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_k), \ldots$ is feasible if there is an infinite execution sequence $\langle \ell_0, s_0 \rangle, \ldots, \langle \ell_k, s_k \rangle \ldots$ for all $i \in \mathbb{N}$, we have $(s_i, s_{i+1}) \models \rho_i$. For a finite path $(\ell_0, \rho_0, \ell_1) \ldots (\ell_{k-1}, \rho_{k-1}, \ell_k)$, we shall write a "compound transition" (ℓ_0, ρ, ℓ_k) where $\rho = \rho_0 \circ \ldots \circ \rho_{k-1}$, and $\circ$ is the relational composition operator defined by

$$(\phi \circ \psi)(X, X') = \exists X''.\ \phi(X, X'') \wedge \psi(X'', X').$$

A *lasso* at a cutpoint location $\ell \in \mathcal{L}^{\circlearrowleft}$ consists of two sequences of transitions, which are referred to as *stem* and *loop*:

$$\mathit{lasso} \ = \quad \ell_0 \xrightarrow{\mathit{stem}} \ell \xrightarrow{\mathit{loop}} \ell$$

The *stem* is a path from the initial location ℓ_0 to the location ℓ. The *loop* consists of a path that starts and ends at the cutpoint location ℓ by following a cyclic path through the control-flow graph. A lasso *induces* an infinite execution if the infinite path $\mathit{stem}(\mathit{loop})^{\omega}$ obtained by traversing the stem and then unwinding the loop infinitely many times is feasible.

6.3 Generating Lassos

Our algorithm for detecting non-terminating executions has two parts: one part generates lassos in the control-flow graph, and the second part checks if each generated lasso induces an infinite execution.

We now describe the algorithm NonTerm (shown in Figure 6.2) that searches for lassos in the control-flow graph of the program.

Non-Deterministic Search. In Algorithm NonTerm, we use the variables ℓ, s, and τ to store the current location, program state, and transition that leads to the current location. The search for lassos is divided into two phases, which follows the lasso structure. The variable ℓ' is used to fix the cutpoint for the loop. During the first phase, we construct the stem part, see lines 6–9 in Figure 6.2. It is chosen nondeterministically and also fixes the cutpoint location ℓ'. The second phase, see lines 11–14, nondeterministically chooses a loop at ℓ'.

Our high-level exposition combines nondeterministic choice and backtracking to achieve exhaustive enumeration of all possible lassos. We use a nondeterministic choice operator Choose, which selects an arbitrary element from a given set. We assume that Choose raises the ChoiceFailure exception when applied on the empty set. The function ChooseNext determines how a stem/loop is extended. When applied on a program state s at location ℓ, it returns a program transition that starts from ℓ, a successor location and state under this transition. The definition of ChooseNext is shown in Figure 6.3. The Choose and ChooseNext operations can be effectively implemented using symbolic execution and depth first search.

We assume that if no more backtracking is possible, i.e., if all possible choices have been explored by the algorithm, then the exception BacktrackingEx-

HAUSTED is raised. In this case, we report that the program is terminating, see line 22.

Check. A discovered lasso is analyzed w.r.t. the non-termination. We use symbolic constraint-based methods for this analysis, which is motivated by the following considerations. Symbolic methods can effectively explore all possible executions that follow the stem and then unwind the loop part, despite their large or unbounded number. Even if a particular execution analyzed by the algorithm is terminating, there may be a similar one—an execution that traverses the same stem and loop, but has a different valuation of the program variables—that is non-terminating.

We apply a non-termination checker by calling the function NONTERMLASSO, see line 15. If the proof of non-termination succeeds, then we assume that it yields an initial state of a non-terminating execution. These state is reported as evidence of non-termination, and the algorithm NONTERM succeeds. In general, the transition relation of the lasso may be nondeterministic, e.g., contain some input statements. In this case, we also require that NONTERMLASSO computes a sequence of valuations for the inputs that are read by the program during the lasso traversal. In Section 6.4, we describe algorithms for proving non-termination that provide a range of precision/efficiency trade-offs.

A failed non-termination check guides the algorithm into the search for a different lasso. We rely on backtracking to explore the alternative choices of the calls to CHOOSE and CHOOSENEXT.

Correctness. The correctness of the algorithm NONTERM relies on two components: the exhaustiveness of the search process for lassos, and the soundness of NONTERMLASSO, i.e., it only gives the positive result if there exists an infinite

execution induced by the lasso.

Theorem 2 (Correctness)**.** *If algorithm* NonTerm *on input program P terminates and returns "non-terminating execution starts from s" then P has an infinite execution starting from state s. If algorithm* NonTerm *on input program P terminates and returns "program terminates" then the execution of P terminates starting from any initial state s.*

Sketch. The first case immediately follows from the correctness of NonTermLasso. For the second case, we observe that if the BacktrackingExhausted exception is raised then NonTerm enumerated all possible lassos. From the correctness of NonTermLasso, we conclude program termination. □

6.4 Proving Feasibility of Lassos

In this section, we propose algorithms for proving *non-termination* of lassos, which we use to implement the function NonTermLasso in our algorithm NonTerm for testing non-termination.

(Non-)Well-Foundedness. First, we describe conditions when a relation can induce infinite sequences. Subsequently, we extend it to deal with lassos.

A binary relation $\rho(X, X')$ over states is *not well-founded* at a state s if there exists an infinite sequence $s_1, s_2, \ldots$ such that $s_1 = s$ and for each $i \geq 1$ we have $(s_i, s_{i+1}) \models \rho$. The relation is *well-founded* at s if there is no such infinite sequence. We are interested in finding the initial states of infinite sequences induced by relations that are not well-founded.

Let $lasso = \ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$ be a lasso where ρ_{stem} and ρ_{loop} are transition relations of the stem and loop, respectively. The lasso induces an infinite exe-

cution if the transition relation of the loop part is not well-founded at a state that is reachable by traversing the stem. Formally, the lasso induces an infinite execution if the relation

$$\exists X.\ \rho_{stem}(X, X') \wedge \rho_{loop}(X', X'')$$

is not well-founded.

6.4.1 Recurrent Sets

We now provide a condition for checking that a relation is not well-founded. We formulate our condition in terms of *recurrent sets*. Let ρ be a relation. A state s' is called a ρ-successor of a state s if $(s, s') \models \rho$. A set of states $\mathcal{G}(X)$ is *recurrent* for ρ if for each state $s \models \mathcal{G}(X)$, there exists a ρ-successor state s' such that $s' \models \mathcal{G}(X)$.

Proposition 1 (Recurrent sets and non-well-foundedness)**.** *A relation $\rho(X, X')$ is not well-founded if and only if there exists a non-empty recurrent set of states, i.e., if for some $\mathcal{G}(X)$, we have:*

$$\exists X.\ \mathcal{G}(X), \tag{6.2}$$

$$\forall X\ \exists X'.\ \mathcal{G}(X) \rightarrow \rho(X, X') \wedge \mathcal{G}(X'). \tag{6.3}$$

Proof. If a non-empty recurrent set exists, then we generate an infinite sequence by picking an element satisfying $\mathcal{G}(X)$ (this is possible by Condition (6.2)), and then constructing an infinite sequence by iteratively applying Condition (6.3). If a relation is not well-founded, let $s_1, s_2, \ldots$ be an infinite sequence induced by the relation. We define $\mathcal{G}(X)$ to be the set $\{s_1, s_2, \ldots\}$. □

We illustrate recurrent sets by example. Consider the relation ρ over the variables x, y and x', y' such that

$$x \geq 0 \wedge x' = x + y \wedge y' = y + 1.$$

We observe that for the construction of an infinite sequence induced by ρ it is necessary that the value of variable x is always positive. One possibility to ensure this condition is to start with a positive value of x and increase it at each step. Hence, we obtain a recurrent set

$$\mathcal{G}_1(x, y) \;=\; x \geq 0 \wedge y \geq 0.$$

An alternative recurrent set admits infinite sequences in which the value of x may decrease initially, but never decreases below zero:

$$\mathcal{G}_2(x, y) \;=\; x \geq 0 \wedge x \geq \frac{1}{2}|y|(|y| + 1).$$

An example infinite sequence for the recurrent set $\mathcal{G}_2(x, y)$ is

$$\langle 6, -3\rangle,\ \langle 3, -2\rangle,\ \langle 1, -1\rangle,\ \langle 0, 0\rangle,\ \langle 0, 1\rangle,\ \langle 1, 2\rangle,\ \langle 3, 3\rangle,\ \ldots.$$

When analyzing the non-termination of a lasso, we need to construct a recurrent set for the loop of the lasso that is moreover reachable by traversing the stem.

Proposition 2 (Recurrent set for lasso)**.** *A lasso* $\ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$ *induces an infinite execution if there exists a recurrent set* $\mathcal{G}(X')$ *for the relation* $\rho_{loop}(X', X'')$ *such that*

$$\exists X\ \exists X'.\ \rho_{stem}(X, X') \wedge \mathcal{G}(X'). \tag{6.4}$$

6.4.2 From Recurrent Sets to Constraint Systems

We now describe two symbolic analyses to construct recurrent sets satisfying the conditions of Proposition 2 for a given lasso. The first, *bitwise*, analysis assumes that the state space is finite, and without loss of generality, encoded using Boolean variables. The second, *linear arithmetic* analysis assumes that every program transition along the lasso can be represented as a (rational) linear constraint over the program variables.

The bitwise analysis enables the precise treatment of low-level features of programming languages, e.g., bit-wise operations and arithmetic modulo fixed widths. The linear arithmetic analysis is an useful abstraction for programs when bit-level precision is not required. In either case, we show how we can reduce the search for recurrent sets to automatic constraint solving.

Bitwise Analysis. For the bitwise analysis, we assume that program variables range over Booleans, and that the transition relation of the lasso is given by a Boolean formula over propositional variables. Since the state space is finite, a lasso induces an infinite execution iff some state is repeated infinitely many times in the course of the execution. Therefore, we can restrict the search to *singleton* recurrent sets (i.e., recurrent sets which have exactly one state). Given a lasso $\ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$, we therefore look for a state s that reaches ℓ by executing the transition *stem*, and after executing the transition *loop*, comes back to itself. We encode this condition in the constraint

$$\exists X\, \exists X'.\ \rho_{stem}(X, X') \wedge \rho_{loop}(X', X'), \tag{6.5}$$

The function NonTermLasso returns a positive result if this constraint is satisfiable, and can be implemented using Boolean satisfiability solving. The valuation of X is an initial state that witnesses non-termination. This is the same as bounded model checking for liveness [CBR01]. The constraints can be resolved by a bit-precise decision procedure such as Cogent [CKS05] or STP [GD07b], that eventually reduces the checks to Boolean satisfiability.

Notice that the constraint in Equation (6.5) may not be satisfied for a syntactic loop in the program, but only for some unrolling of this loop. Consider the program

```
while (x == y) { x = not x; y = not y; }
```

which has an infinite loop if `x` and `y` are initially equal, but for which the constraint

$$(\mathtt{x} = \mathtt{y}) \wedge (\mathtt{x} = \neg\mathtt{x} \wedge \mathtt{y} = \neg\mathtt{y})$$

obtained from Equation (6.5) is unsatisfiable. However, since we exhaustively generate all lassos, we shall eventually consider a lasso where the loop is unrolled once:

```
[x==y]; x' = not x; y' = not y;
        [x'==y']; x'' = not x'; y'' = not y';
```

for which a singleton recurrent set exists.

Linear Arithmetic Analysis. The linear arithmetic analysis assumes that the program transitions are representable using conjunctions of linear inequalities over the program variables.

Our algorithm follows a constraint-based approach for the synthesis of auxiliary assertions for temporal verification, e.g., linear and non-linear invariants and ranking functions [CSS03, SSM04, Cou05, Kap06]. We extend its applicability to synthesizing recurrent sets.

The constraint-based approach to the generation of auxiliary assertions reduces the computation of an assertion to a constraint solving problem. The reduction is performed in three steps. First, a *template* that represents an assertion to be computed is fixed in an *a priori* chosen language. The parameters in the template are the unknown coefficients that determine the assertion. Second, a set of *constraints* over these parameters is defined. The constraints encode the validity of the assertion. This means that every solution to the constraint system yields a valid assertion. Third, an *assertion* is obtained by solving the resulting constraint system.

Our approach to generate recurrent sets follows the same three steps. We use *templates* over linear inequalities to represents recurrent sets. We derive constraints over template parameters that encode the conditions in Equations (6.2)–(6.4) from Section 6.4.1. Then, we solve the constraints and obtain a recurrent set.

Recurrent sets are defined by universally quantified conditions. As for invariant generation in linear arithmetic, our main technical tool for the elimination of universal quantification will be Farkas' lemma from linear programming.

Theorem 3 (Farkas' Lemma [Sch86])**.** *A satisfiable system of linear inequalities* $Ax \leq b$ *implies an inequality* $cx \leq \delta$ *if and only if there exists a non-negative vector* λ *such that* $\lambda A = c$ *and* $\lambda b \leq \delta$*.*

We now give the details of our algorithm for the computation of recurrent sets. We assume that the transition relations of the stem and loop parts are given by

systems of inequalities. In particular, we assume that the transition relation of the loop is given by a guarded command with the guard $Gx \leq q$ and updates $x' = Ux + u$.[1] Our algorithm computes a recurrent set $\mathcal{G}$ that is an instantiation of a template consisting of a conjunction of linear inequalities:

$$\mathcal{G} = Tx \leq t.$$

First, we present a translation of Condition (6.3) into constraints over template parameters. We eliminate the existential quantification in Condition (6.3) by substituting the definition of the primed variables given by the loop update:

$$\forall x.\ \mathcal{G}(x) \rightarrow \rho_{loop}(x, Ux + u) \wedge \mathcal{G}(Ux + u),$$

which we write in matrix form:

$$\forall x.\ Tx \leq t \rightarrow Gx \leq g \wedge TUx \leq t - Tu.$$

Here, the template parameters T and t are existentially quantified. Next, we eliminate the universal quantification by encoding the validity of implication using Farkas' lemma:

$$\exists \Lambda \geq 0.\ \Lambda T = \begin{pmatrix} G \\ TU \end{pmatrix} \wedge \Lambda t \leq \begin{pmatrix} g \\ t - Tu \end{pmatrix}. \tag{6.6}$$

[1] We use low case x instead of X to denote program variables in this section to avoid confusion between matrices and vectors.

We can translate Condition ((6.2)) into the constraint:

$$\forall \mu \geq 0.\ \mu T = 0 \rightarrow \mu t \geq 0.$$

The translation for (6.4) is similar. Together with (6.6), this leads to the final constraint defining recurrent sets that contains quantifier alternation, from existential to universal, as well as non-linear constraints arising from the multiplication between template parameters and variables that encode the implication validity. The constraints are similar to those for invariant generation [CSS03], and we can use existing techniques based on instantiations and case splitting. Unfortunately, there is no practical constraint solver that supports quantifier alternation. We propose an alternative solution, which we can be implemented using a constraint solver that can iteratively enumerate all solutions. We enforce Conditions (6.2) and (6.4) by evaluating them for the values of T and t that the constraint solver computes for constraint (6.6). If the conditions are not satisfiable, then we require the solver to find alternative values for T and t. A constraint logic programming-based solver, e.g., clp(Q,R) [Hol95], can implement this backtracking search. We summarize the described algorithm NonTermLasso in Figure 6.4.

Theorem 4. *If Algorithm* NonTermLasso *on an input lasso with stem* $S(xx') \leq s$*, loop* $Gx \leq g \wedge x' = Ux + u$*, and template* $Tx \leq t$ *terminates and returns "recurrent set"* $T^*x \leq t^*$ *then* $T^*x \leq t^*$ *is a recurrent set for the lasso.*

We illustrate the constraint generation process on the loop:

$$x \geq 0 \wedge x' = x + y \wedge y' = y + 1,$$

which we write in matrix form as

$$\underbrace{\begin{pmatrix}-1 & 0\end{pmatrix}}_{G}\begin{pmatrix}x\\y\end{pmatrix} \leq \underbrace{0}_{g} \wedge \begin{pmatrix}x'\\y'\end{pmatrix} = \underbrace{\begin{pmatrix}1 & 1\\0 & 1\end{pmatrix}}_{U}\begin{pmatrix}x\\y\end{pmatrix} + \underbrace{\begin{pmatrix}0\\1\end{pmatrix}}_{u}.$$

First, we assume a template:

$$t_x x + t_y y \leq t \quad \wedge \quad s_x x + s_y y \leq s,$$

where t_x, t_y, t, s_x, s_y, and s are unknown parameters. Then, we have:

$$TU = \begin{pmatrix}t_x & t_x + t_y\\ s_x & s_x + s_y\end{pmatrix} \quad \text{and} \quad t - Tu = \begin{pmatrix}t - t_y\\ s - s_y\end{pmatrix}.$$

Following (6.6), for:

$$\Lambda = \begin{pmatrix}\lambda_{11} & \lambda_{12}\\ \lambda_{21} & \lambda_{22}\\ \lambda_{31} & \lambda_{32}\end{pmatrix}$$

we obtain the constraint system:

$$\exists \Lambda \geq 0.\ \Lambda \begin{pmatrix}t_x & t_y\\ s_x & s_y\end{pmatrix} = \begin{pmatrix}-1 & 0\\ t_x & t_x + t_y\\ s_x & s_x + s_y\end{pmatrix} \wedge \Lambda \begin{pmatrix}t\\ s\end{pmatrix} \leq \begin{pmatrix}0\\ t - t_y\\ s - s_y\end{pmatrix}.$$

We compute a solution:

t_x	t_y	t	s_x	s_y	s
-1	0	0	0	-1	0

,

which defines the recurrent set $x \geq 0 \wedge y \geq 0$. It is straightforward to check that the validity of the corresponding implication in Condition (6.3) is established by Λ such that:

$$\Lambda = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix},$$

for which the constraint below evaluates to true:

$$\Lambda \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ -1 & -1 \\ 0 & -1 \end{pmatrix} \wedge \Lambda \begin{pmatrix} 0 \\ 0 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Our algorithm requires that a template for recurrent sets is provided. We propose an iterative strengthening heuristic to find a template for which a recurrent set exists. We start with a template that is a singleton conjunction, and incrementally add more conjuncts if the constraint solving fails.

6.4.3 Weaker Conditions for Recurrent Sets

A lasso induces an infinite execution iff some unrolling of its loop part is not well-founded at a state reachable by executing the stem. These unrollings determine weaker conditions on recurrent sets, which can sometimes lead to more succinct representations for recurrent sets.

We first illustrate the weakening by example, and then provide a formal ac-

count. Consider a lasso

$$\begin{aligned}\rho_{stem} = &\quad y' = 0,\\ \rho_{loop} = &\quad x \geq 0 \wedge x' = x + y \wedge y' = 1 - y.\end{aligned}$$

The lasso induces infinite executions, and a recurrent set is:

$$\mathcal{G}_1(x, y) = \quad x \geq 0 \wedge (y = 0 \vee y = 1).$$

We observe that the recurrent set has disjunctions, which are difficult for constraint solvers to reason about. However, we may also consider a loop relation ρ_{loop^2} obtained from the given one by unwinding it one time:

$$\begin{aligned}\rho_{loop^2}(X, X') &= \rho_{loop} \circ \rho_{loop}(X, X')\\ &= x \geq 0 \wedge x + y \geq 0 \wedge x' = x \wedge y' = y.\end{aligned}$$

For this relation we compute a recurrent set

$$\mathcal{G}_2(x, y) = \quad x \geq 0 \wedge y = 0.$$

This set is represented using a conjunction of atomic predicates. Any infinite sequence induced by the lasso $stem.loop^2$ is also induced by the original lasso.

We now define the i-th weakening for recurrent sets. Given a binary relation ρ, we say that $\mathcal{G}$ is i-th recurrent for ρ, for $i \geq 1$, if it is recurrent for the relation ρ^i, which is obtained from ρ by unwinding it i-many times, i.e.,

$$\forall X\ \exists X'.\ \mathcal{G}(X) \rightarrow \rho^i(X, X') \wedge \mathcal{G}(X'), \tag{6.7}$$

where

$$\rho^i(X, X') = \begin{cases} \rho(X, X') & \text{if } i = 1, \\ \rho(X, X') \circ \rho^{i-1}(X, X') & \text{if } i > 1. \end{cases}$$

The constraint-based non-termination check (Algorithm NonTermLasso) can be implemented using i-th recurrent sets. We observe that we can account for i-th weakening, where $i > 1$, by considering the (unrolled) loop relation:

$$\rho^i_{loop}(x, x') = \; GU^{i-1}x \leq g - \sum_{j=1}^{i-1} GU^{j-1}u \wedge$$
$$x' = U^i x + \sum_{j=1}^{i} U^{j-1}u.$$

As a heuristic, when proving non-termination, we first try to compute a recurrent set following the original definition, i.e., no weakening is applied. If the computation fails, then we continue with a weaker definition of recurrent sets until either an i-th recurrent set is computed or an upper bound on the number of weakening attempts is reached. The template strengthening heuristic from the previous section can be combined with weakening of recurrent sets. We increase the number of conjuncts in the template only after a sequence of weakening steps, up to an a priori given bound, is explored. Such a combination attempts to avoid the transition to more expensive constraint solving tasks by first trying to simplify recurrent sets through their weakening.

6.5 Experiences

6.5.1 Implementation

We have implemented TNT , a tool that implements the termination checking algorithm for C programs. TNT has an outer loop that performs concolic execution by running the program on concrete as well as symbolic inputs. The constraints generated during concolic execution are bit accurate, and solved using the decision procedure STP [GD07b]. In our symbolic execution, the heap is always kept concrete, we only allow symbolic constants with base types. The template-based search for recurrent sets is implemented using a Sicstus Prolog based constraint solver for invariant generation.

In preliminary experiments, we checked for non-termination of simple and small programs, including the programs from Section 6.1, and an abstraction of the non-termination bug from [CPR06]. In each case, the non-terminating loop was identified in a few seconds.

One limitation of the current implementation is that the heap is concrete. While this means that the recurrent sets benefit from precise address information, it is also a limitation of the tool to find potentially infinite executions that depend on shape assumptions on data structures. For example, we cannot catch infinite executions arising from acyclic list traversal routines when they are applied on circular lists as input. Integrating our work with symbolic shape information is left as future work.

6.5.2 Mondriaan Memory Protection

In addition to the simple examples, we ran our tool on an early implementation of the Mondriaan memory protection system [WCA02, Wit04].

Mondriaan is a protection scheme that allows flexible memory sharing and fine-grained permission control between different user applications and a trusted supervisor mode running on an OS. Unlike usual virtual memory that operates at the page level, Mondriaan allows arbitrary permissions control at the granularity of individual memory words. In the Mondriaan implementation, memory is organized as a linear address space, divided into *user segments*. Each user segment determines a range of addresses and permissions associated with addresses in that range, and is defined as a triple $\langle b, l, p \rangle$ of a base address b, a length l, and a permission p. By setting the permissions of a *user segment*, a process can flexibly and safely share its address space with other processes. A privileged supervisor mode provides an API to modify permissions. The permissions associated with a user segment can be modified using this API by specifying the base word address, the length of the user segment in words, and the desired permission (none, read-only, read-write, or execute-read). Each *protection domain* (i.e., group of processes sharing the address space) maintains a permissions table, stored in privileged memory, that specifies the permission associated with each address of the address space.

The permissions table is organized as a compressed multi-level table, with three levels: a root table, a mid-level table, and a leaf table. Figure 6.5 shows how the table is indexed by a 32-bit address. The root table has 1024 entries (indexed by the top 10 bits in Figure 6.5), each of which maps a 4MB block of memory. Each mid-level table has 1024 entries (indexed by the next 10 bits), each of which maps a 4KB block. The leaf tables have 64 entries each (indexed by 6 bits), and each entry provides individual permissions for 16 four-byte words.

To save space, upper level table entries can either be pointers to lower level tables (or NULL), or directly hold a permissions vector for sub-blocks. In this

case, the rightmost bit of the entry holds whether the entry is a pointer or a permissions vector (since addresses are word-aligned, the last two bits of an address are always zero). Permission vectors stored in upper levels of the table only store permissions for 8 sub-blocks (instead of 16 in the leaves). An auxiliary function `uentry_is_data` is used to check whether an entry is a pointer to a lower level table or a permissions vector.

We chose Mondriaan as our case study because (1) correctness is crucial, since the code runs in privileged mode in the kernel, (2) the code was complicated enough to warrant its author to annotate the code with properties (both as assertions and as comments), and to suggest it as a challenge for formal verification, and (3) the code is low-level, performing extensive bitwise operators that extract indices to arrays, and memory-intensive, traversing multiple levels of data structures, and both these features provide exceptional challenges to static analysis.

Initially, the Mondriaan implementation creates the root permission table with 1024 entries all filled with 0s (that is, no permissions associated with any address). Figure 6.9 shows source code to perform updates from an early version of the Mondriaan memory protection system [WCA02, Wit04] that was provided to us by the author Emmett Witchel as a challenging verification problem. The code updates the permissions of a user segment in the permission table by taking as input a user segment and updating the permissions of the segment in the table.

The actual update is performed by the recursive procedure `_mmpt_insert`, shown in Figure 6.9, which takes as input a pointer to the permission table structure `mmpt`, a user segment (a base address, a pointer `len` to the length of the segment, and the desired protection `prot`), and additional control parameters such as a pointer to the current table (initially, the root table), the current level

(root, mid-level, or leaf), and flags determining whether or not some table can be freed and whether or not to allocate. An insertion into the table for a user segment is performed as a call

$$\texttt{\_mmpt\_insert}(\texttt{mmpt}, \texttt{base}, \&\texttt{len}, \texttt{prot},$$
$$\texttt{mmpt}\rightarrow\texttt{tab}, 0, \&\texttt{nonzero}, 1);$$

The code is complicated, as it must consider several different cases in inserting new permissions to the table, and performs dynamic allocation or freeing of lower level tables when they are not necessary. Further, a user segment is broken into parts in the insertion process, and the insertion routine is called recursively for each part. Instead of explaining the code line by line, we will explain the functional behavior of `_mmpt_insert`.

As shown in Figure 6.6, `_mmpt_insert` splits the memory range from `base` to `base`+`len` in aligned 4MB blocks at the root level. For an update, this can result in one of the following two cases:

Exact Coverage. For every 4MB sub-block of the user segment that is aligned with the sub-block size (512KB), the permission vector is set in the corresponding entry in the root table. If the root entry was a pointer to a mid-level table, then the mid-level table is recursively deallocated.

Overlapped Coverage. Breaking the user segment into aligned 4MB blocks can leave "extra" blocks at the beginning and at the end, that is, the first and last blocks of the user segment may not be aligned with 512KB sub-blocks at the root level. For these blocks, `_mmpt_insert` recursively goes to the mid-level table (allocating the mid-level table if necessary) and sets permissions in the mid-level table.

Depending on the stored value in the root entry for the base address, `_mmpt_insert` first does following action:

1. If the root entry is 0 then it allocates a new mid level permission table of 1024 entries and fills the entries in the new table with zeros. The root entry is set as a pointer to this table.

2. If the root entry is a permission vector then it allocates a mid-level permission table and fills it with permission vectors. The root entry is set as a pointer to this table.

3. If the root entry is a already a pointer to a mid-level table then it follows the pointer to the mid-level table.

Then, `_mmpt_insert` computes the the memory range of the first or last section and makes a recursive call with parameters this memory range, pointer to this mid level table. With the same logic as for the root table, it divides the memory range in 4KB blocks and does the same operations for the permissions. For the possibly unaligned first and last block, it generates leaf tables. Permissions in leaf tables are set by a recursive call to `_mmpt_insert`. If leaf tables have unaligned blocks then permissions are approximated in sub-blocks of size 4 bytes in the leaf table.

Termination Bug in `_mmpt_insert`. When we ran TNT on `_mmpt_insert`, it found a non-terminating execution. This bug is caused by calling `_mmpt_insert` with a user segment whose length is not a multiple of 4. To illustrate, consider two consecutive calls to `_mmpt_insert` just after initialization (found by TNT). The first call sets read permission for a user segment of 4 bytes with base address 0. The second one sets the permission for a user segment of 3 bytes with base address

0. The first call terminates, while the second call runs into infinite recursion. Figure 6.7 shows the state of memory table after first call.

Let us trace the code of `_mmpt_insert` line by line during the second call. Procedures called by `_mmpt_insert` are summarized in Figure 6.8.

In the second call, `_mmpt_insert` is called with a (constant) pointer to the `mmpt` structure, a base address `base` of 0, the address of the length variable `len` which contains the length 3, a pointer to the root table, and `level` = 0 (signifying root entry). (We omit the other, unimportant parameters.) At line 4, the insertion routine checks that $*$`len` is not zero, that is, the current memory chunk needs to be filled in the table. At line 5, the helper procedure `make_idx` computes the index of the entry in the root table which corresponds to `base`. This extracts the top 10 bits of `base`, which turns out to be 0. At line 6, the last comparison in the conditional checks whether the value of *`len` is greater than the block size of the root entry (4MB). Since the length is 3, this check fails and control passes to line 19. The conditional on line 19 passes, since the passed table is not a leaf table, and the entry in the root table is a pointer to a mid-level table. (Recall the configuration of the permission table after the first call from Figure 6.7.)

Now at line 20, `_mmpt_insert` makes a recursive call with a pointer to the mid-level table that is pointed to by the zeroth entry of the root table, `tab`[`idx`]. The parameters `base` and `len` do not change, but `level` increases to 1, denoting that the current table is a mid-level table. In this recursive call, the control again comes to line 20, this time making another recursive call with a pointer to a leaf table, and with level equal to 2.

In this second recursive call, the conditions at lines 6, 19, and 21 all fail because the current table is a leaf table. Thus, control jumps to line 41. The `for` loop at line 41 does not execute at all, since *`len` (which is equal to 3) is smaller

then the sub-block length (4 bytes) of the leaf table. Thus, control directly jumps to line 50 and makes a recursive call with base 0, length 3, a pointer to the root table, and level reset to 0. These are the same parameters which were passed at the start of execution of the procedure `_mmpt_insert`. This causes an infinite loop, as this recursion does not terminate.

The complexity of this infinite execution (there are 3 levels of recursive calls involved) shows the utility of having a tool like Tnt check the executions for possible non-termination.

Corrected `_mmpt_insert`. On inspection of the bug and the Mondriaan test suite, we found that there is an unchecked assumption on the correct operation of the program that the lengths of user segments are always a multiple of 4. We put in this check in the update code. This time, Tnt timed out without identifying any infinite execution. Since our implementation does not implement a termination-check based acceleration, most of the time was spent in producing longer and longer symbolic traces of recursive calls.

We then proved termination of the corrected version by hand. The proof of termination involves a lexicographic ranking on the pair $(*\texttt{len}, 2 - \texttt{level})$, as on every recursive call, either the length goes down or if the length remains the same, the distance of the level from the leaf tree goes down. The reasoning for termination uses crucially the invariant that the length is always a multiple of 4 to rule out the previous infinite execution.

We believe this example is a good challenge problem for termination checkers. Unfortunately, well-documented limitations of current termination checkers (to deal with bitwise operators, or shared data structures on the heap) make it difficult to prove termination of this program automatically with the current tools.

6.6 Acceleration for NonTerm

In this section we describe a practical extension for our non-termination checking algorithm NonTerm shown in Section 6.3. We describe how to avoid redundant non-termination checks by accelerating the traversal of terminating loops.

Given a lasso, the non-termination checking algorithm NonTermLasso, as described in the previous section, can fail to return a positive result. For complete checking algorithms, the failure is caused by the termination of the lasso. Incomplete algorithms can produce an indefinite result, which leaves open the possibility that the lasso may be terminating. Since the exploration of terminating loops does not advances the search for infinite executions, we propose a modification of NonTerm that removes such loops from consideration. See Figure 6.10 for the modified statements.

If the lasso can be proven terminating no matter what the input is, we lead the execution through the loop until it exits. In order words, we fully unwind the loop by executing the loop sequence, and rely on the proof of termination to guarantee the convergence of the unwinding. Thus, we eventually reach line 18.9. The resulting sequence will be used to seed the selection of the next stem to be considered, i.e., the next stem will be chosen to contain the sequence as a prefix. Thus, NonTerm can reach interesting parts of code by passing across loops in one step, without any interruption at each iteration.

If the termination property of the lasso cannot be determined, we continue our search for non-terminating lassos. During the subsequent iterations of NonTerm, we shall only consider lassos that have the current stem as a prefix. Thus, we ensure that the search makes progress.

```
          input
            P: program
          vars
            s, τ, ℓ, ℓ' : program state, transition, and control locations
            stem, loop : sequences of transitions
          begin
1           ℓ := ℓ_0
2           Choose s ∈ V.X
3           stem := ϵ
4           loop := ϵ
5           try
6             repeat                                   (* selecting stem *)
7               τ, ⟨ℓ, s⟩ := ChooseNext(ℓ, s)
8               stem := stem • τ
9             until ℓ ∈ L^⟳ and Choose {true, false}
10            ℓ' := ℓ                                  (* fixing cutpoint location *)
11            repeat                                   (* selecting loop *)
12              τ, ⟨ℓ, s⟩ := ChooseNext(ℓ, s)
13              loop := loop • τ
14            until ℓ = ℓ' and Choose {true, false}
15            if NonTermLasso(stem, loop) then
16              s := initial state witnessing non-termination
17              return "non-terminating execution starts from s"
18            else
19              raise ChoiceFailure
20          catch ChoiceFailure do
21            backtrack
22          catch BacktrackingExhausted do
23            return "program terminates"
          end.
```

Figure 6.2: Algorithm NonTerm for testing non-termination. The operator • adds a transition at the end of a sequence. The functions Choose and ChooseNext are backtrackable.

input
  ℓ : control location
  s : program state
vars
  S : set of state-transition pairs
begin
  $S := \{(\tau, \langle \ell', s' \rangle) \mid \tau = (\ell, \rho, \ell') \in \mathcal{T} \text{ and } (s, s') \models \rho\}$
  return CHOOSE S
end.

Figure 6.3: Auxiliary function CHOOSENEXT for the nondeterministic selection of an outgoing transition, a successor location and state. The function CHOOSE raises the CHOICEFAILURE exception when applied on the empty set. We implicitly assume the fixed program P which determines the possible states s' and transitions τ.

input
  $S \binom{x}{x'} \leq s$: transition relation of the stem
  $Gx \leq g \wedge x' = Ux + u$: transition relation of the loop
  $Tx \leq t$: template for recurrent set
vars
  Φ : auxiliary constraint
  s, s' : program states – valuations of x and x'
begin
  $\Phi := \quad \exists \Lambda \geq 0.\ \Lambda T = \begin{pmatrix} G \\ TU \end{pmatrix} \wedge \Lambda t \leq \begin{pmatrix} g \\ t - Tu \end{pmatrix}$
  try
    CHOOSE $(T^*, t^*) \models \Phi$
    if exist $(s, s') \models S \binom{x}{x'} \leq s \wedge T^* x' \leq t^*$ **then**
      return "recurrent set $T^* x \leq t^*$"
    else
      backtrack
  catch CHOICEFAILURE **do**
    return "no recurrent set for template $Tx \leq t$"
end.

Figure 6.4: Auxiliary function NONTERMLASSO for checking non-termination of linear arithmetic lassos.

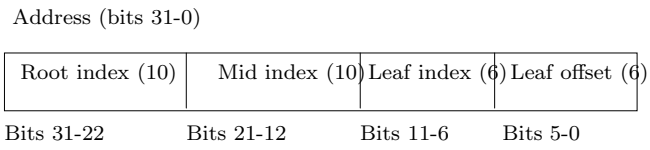

Figure 6.5: Modriaan permission table indexing

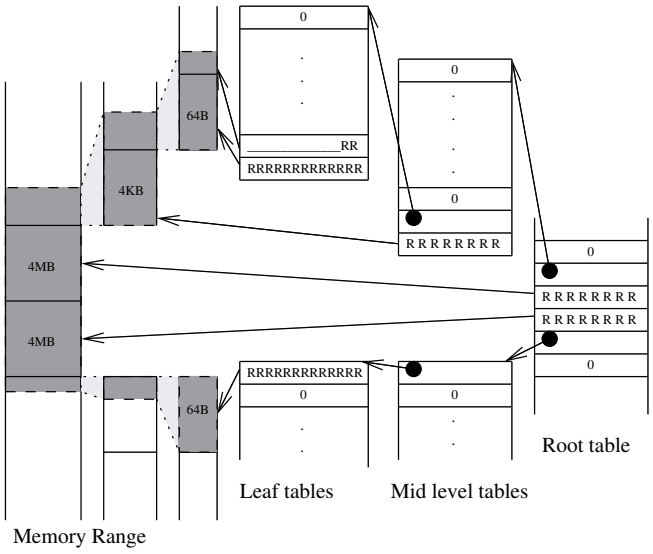

Figure 6.6: Typical permission table generated by `_mmpt_insert`

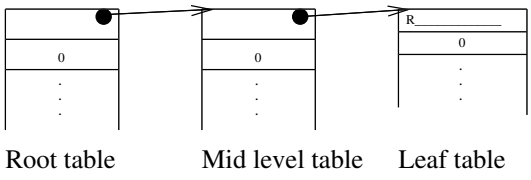

Figure 6.7: Table state after first call of `_mmpt_insert`

`tab_base`(struct mmpt* mmpt,int base,int level) Returns the lower boundary of memory range corresponds to the permission table of given level whose corresponding memory range contains `base`
`tab_len`(struct mmpt* mmpt,int level) Returns the size of memory block corresponds to each entry of permission table of given level.
`tab_addr`(struct mmpt* mmpt, int base, int idx, int level) Returns the lower boundary of memory range corresponds to the entry at index `idx` in permission table of given level whose corresponding memory range contains `base`
`tab_nentries`(struct mmpt* mmpt,int level) Returns number of entries in the table of given level.
`subblock_len`(struct mmpt* mmpt,int level) Returns sub block size of the level.
`make_idx`(struct mmpt* mmpt,int base,int level) Returns the index of first entry in permission table whose corresponding memory range contains `base`
`uentry_is_data`(struct mmpt* mmpt,int entry) Checks if the entry in the permission table is a pointer or a permission vector.
`entry_prot`(mmpt, permission_entry, i) Returns i^{th} permission field in `permission_entry`

Figure 6.8: Summary of functions called by `_mmpt_insert`

```
1.  static void
2.  _mmpt_insert(struct mmpt* mmpt, unsigned long base, unsigned long* len, int prot,
              tab_t* tab, int level, int* nonzero, int allocate_ok) {
3.    unsigned int idx; tab_t entry;
4.    if(*len == 0) return;
5.    idx = make_idx(mmpt, base, level);
6.    if(level < 2 && base == tab_base(mmpt, base, level + 1) && *len >= tab_len(mmpt, level + 1)) {
      // CASE A: Upper level, new region is aligned & spans at least one entry
7.       unsigned int entry_len;
8.       if(tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
9.          look_for_nonzero(mmpt, (tab_t*)tab[idx], level, nonzero);
10.         table_free(mmpt, (void*)tab[idx], level + 1);
11.        tab[idx] = 0;
12.      }
13.      entry = tab[idx];
14.      entry_len = make_entry(mmpt, base, *len, prot, level, &entry);
15.      tab[idx] = entry;
16.      *len -= entry_len;
17.      base += entry_len;
18.      _mmpt_insert(mmpt, base, len, prot, mmpt->tab, 0, nonzero, allocate_ok);
19.   } else if(level < 2 && tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
      // CASE B: Upper level, pointer entry
      // Recurse down through pointer
20.      _mmpt_insert(mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, nonzero, allocate_ok);
21.   } else if(level < 2 && ((base & (subblock_len(mmpt, level)-1)) != 0 || *len < subblock_len(mmpt, level))) {
      // CASE C: Upper level, NULL or data entry, new region doesn't fit in
      // subblock (not aligned or not big enough)
22.      unsigned long upper_data_entry = tab[idx];
23.      unsigned int i;
24.      *nonzero |= (tab[idx] != 0);
25.      if(allocate_ok) {
26.         unsigned long sub_len;
27.         tab[idx] = (tab_t)xmalloc(tab_nentries(mmpt, level + 1) * sizeof(*mmpt->tab));
28.         memset((tab_t*)tab[idx], 0, tab_nentries(mmpt, level + 1) * sizeof(*mmpt->tab));
29.         for(i = 0; i < 1<<mmpt->lg_num_subblock[level]; ++i) {
30.            sub_len = subblock_len(mmpt, level);
31.            _mmpt_insert(mmpt, tab_base(mmpt, base, level+1) + i * subblock_len(mmpt, level), &sub_len,
                     entry_prot(mmpt, upper_data_entry, i), (tab_t*)tab[idx], level + 1, nonzero, allocate_ok);
32.         }
33.         _mmpt_insert(mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, nonzero, allocate_ok);
34.      } else {
35.         unsigned int tlen = tab_len(mmpt, level + 1);
         // CASE D: Upper level, NULL or data entry, new region doesn't fit in
         // subblock (not aligned or not big enough), and not
         // allocating new tables
36.         if(*len < tlen) return;
37.         *len -= tlen;
38.         _mmpt_insert(mmpt, tab_addr(mmpt, base, idx+1, level), len, prot,
                     mmpt->tab, 0, nonzero, allocate_ok);
39.      }
40.   } else {
      // CASE E: Any level, NULL or data entry, fill in the rest of
      // this table and recurse for the remainder if necessary.
41.      for(; *len >= subblock_len(mmpt, level)
             && idx < tab_nentries(mmpt, level); idx++) {
42.         int entry_len;
43.         *nonzero |= (tab[idx] != 0);
44.         entry = tab[idx];
45.         entry_len = make_entry(mmpt, base, *len, prot, level, &entry);
46.         tab[idx] = entry;
47.         *len -= entry_len;
48.         base += entry_len;
49.      }
50.      _mmpt_insert(mmpt, base, len, prot, mmpt->tab, 0, nonzero, allocate_ok);
51.   }
52. }
```

Figure 6.9: Mondriaan insertion code

```
18.1    else if TermLasso(stem, loop) then
18.2       repeat                                   (* unwinding loop *)
18.3          S := {s' | s --loop--> s'}
18.4          if S ≠ ∅ then
18.5             Choose s ∈ S
18.6             stem := stem • loop
18.7          else
18.8             loop := ϵ
18.9             break
18.10      done
18.11   else
```

Figure 6.10: Acceleration of the algorithm NonTerm for testing termination. Lines 18.1—18.11 replace line 18 in Figure 6.2. We unwind the loop part of terminating lasso without intermediate checks for non-termination. Recall that the variable s holds the value of the current program state, which successors are computed during loop unwinding.

References

[AGT08] S. Anand, P. Godefroid, and N. Tillmann. "Demand-Driven Compositional Symbolic Execution." In *TACAS*, 2008.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BCE08] P. Boonstoppel, C. Cadar, and D. Engler. "RWset: Attacking Path Explosion in Constraint-Based Test Generation." In *TACAS*, 2008.

[BCH04] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. "Generating Tests from Counterexamples." In *ICSE*, 2004.

[BDE07] M. Berkelaar, J. Dirks, K. Eikland, and P. Notebaert. "lp_solve (5.5.0.10)." 2007.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. "Korat: automated testing based on Java predicates." In *ISSTA*, 2002.

[Blo06] Joshua Bloch. "Nearly all binary searches and mergesorts are broken.", June 2006. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

[BMS05] A. Bradley, Z. Manna, and H. Sipma. "The Polyranking Principle." In *ICALP*, pp. 1349–1361, 2005.

[CBR01] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. "Bounded Model Checking Using Satisfiability Solving." *Formal Methods in System Design*, **19**(1):7–34, 2001.

[CDE08] C. Cadar, D. Dunbar, and D.R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In *OSDI*, 2008.

[CGP06] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. "EXE: automatically generating inputs of death." In *CCS*, 2006.

[CKS05] B. Cook, D. Kroening, and N. Sharygina. "Cogent: Accurate Theorem Proving for Program Verification." In *CAV*, 2005.

[CL05] D. Coppit and J. Lian. "yagg: an easy-to-use generator for structured test inputs." In *ASE*, 2005.

[Cla76] L. Clarke. "A system to generate test data and symbolically execute programs." *IEEE Trans. Software Eng.*, **2**:215–222, 1976.

[CLO07] J. Clause, W. Li, and A. Orso. "Dytan: A Generic Dynamic Taint Analysis Framework." In *ISSTA*, 2007.

[Cou05] P. Cousot. "Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming." In *VMCAI*, 2005.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. "Termination proofs for systems code." In *PLDI*, 2006.

[CS02] M. Colón and H. Sipma. "Practical Methods for Proving Program Termination." In *CAV*, pp. 442–454, 2002.

[CSS03] M. Colón, S. Sankaranarayanan, and H.B. Sipma. "Linear Invariant Generation Using Non-linear Constraint Solving." In *CAV*, 2003.

[CVE03] "CVE-2003-0466." 2003.

[Den76] D. E. Denning. "A Lattice Model of Secure Information Flow." *Commun. ACM*, **19**(5):236–243, 1976.

[DRS03] N. Dor, M. Rodeh, and S. Sagiv. "CSSV: towards a realistic tool for statically detecting all buffer overflows in C." In *PLDI*, 2003.

[FOB05] J. C. Foster, V. Osipov, and N. Bhalla. *Buffer Overflow Attacks*. Syngress, 2005.

[GD07a] V. Ganesh and D. L. Dill. "A Decision Procedure for Bit-Vectors and Arrays." In *CAV*, 2007.

[GD07b] V. Ganesh and D.L. Dill. "A Decision Procedure for Bit-Vectors and Arrays." In *CAV*, 2007.

[GG75] J. B. Goodenough and S. L. Gerhart. "Toward a Theory of Test Data Selection." *IEEE Trans. Software Eng.*, **1**(2):156–173, 1975.

[GHJ07] A. Groce, G. J. Holzmann, and R. Joshi. "Randomized Differential Testing as a Prelude to Formal Verification." In *ICSE*, 2007.

[GHM08] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. "Proving non-termination." In *POPL*, pp. 147–158, 2008.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. "DART: directed automated random testing." In *PLDI*, 2005.

[GLM07] P. Godefroid, M. Y. Levin, and D. Molnar. "Active Property Checking." Technical report, Microsoft, 2007.

[GLM08] P. Godefroid, M.Y. Levin, and D. Molnar. "Automated Whitebox Fuzz Testing." In *NDSS*, 2008.

[God07] P. Godefroid. "Compositional Dynamic Test Generation." In *POPL*. ACM, 2007.

[Hol95] C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[JK97] R. Jones and P. Kelly. "Backwards-compatible bounds checking for arrays and pointers in C programs." In *Third International Workshop on Automated Debugging*, pp. 155–167. Linkoping University Electronic Press, 1997.

[JM05] R. Jhala and R. Majumdar. "Path slicing." In *PLDI 05*. ACM, 2005.

[Joh75] S.C. Johnson. "YACC – Yet another compiler-compiler." *Bell Labs Tehnical Report*, (32), 1975.

[JSS07] P. Joshi, K. Sen, and M. Shlimovich. "Predictive testing: amplifying the effectiveness of software testing." In *ESEC/SIGSOFT FSE*, 2007.

[Kap06] D. Kapur. "Automatically Generating Loop Invariants Using Quantifier Elimination." Technical Report 05431 (*Deduction and Applications*), IBFI Schloss Dagstuhl, 2006.

[Kin76] J. C. King. "Symbolic Execution and Program Testing." *Commun. ACM*, **19**(7):385–394, 1976.

[KL88] B. Korel and J. Laski. "Dynamic Program Slicing." *Information Processing Letters*, **29**:155–163, 1988.

[KM04] S. Khurshid and D. Marinov. "TestEra: Specification-Based Testing of Java Programs Using SAT." *Autom. Softw. Eng.*, **11**(4):403–434, 2004.

[Knu97] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1997.

[KY94] B. Korel and S. Yalamanchili. "Forward Computation of Dynamic Program Slices." In *ISSTA*, 1994.

[LA03] E. Larson and T. Austin. "High Coverage Detection of Input-Related Security Faults." In *USENIX*, 2003.

[LS75] M.E. Lesk and E. Schmidt. "Lex – A lexical analyser generator." *Bell Labs Tehnical Report*, (39), 1975.

[LS06] R. Lämmel and W. Schulte. "Controllable Combinatorial Coverage in Grammar-Based Testing." In *TestCom*, 2006.

[Mau90] P. M. Maurer. "Generating Test Data with Enhanced Context-Free Grammars." *IEEE Software*, **7**(4):50–55, 1990.

[MPL04] W. Masri, A. Podgurski, and D. Leon. "Detecting and Debugging Insecure Information Flows." In *ISSRE*, 2004.

[MSO06] "CVE-2006-5994." 2006.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufman, 1997.

[MX07] R. Majumdar and R. Xu. "Directed test generation with symbolic grammars." In *ASE*, 2007.

[MX09] R. Majumdar and R. Xu. "Reducing Test Inputs Using Information Partitions." In *CAV*, 2009.

[Mye79] G.J. Myers. *The art of software testing*. Wiley, 1979.

[Nis02] "The economic impacts of inadequate infrastructure for software testing." Technical report, National Institute of Standards and Technology, 2002.

[NMR02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs." In *CC*, 2002.

[NS07] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In *PLDI*, 2007.

[RL04] O. Ruwase and M. Lam. "A Practical Dynamic Buffer Overflow Detector." In *NDSS*, 2004.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[Sip97] M. Sipser. "Introduction to the Theory of Computation." pp. 91–101, 1997.

[SMA05] K. Sen, D. Marinov, and G. Agha. "CUTE: a concolic unit testing engine for C." In *ESEC/SIGSOFT FSE*, 2005.

[SN05] J. Seward and N. Nethercote. "Using Valgrind to detect undefined value errors with bit-precision." In *USENIX*, 2005.

[SSM04] S. Sankaranarayanan, H.B. Sipma, and Z. Manna. "Non-linear loop invariant generation using Gröbner bases." In *POPL*, 2004.

[ST85] D. Sleator and R. Tarjan. "Self-Adjusting Binary Search Trees." *J. ACM*, **32**(3):652–686, 1985.

[Tip95] F. Tip. "A Survey of Program Slicing Techniques." *Journel of Programming Languages*, **3**:121–189, 1995.

[VPK04] W. Visser, C. S. Pasareanu, and S. Khurshid. "Test input generation with java PathFinder." In *ISSTA*, pp. 97–107, 2004.

[VPP06] W. Visser, C. S. Pasareanu, and R. Pelánek. "Test input generation for java containers using state matching." In *ISSTA*, 2006.

[WCA02] E. Witchel, J. Cates, and K. Asanović. "Mondrian memory protection." In *Proc. ASPLOS*, pp. 304–316, 2002.

[Wei79] M. Weiser. "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method." *Ph.D Thesis*, 1979.

[WFB00] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities." In *NDSS*, 2000.

[Wit04] E. Witchel. *Mondriaan Memory Protection.* PhD thesis, Massachusetts Institute of Technologys, 2004.

[XA05] Y. Xie and A. Aiken. "Saturn: A SAT-Based Tool for Bug Detection." In *CAV*, pp. 139–143, 2005.

[XCE03] Y. Xie, A. Chou, and D. Engler. "ARCHER: using symbolic, path-sensitive analysis to detect memory access errors." In *ESEC/SIGSOFT FSE*, 2003.

[XMG08] R. Xu, R. Majumdar, and P. Godefroid. "Testing for Buffer Overflows with Length Abstraction." In *ISSTA*, 2008.

[ZLL04] M. Zitser, R. Lippmann, and T. Leek. "Testing static analysis tools using exploitable buffer overflows from open source code." In *SIGSOFT FSE*, 2004.

[ZLL05] M. Zhivich, T. Leek, and R. Lippmann. "Dynamic Buffer Overflow Detection." In *BUGS*, 2005.