Combining Static Analysis and Targeted Symbolic Execution for Scalable Bug-finding in Application Binaries

by

Muhammad Riyad Parvez

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Manual software testing is laborious and prone to human error. Yet, it is the most popular method for quality assurance. Automating the test-case generation promises better effectiveness, especially for exposing "deep" corner-case bugs. Symbolic execution is an automated technique for program analysis that has recently become practical due to advances in constraint solvers. It stands out as an automated testing technique that has no false positives, it eventually enumerates all feasible program executions, and can prioritize executions of interest. However, "path explosion", the fact that the number of program executions is typically at least exponential in the size of the program, hinders the adoption of symbolic execution in the real world, where program commonly reaches millions of lines of code.

In this thesis, we present a method for generating test-cases using symbolic execution which reach a given *potentially buggy* "target" statement. Such a potentially buggy program statement can be found by static program analysis or from crash-reports given by users and serve as input to our technique. The test-case generated by our technique serves as a proof of the bug. Generating crashes at the target statement have many applications including re-producing crashes, checking warnings generated by static program analysis tools, or analysis of source code patches in code review process.

By constantly steering the symbolic execution along the branches that are most likely to lead to the target program statement and pruning the search space that are unlikely to reach the target, we were able to detect deep bugs in real programs. To tackle exponential growth of program paths, we propose a new scheme to manage program execution paths without exhausting memory. Experiments on real-life programs demonstrate that our tool WatSym, built on selective symbolic execution engine S2E, can generate crashing inputs in feasible time and order of magnitude better than symbolic approaches (as embodied by S2E) failed.

Acknowledgements

First, I would like to thank Professor Paul Ward, for invaluable mentorship, without which, this thesis would not have been possible. I would like to thank Professor Vijay Ganesh, Dr. Glenn Wurster for their valuable time, and insightful comments that have shaped this thesis.

I would also like to extend my appreciation to my family and friends for their selfless support, care, and inspiration.

Dedication

This is dedicated to my father.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we describe the problem being addressed in this thesis and present a very brief overview of our contributions.

1.1 Motivation

A crucial aspect of software development is ensuring that the program behaves as the requirement. Anything that makes the program to deviate from its intended behavior is known as software bug/defect. Although not all software bugs are expensive, some bugs are simply benign, some software bugs are fatal enough to crash spacecrafts in air [1], make nuclear centrifuges spin out of control [66], or recall hundred thousands of faulty cars resulting in billions of dollars in damages [3]. Worse, security-critical bugs are arguably the most expensive bugs, tend to be hard to detect, harder to protect against, and up to 100 times more expensive after the software is deployed [16].

With the exception of a few safety-critical industries, such as avionics, automotive, or medical equipment, most of the software industry today relies on testing for its quality assurance. Testing a program consists of exercising multiple different paths by manually providing each input for each program path and checking whether "programs do the right thing." In other words, testing is a way to produce partial evidence of correctness for only the test-cases in regression test-suite, and thus increase confidence in the tested software. Test suites provide inadequate coverage of all the inputs a program could handle. For example, the test-suite of the Chromium browser contains about one hundred thousand tests [22]. This suite is thoroughly comprehensive by industry standards, yet it represents only a small fraction of all possible inputs the browser may receive.

Test suites also tend to be tedious to write and maintain. Statistics show that, on average, developers spend as much as half of their time doing testing [72].

Automatic testing allows developers and users to analyze programs for bugs and potential vulnerabilities. Variants of both black-box and white-box automated testing techniques work on the same goal: identify as many real bugs as possible with minimal amount of input from user. Each detected bug is usually accompanied by evidence, i.e., a test case, an input that forces the program to exhibit the unexpected behavior. Test cases eliminate false positives, ensure reproducibility, and provide the developer with concrete and actionable information about the underlying problem. Even if no bugs are identified, exercised test cases serve as a regression test suite, a standard software engineering practice that tries to detect new bugs introduced by future changes in program as early as possible [72].

One simple way to automatically find bugs is to perform random testing, popularly known as fuzzing [15], which is a program testing technique that runs the program on randomly generated (often randomly mutating already valid input) inputs. Fuzzers typically try extreme values, such as 0 and the native maximum integer. This form of testing is called "black-box" in general, because the input generation does not take into account the structure of the program under test. Although new fuzzers take program structure into account the structure of the program under test. Despite their conceptual simplicity, fuzzers are effective at discovering bugs [90] (albeit shallow) and are currently the state of the practice in automated testing. However, plain random fuzzers are ineffective at discovering corner-case bugs; bugs that manifest only under particular inputs in the program which can only exposed via systematically analyzing the program. Moreover, the vast majority of the generated inputs are therefore redundant because they force the program to execute already covered code.

Formal verification [47] is a semi-automated technique to prove program-correctness that needs developer written formal program specification. seL4 [63] is such a recent effort of formally verifying an operating system kernel. Although formal verification can guarantee certain programs to be bug-free, switching to a formal development model is not feasible for most of the programs because of very high requirement of both resources and development time. Despite recent advancements, which have brought down the cost of building formally-proven software, it still takes on the order of person-years to develop a few thousand lines of verified code. This rate is currently unsustainable for most commodity software. Therefore, formal verification is not viable outside safety-critical or mission-critical systems.

Another automated way to detect bugs in program is using static program analysis [54]. Static program analysis analyze the programs without actually executing the programs;

instead, static analysis tools work on a simplified abstract version of the programs. The abstraction of the program that static program analysis works on allows itself to scale to large programs. But static analysis tools tend to generate lot of warnings and significant percentage of generated warnings are not severe enough to be fixed. Static program analysis tools are not used widely because of large amount of non-actionable warnings [30] [60]. It is often prohibitively expensive to manually check/fix all the warnings generated by static program analysis tools. The goal of this thesis is to provide a technique that will automate the process of checking security-critical bugs from warnings generated by static program analysis tools.

An increasingly popular program analysis technique is symbolic execution [33] [59] [62]. Unlike manual or random testing, symbolic execution systematically explores the program by analyzing one program path at a time and then checks for potential bugs in that path. For every analyzed program path, symbolic execution checks whether under which all possible executions of the path are safe or doest not violate the specification of the program by generating a condition (logical formula) that asserts program safety for all possible executions. If the formula is falsifiable, specification can be violated and a counterexample (a witness/proof of specification violation), i.e., a test-case is generated. Having a test-case to reproduce a bug not only ensures the existence of bug, but also gives programmers great help for debugging it. Over the past decade, numerous symbolic execution tools have appeared showing the effectiveness of the technique in a vast number of areas. Unlike formal verification, symbolic execution can provide partial path-based verification (ensures certain types of errors will not occur in verified program paths). KLEE [39], S2E [44] are some of the most popular symbolic execution engines.

Albeit a well-researched and very useful technique, symbolic execution still faces two significant scalability challenges.

The first challenge, originates from the path-based nature of the technique: number of program paths grows exponentially with number of conditional branch statements in real-world programs known as "path/state explosion"[1] poses a significant challenge for analyzing large programs, where the number of paths is typically very large. Path explosion is a well studied problem that persists throughout modern symbolic execution engines [39] [42] [44] [57].

The second challenge is reasoning about feasibility of program paths. A program path is feasible if there exists at least one input that can drive execution to that path. In other words, if logical formula that represents the program path is satisfiable then the

[1]Depending on the context, "path" "execution state" may be used interchangeably, an execution state corresponds to a program sub-path explored so far.

path is feasible; otherwise, it is infeasible. Unfortunately, checking whether a formula is satisfiable is a hard problem, the complexity of solving formulas depend on the domain of logic. Although modern constraint solvers can solve most of the generated formulas very quickly [32], some formulas can still be the bottleneck. To mitigate the high solving times, the symbolic execution community has invested a lot time and effort in developing countermeasures, such as caches [39], and simplifications [57], which mitigate the problem but do not solve it in the general case.

One simple way to check whether a warning generated by a static analysis tool is an actual bug is to synthesize an input that will execute potentially buggy program statement[2] with the values needed to reveal the bug pointed out by the warning. Since symbolic execution can automatically explore different program paths and generate concrete input for that path. Symbolic execution can be used to explore feasible program paths that execute the target instruction and check whether there is any execution on that path that can violate program specification; if such an execution is found, symbolic execution generates a test-case as a proof of violation. The presence of bugs can be verified by running the generated test-case independently.

This thesis focuses on checking the warnings generated by static analysis tools using symbolic execution. More specifically, we describe an heuristic search strategy, usually known as targeted search, to find the execution (i.e., the test-case) that can violate program specification at the site pointed out by static program analysis tools. We also describe heuristics to mitigate path/state explosion problem of symbolic execution. The problem of finding feasible paths to target has been addressed by previous research, however, few techniques cope with real-world binaries since they mostly work on intermediate representation. Program binaries present challenges for program analysis techniques due to their lack of high-level semantic information lost in compilation process. Recovering such information dynamically is often infeasible. But analyzing binaries is important, because often developers do not have source code for third-party libraries. Therefore, whole program dynamic analysis only feasible on the program binary. Moreover, focusing on binaries provide us the ability to different types of programs including operating system kernels [74].

Formally the problem statement can be stated as follows:

"Given a potentially vulnerable instruction in a program binary, synthesize an execution (concrete input that will execute the instruction with specific values needed) to exploit that vulnerability."

[2]We use the terms program instruction and program statement indistinguishably.

1.2 Contribution

By combining static binary analysis with targeted symbolic exploration, this thesis makes the following contributions:

- We propose a targeted search strategy, a best-first ("best" program path that can execute the target with the lowest amount of effort at that instance, described in detail in section 2.2.6) search strategy tailored for security bugs to find an execution (input values) that exercises target instruction with the values needed to reveal the bug. Our proposed distance estimation heuristic introduces the concept of "target loops", and "target function" in targeted search in addition to branch distance heuristic [38]. We also introduce the concept of "bypassing the target" to detect whether any execution has bypassed the target instruction instead of executing the target. We propose a very precise heuristic to detect "bypassing the target" based on intra-procedural control-flow graph of the target function.

- To mitigate "path explosion" [41], we propose a technique for pruning program paths that can not reach the target instruction during targeted symbolic exploration. Our technique uses the notion of critical edge in control-flow graph for a target instruction and detects critical edges [91] in the control-flow graph of the program binary. Based on the detected critical edges, we prune the paths that do not follow the critical edges, i.e., can not reach the target instruction. We also propose another path pruning technique tailored specifically for security bugs to mitigate the path explosion problem caused by loops.

- In addition to "path explosion" problem, online symbolic execution engines also suffer from "memory explosion" [42] problem because of exponential growth of number of program paths, i.e., execution states. We propose a new execution state management scheme to address memory explosion problem of online symbolic execution engines for targeted exploration. To keep memory usage limited, we propose a technique that "suspends" execution states that are not nearer than other execution states to the target at that instance. Suspended execution states can be resumed later if all the active execution states are done processing.

- We implement our proposed targeted search technique, heuristics for tackling path, and memory explosion in a tool called WatSym, a targeted symbolic execution tool based on S2E [44] an hypervisor built on top of QEMU [20], an open-source hypervisor, and KLEE [39], an open-source symbolic execution engine. WatSym is capable

of executing the whole operating system symbolically. By implementing on top of S2E, WatSym can execute vast majority of programs without the source code and analyze them in real environment (e.g., real shared libraries and operating system) instead of model of the environment (simplified and verification-friendly emulation of real environment). We also evaluate WatSym on three real world bugs including a security vulnerability and show that WatSym outperforms traditional symbolic execution tools.

1.3 Thesis Outline

The rest of the thesis presents in detail the design, implementation, and evaluation of our proposed targeted search strategy implemented in tool WatSym.

Chapter 2 provides more background on widely used primitives in program analysis, symbolic execution, different search strategies used in symbolic execution, and common problems in symbolic execution including the line reachability problem.

Chapter 3 surveys the work related to our contributions and positions our contributions with respect to existing related work.

Chapter 4 presents the new concepts we have introduced, and our contributions. It presents our targeted search heuristic, heuristics for tackling path explosion caused by loops, and new execution state management scheme for tackling memory explosion.

Chapter 5 describes implementation details of our tool WatSym, and S2E [44].

Chapter 6 provides experimental evaluation of WatSym and S2E [44] on finding real-world bugs.

Chapter 7 describes theoretical, and practical hardness of the problem we are addressing and future direction of research.

Chapter 8 ends the thesis with conclusions.

Chapter 2

Background

In this chapter, we provide background for topics discussed in this thesis and provide definitions for concepts relevant to this thesis.

2.1 Program Analysis

To make targeted search efficient, we use both static and dynamic program analysis to steer the execution to the target and trim down the search space. In this section we provide useful definitions related to program analysis.

Definition 2.1.1. Basic Block: A basic block is a maximal set of contiguous program instructions with only one entrance instruction and only one exit instruction (not necessarily a jump instruction). Whenever the first/entrance instruction in a basic block is executed, the rest of the instructions are executed exactly once, in order provided that the execution does not hault for errorneous conditions.

Definition 2.1.2. Program Execution Path: A program execution path is the sequence of instructions that are executed during program execution. If there exists at least one input that can force the program to execute a particular program execution path, that path is known as feasible program path. If no such input exists for a particular program path, that program path is known as an infeasible program path.

Definition 2.1.3. Control-Flow Graph: A control-flow graph (CFG) is a directed graph representation of a program. The nodes of a CFG represent the program's basic blocks

while directed edges represent jumps from basic blocks (i.e., control-flow). The CFG representation is essential for many static analysis such as reachability and dominator analysis, and program optimization.

Definition 2.1.4. Branch Distance: Branch distance is the number of conditional branch instructions in the shortest path from one basic block to another basic block in the control flow graph. Branch distance is usually calculated in weighted CFG where each edge for conditional branch is assigned weight one and other edges for unconditional branch instructions are assigned weight zero.

Definition 2.1.5. Call Graph: A call graph is a directed graph that captures the calling relationship between procedures of a program. Each node corresponds to a procedure and each edge (p_1 , p_2) indicates that procedure p_1 may call procedure p_2. The root nodes of call graph represent program entry points or procedures; usually executables have one entry point and shared libraries have multiple entry points. Call graphs, similar to CFGs, can not be entirely extracted by static analysis because of memory/register indirect jumps (e.g., function pointers in C/C++ programs).

```
1  if (input < 100)
2    f(0);
3
4  if (input > 100)
5    if (input > 200)
6      f(input)
7
8  void f(int x) {
9    if (x == 999)
10     abort()
11 }
```

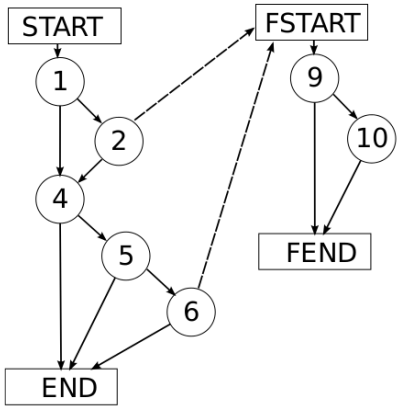

Figure 2.1: Source code snippet and associated inter-procedural control-flow graph with return edges omitted. Each basic block is labeled with the line number it corresponds to [69].

Definition 2.1.6. Inter-procedural Control-Flow Graph: An inter-procedural control-flow graph (iCFG) is a combination of a program's call graph and the control-flow graph of each

procedure.[1] Specifically, each call graph node is replaced with the CFG corresponding to its associated procedure, all edges pointing to it are redirected to the procedure's entry basic block and all edges pointing out of it are assigned to the procedure's basic blocks which contain the actual function calls. Figure 2.1 shows a simple code snippet and its associated CFG.

2.2 Symbolic Execution

Symbolic execution [62] is a program analysis technique that enables reasoning about program correctness in the domain of logic. Informally, we can view symbolic execution as a way of executing programs that contain symbolic values. A symbolic value is defined by the symbol and the set of concrete values (an instance of the value type) it can range over. For instance, we can define α to be a symbol for a 32-bit integer variable that can range over any value from the set of all 32-bit integers (such a set can be viewed as the type of the symbol).

There are two variants of symbolic execution: (1) Static Symbolic Execution, and (2) Dynamic (Path-based) Symbolic Execution.

2.2.1 Static Symbolic Execution

Static symbolic execution is a program verification technique that translates the whole program into logical formulas, where the formulas represent the desired program property over any program path. Static symbolic execution verifies specific properties of the program under certain assumptions (e.g., number of loop iterations is bounded). In other words, it is sound and complete under certain assumptions with respect to the specific program properties. Potential property violations in program are encoded as logical assertions that will falsify the formula if that property is violated.

Static symbolic execution is doomed to perform poorly whenever precise static transformation to logical formula is not possible. Unfortunately, this is frequent in practice due to complex program statements (pointer manipulations etc.) and calls to operating-system and library functions that are hard or impossible to reason about statically with good enough precision. Moreover, the generated logical formulas are very hard for underlying

[1] In this thesis, we use the term CFG to refer to inter-procedural CFG for the sake of simplicity; when we need to refer to intra-procedural CFG, we will explicitly mention it.

constraint solver because the logical formula that represents all program execution paths is many times larger than the formula that only represents only a specific program path. Because of this, static symbolic execution is not widely used in practice. Calysto [27] and Saturn [46] are static symbolic execution tools.

2.2.2 Dynamic Symbolic Execution

Dynamic symbolic execution is a way of interpreting programs that have symbolic values.[2] First, the analysis performed on the program is symbolic; instead of operating on concrete inputs (e.g., instances of the value type), program inputs are substituted by symbols (variables) that represent all possible inputs. Second, the analysis is an execution; program instructions are evaluated in the forward direction, similar to an interpreter, program values are computed as a function of the symbolic input values, and a symbolic execution context mapping from each variable to a symbolic expression expressed in terms of the symbolic input values is maintained throughout execution. For each path, symbolic execution builds up a logical formula that represents the condition under which the program will execute the exact same program path. The logical formula is expressed in terms of the symbolic input variables and is called the *path predicate* or *path constraints*.

A symbolic execution state is defined by: (1) the path predicate, and (2) current symbolic execution context. Each execution state represents a feasible program execution path[3]. A symbolic execution state allows us to check specific properties on that program path. For example, checking the validity of assertions (e.g., does this assertion always hold for this program path? In other words, is there any input value that will drive the execution to this path and also violates this assertion?), and thus reason about correctness. Dynamic symbolic execution can be viewed as partial path-based verification. For large programs where full verification is infeasible, partial path-based verification is particularly useful.

A program can be treated as a collection of feasible program execution paths. For example, a program consists of one conditional statement "`if (x>0)then ... else ...` " can be viewed as a collection of two feasible paths: one that satisfies the branch condition `x>0` and another that satisfies the branch condition `x<=0`. To execute both of these paths,

[2]In this thesis, we use the term symbolic execution to refer to path-based dynamic symbolic execution; when we need to refer to static symbolic execution, we will explicitly mention it. Some authors [56] have used the term dynamic symbolic execution synonymous with concolic testing. However, we disagree with that definition and in this thesis, dynamic symbolic execution and concolic testing are different.

[3]Depending on the context, the two terms may be used interchangeably [41]. An "execution state" corresponds to a program path to be explored.

```
void autoShiftGear(unsigned &rpm,
                   unsigned &gear)
{
  if (rpm > 2500) {
    gear = gear + 1;
    rpm = rpm * 0.5;
  } else if (rpm < 1000) {
    gear = gear - 1;
    rpm = rpm * 2;
  }
}
```

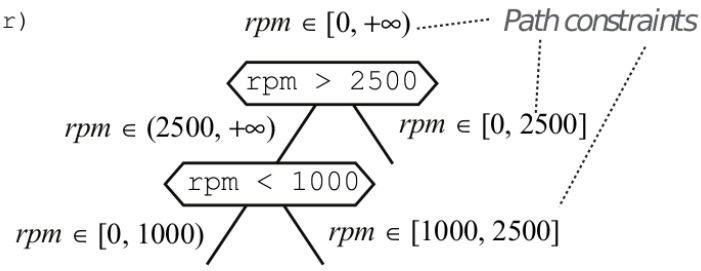

Figure 2.2: Symbolic execution creates an execution tree with path constraints [43].

it is not necessary to try all possible values of `x`, but rather just one value greater than 0 and one value less than 0.

A symbolic execution engine explores this set of paths into a symbolic execution tree, in which each possible execution corresponds to a feasible program path from the root of the tree to a leaf corresponding to a terminal execution state. To do so, (at least one) symbolic inputs have to be provided to the program, i.e., instead of setting an input variable `x` to a concrete value (e.g. `x=0`), it is viewed as a set λ of all possible values `x` could take. Then, any time a branch instruction (with predicate p) accesses at least one value that is depends (directly or indirectly) on symbolic value, `x`, execution is split into two different executions `ExecutionState_i` and `ExecutionState_j`, two copies of the program's current execution state are created, and `ExecutionState_i`'s path constraints record that the variables accessed by the branch instruction must be constrained to make p true, while `ExecutionState_j`'s path constraints record the negation of the constraints so that p must be false. This is known as "forking" of execution states. In fig. 2.2, `rpm` is marked as symbolic, i.e., it can hold any value allowed by a 32-bit unsigned integer. When execution reaches the first branch, execution forks into two different executions, one of them proceeds with $rpm > 2500$ as a constraint added to the path constraints, the other with $rpm \leq 2500$. The process repeats recursively: `ExecutionState_i` may further fork into `ExecutionState_i_i` and `ExecutionState_j_j` , and so on. A node s in the tree represents an execution state (that includes both symbolic and concrete state) of the program, and an edge `ExecutionState_i` $\rightarrow$ `ExecutionState_j` indicates that `ExecutionState_j` is `ExecutionState_i`'s successor on any execution path satisfying the constraints in `ExecutionState_j`.

Symbolic execution relies on a constraint solver to decide which program paths are

feasible and to compute concrete input values that can be used as test cases. In the example of fig. 2.2, when execution reaches the first branch statement, the symbolic execution engine queries the constraint solver whether both program paths are feasible. For this, the engine sends queries $rpm > 2500$ and its negation to the solver under the current path constraints $rpm \in [0, +\infty)$. The solver replies that given the constraints, both queries are satisfiable, i.e., both paths are feasible. When the second branch is reached, the solver checks that $rpm < 1000$ is feasible under the constraints $rpm \in [0, +\infty) \wedge rpm \leq 2500$. This process goes on until all the feasible program paths are explored.

If constraint solver finds path constraints for a program path, hence an execution state, is unsatisfiable, the engine refrains from creating a new state for that infeasible path. Finally, when the execution states terminate (e.g., the program exited successfully, crashed, or terminated forcefully), the solver can compute the range of concrete values for the given symbolic inputs that will exercise the respective program paths. The generated concrete input, i.e., test-case, can be useful to reproduce bugs, such as crashes or assertion failures.

Current implementations of symbolic execution can be broadly divided into two categories:

- Offline Symbolic Execution
- Online Symbolic Execution

2.2.3 Offline Symbolic Execution

Offline symbolic execution only explores one program path at a time. Program source code or binaries are instrumented to maintain shadow symbolic state of the concrete state. The program under test is executed with a concrete input, the instructions program also maintains a shadow symbolic state for the path exercised by the concrete input. After the end of concrete execution, symbolic path constraints from previous are modified by negating one of the constraints at a branch point to take the not-taken branch. A new test-case is generated by solving the modified formula provided that the new formula is satisfiable. This process is known as "flipping the branch" which means selecting a branch-point and flip the constraint associated with the taken branch in previous run to generate a new test-case that will take the not-taken branch. Test-cases for new program paths are generated offline, hence the name offline symbolic execution engine. This approach is known as *concolic* (**conc**rete+symb**olic**) testing [56], a juxtaposition of both concrete and symbolic execution as the program is mainly executed in concrete mode but also a shadow

symbolic state is maintained for the respective path dictated by the given concrete input. The initial input needed for offline symbolic execution is known as seed input. Offline symbolic execution engines are CUTE [77], DART [56], and Sage [57].

The main benefit of offline symbolic execution is the simplicity of implementation and low resource requirement. Offline engines only follow the program path dictated by the given concrete input, thus, do not have to fork new execution states or switch between different execution states in run-time.

The main disadvantage with offline symbolic execution is inefficiency. For every explored program path, we need to first re-execute a (potentially) very large number of instructions until we reach a branch instruction where new program path is forked, and then begin to explore new instructions. This is because many program paths share common program sub-paths. Although the cost of concrete execution of instructions is negligible in practice, the cost of symbolic execution of instructions is often prohibitive.

Concolic execution is faster than symbolic execution of single program path since it does not have to fork new execution states and check feasibility of new program paths. However, we rarely analyze only one program path; for analyzing multiple program paths, symbolic evaluation of same instructions (instructions in common sub-path prefix in different programs paths) for each program path results in redundant work, and inefficiency. Further, offline symbolic execution needs a concrete test case/seed input to begin the analysis.

2.2.4 Online Symbolic Execution

Online symbolic execution checks programs by systematically enumerating program paths in single run. At every symbolic conditional branch, the engine checks the feasibility of path following each branch target and "forks" a new execution state to explore each feasible branch target. Online engines fork new execution states that follow new feasible program paths, maintain different execution states, and switch between different execution states in similar way to an operating system maintaining different processes of same program and switching bet'ween those processes. Because of the forking of execution states, online engines can explore multiple program paths without having to execute same instructions multiple times. Another advantage of online symbolic execution is it can execute the program without any concrete input since it can find feasible program paths by itself. KLEE [39], S2E [44], and Mayhem [42] are online symbolic execution engines.

Online symbolic execution engines can analyze a specific set of instructions into two

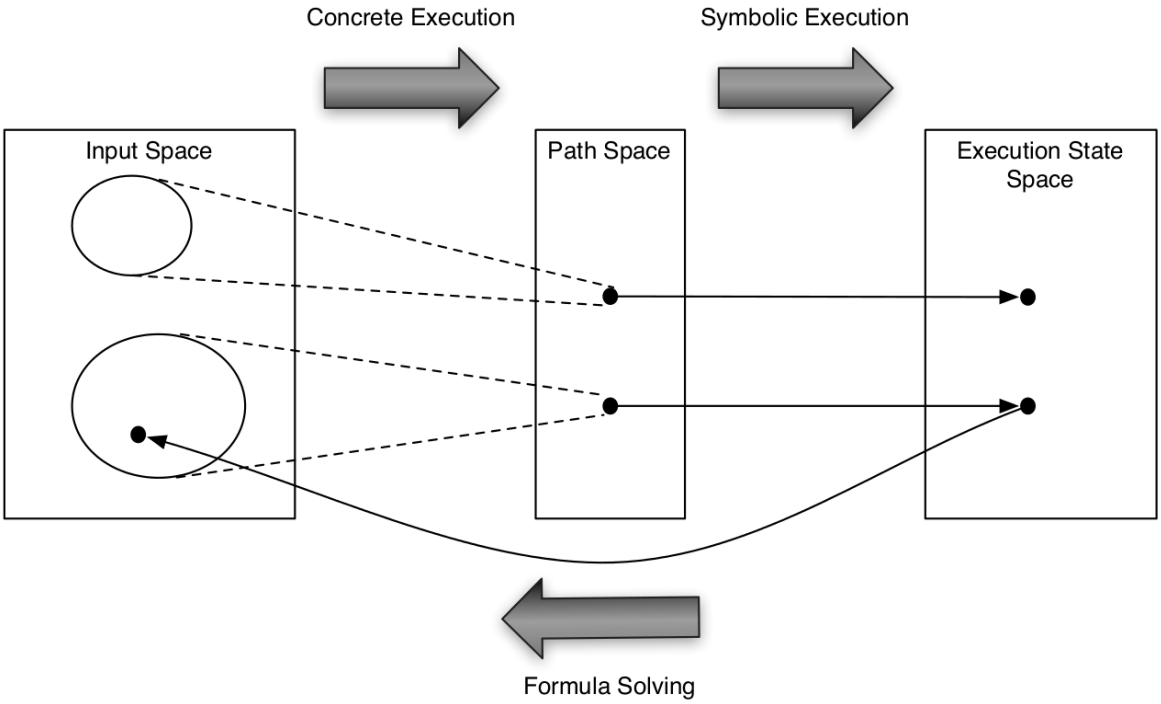

Figure 2.3: Inputs, paths, execution states and their connections [25].

different modes: (1) multi-path[4], and (2) single path mode. In single path mode, online engines do not fork new execution states for feasible program paths as opposed to multi-path where execution states fork new execution states for each feasible symbolic branch. Single path mode is important to focus the analysis on program paths of interest (by not forking new execution states in the environment) and to mitigate the exponential growth of program paths known as path explosion problem. Online engines can switch back and forth from single path to multi-path mode for each execution state.

Concrete vs Symbolic Execution. Not all variables in program hold symbolic values even if the program is given all symbolic inputs. During execution if it is determined that an instruction will not access any symbolic data, the instruction can safely be evaluated concretely; this is known as concrete execution. Concrete execution of an instruction is as fast as normal execution of an instruction.

[4]In this thesis, for the sake of simplicity, we use term "symbolic exploration" synonymous with online symbolic execution in multi-path mode.

To reduce the overhead of symbolic evaluation of instructions, many online engines use lightweight data-flow analysis [54] to determine whether or not an instruction will access symbolic data. Online engines mix concrete and symbolic execution of instructions based on the type of data (symbolic or concrete) an instruction will access; this is known as "mixed execution."

2.2.5 Path Explosion

Every conditional branching instruction in the program potentially doubles the number of program paths/execution states that need to be explored. In the presence of unbounded loops, the number of program paths becomes unbounded. This phenomenon of exponential growth of the number of paths is known as the "path explosion" problem [39].

Moreover, path explosion also causes memory explosion for online symbolic execution engines. In multi-path mode, online symbolic execution engines fork a new execution state for each feasible path, exponential growth in path also means exponential growth in number of execution states which results in memory explosion. Moreover, the symbolic domain of the execution state itself grows linearly with the number of executed instructions by that state. Although many execution states have common data among them due to common sub-path and it is possible to share common data among execution states, it is often not adequate to tackle exponential growth of memory requirement.

2.2.6 Search Heuristics for Symbolic Exploration

For non-trivial programs, it is infeasible in practice to explore all program paths even if the number of feasible program paths is bounded. Therefore, only a fragment of the feasible program paths can be explored. In other words, only a limited number of states can be explored among a huge number of execution states; which one should be explored next? This is a scheduling problem and is known in the literature as the path selection or execution state prioritization problem. A search strategy of an online symbolic execution engine dictates which state to execute from the current set of execution states.

Different programs behave differently and identifying which states need to be explored for an arbitrary program is hard. Nevertheless, finding well-tuned heuristics that work well for specific domains is an active area of research.

We categorize strategies into two main categories: coverage optimized search and targeted search.

Coverage-optimized search Symbolic execution is typically used for test case generation. Generating test cases that achieve higher code coverage gives the analyst higher confidence about the correctness of the tested software. Coverage optimized search strategies prioritize the execution states that are more likely to explore uncovered program regions; hence they try to improve code coverage of the program. KLEE [39] was the first symbolic execution engine that demonstrated high code coverage can be achieved on a diverse set of utilities written in C. KLEE employed a number of coverage optimized search heuristics to select states. Almost all the available online engines [39] [42] [44] [57] [83] have built-in coverage-optimized search.

Targeted search The targeted search problem is given a program statement, find an input that drives program execution to that statement. This problem is known as the program-statement reachability problem. Automatically finding a feasible program path, i.e., an input that will execute a specific program statement is in general an undecidable problem.

Targeted search strategies prioritize the execution states that are more likely to execute specific program statement and use both static and dynamic program analysis to trim down the search space that would otherwise be too large to explore in a naive approach. Targeted search strategies are particularly useful for checking static analysis warnings [58], reproducing crashes [73] [91], finding root cause of a crash [92], and fault localization [24].

Path selection/state prioritization algorithms are based on heuristics. The hard nature of path based program analysis (undecidability, exponential growth and so on) makes behavior of these algorithms on arbitrary programs unpredictable. However, it is possible a domain-specific fine-tuned heuristic can be useful, practical, and efficient on many programs.

Chapter 3

Related Work

In this chapter, we briefly describe the existing symbolic execution based program analysis techniques and works related to our contribution. We start with a brief overview of the symbolic execution engines that made the technique applicable to real-world software and also state of the art symbolic engines.

Dynamic symbolic execution [62] as program analysis technique has been popularized by Concolic Unit Testing Engine (CUTE) [77], Directed Automated Random Testing (DART) [56], and EXE [40]. Later KLEE [39], Sage [57], Mayhem [42], and S2E [44] have made symbolic execution based program analysis practically usable. There are numerous prior approaches for symbolic execution based on program source code, intermediate representation, or program executable binary. We provide brief overview on related work of targeted symbolic execution.

3.1 Targeted Search

Burnim et al. [38] first introduced CFG-directed search which used branch distance as coverage-optimized search strategy where the execution states that are nearest to any uncovered program region are prioritized. Although CFG-directed search has been proposed for improving program coverage, it needs to compute all-pair-shortest-paths between basic blocks in the program which is prohibitively expensive. CFG-directed search and its variants are used extensively as targeted search strategy since computation of single source shortest path is cheap.

Execution Synthesis Debugging (ESD) [91] is a debugging tool that reproduces crashes (crashing inputs and thread interleaving) of concurrent programs. ESD finds current instruction of each thread at the time of the crash from the core-dump, determines cause of crash and tries to synthesize an input, and a thread interleaving that will reproduce the crash. ESD uses minimal proximity heuristic (i.e., number of instructions) from the currently executing instruction to the instruction at the time of the crash in the shortest path if the target function in its call stack. ESD does not detect when an execution state has bypassed the target and still has the target function in its call stack. In that case, ESD becomes slower because ESD continues processing of a state that has already bypassed the target.

Hercules [73] is another crash-reproducing tool based on S2E. The targeted search strategy of Hercules tries to find the reason why some program paths that go through target are infeasible from unsatisfiable core or *UNSAT core* of the path constraints. The unsatisfiable core of a formula is minimal subset of clauses that are unsatisfiable; in other words, an UNSAT core is one of the reasons why a formula is not satisfiable because there can be more than one UNSAT core. Since every clause in the formula is associated with a taken branch so far, Hercules identifies a minimal subset of taken branches that make the program path infeasible. It then flips the last taken branch in the subset, i.e., satisfying the reason of unsatisfiability to find a feasible path to the target instruction.

Babic et al. [28] used the shortest path on the *Visibly Push-down Automata (VPA)* [29] graph as a distance estimation heuristic for targeted search strategy. Paths enumerated from a context-insensitive CFG do not maintain the restriction of well-matched function calls and returns. If paths do not maintain the restriction of well-matched function calls and returns, a callee function can return to other function than the calling function because it is allowed in context-insensitive CFG. The authors have also introduced *loop-pattern heuristic* which tries to find a pattern of taken branches inside loops to find loop-induced overflows.

Dowser [58] finds buffer overrun in programs using targeted search to verify the warnings generated by its own developed static analysis. Similar to works of Babic et al. [28], Dowser assigns each branch in the loop a score based on the probability that the branch will lead to a new way of doing complex pointer arithmetic. The idea is that complex pointer arithmetics are hard to reason about for developers and are more susceptible to causing buffer overrun. For example, a pointer arithmetic that has multiplication is more complex than a pointer arithmetic that has only addition. Dowser also ranks the generated warnings by the complexity of surrounding code regions of the warnings. For example, a pointer arithmetic inside a loop is more complex than a pointer arithmetic outside of loop. Because complex code regions are hard reason about and also lead to complex pointer arithmetic.

Katch [71] introduced patch-testing technique using targeted symbolic exploration to test program statements modified by patches. Katch uses existing regression test suite to find the closest test case to use as seed input for symbolic exploration and uses a variant of CFG directed search that 'flips branch' iteratively to get closer to the target.

Requiring Target Instruction Exercising Input. Many targeted search heuristics [58] [70] [71] [73] [81] need a test-case/feasible program path that exercises the target instruction. Existing target exercising test-case is used as seed input for symbolic exploration. Seed inputs provide symbolic context for symbolic exploration, an initialization point in the search space, takes symbolic exploration closer to the target, and provides symbolic exploration with invaluable sub-path prefix. A sub-path prefix is the sub-path from a program entry point that is shared by two different program paths. With the symbolic context, a feasible sub-path prefix to the target, and other information gained from dynamic analysis (e.g., relevant input bytes to the target) make synthesizing an input significantly easier albeit still a hard problem. The requirement of a target instruction exercising test-case limits the applicability of targeted search tools. Very few programs have a regression test-suite with more than 80% code coverage.

Detecting Paths that Bypass Target Instruction. Detecting whether an execution state has bypassed the target is very important for targeted exploration. Because if a state has bypassed the target, continuing the execution of that state can be counter-productive. To our best knowledge, only Zero Effort Software Testing Infrastructure (ZESTI) [70] addresses the issue of target bypassing. ZESTI estimates maximal number of instructions or basic blocks need to be executed to reach the target. ZESTI stops exploration of a path if the target has not been executed after execution of the estimated number of instructions and a new execution path is selected. This is known as *depth-bounded symbolic execution* as the depth of the symbolic execution tree is bounded. *Depth-bounded symbolic execution* of ZESTI does not detect whether the state has bypassed the target or not; ZESTI assumes it has either bypassed the target or drifted away from the target if the execution did not reach the target after executing estimated number of instructions.

Run-time Information. Crash-reproducing tools [73] [91] uses the core-dump of the crash which provides rich run-time information (e.g., the function call stack at the time of the crash, value of the variables, the value of the pointer in the case of a segmentation fault) that can be used during reproducing crashes.

Escaping Local Minima in Search Space. Targeted search strategy implemented in offline symbolic execution engines [56] [71] [77] are vulnerable to local minima in search

space. Search process can get trapped in local minima and often can not escape local minima without introducing randomization in the search process.

Semantically Rich Static Analysis. Many targeted search tools [58] [70] [71] [91] work on the assumption on the availability of program source code. Program source code provides rich semantic information for the program. Rich semantic information makes static analysis more precise and more structural information can be utilized during run-time of symbolic exploration. Static analysis on program binary is a very hard problem (both in theory, and practice) because almost all high-level semantic information is discarded during compilation process of binary generation. Although static binary analysis is very active area of research [34] [79], recent tools working on binary are far more imprecise than tools working on source code.

Backward Targeted Search. Mao et al. [68] introduced *Call-Chain-Backward Symbolic Execution* as a guiding technique for directed backward symbolic execution, which starts from the target program instruction and works backward until it finds a feasible path from an entry point of the program to the target instruction. The authors have also introduced a mixed backward-forward search strategy based on both forward search to the target and backward search from the target to a program entry point. Although the authors have found that mixed backward-forward search strategy is often faster than simple forward or backward search strategy in relatively simple programs, applying such backward search in real environment is highly complex and its benefit yet to be seen.

3.2 Path Pruning

Control-Flow Graph Based Path Pruning. Brumley et al. [35] prunes program paths by computing *chop* of CFG. *Chop* of CFG only contains the relevant sub-graph (strongly connected component that contains the target) to the target. Therefore, any program path in CFG that exits the *chop* can be pruned [73]. Chop provides very strong path pruning condition where pruned paths are guaranteed not to reach the target provided that the extracted CFG is complete. Because of this strong guarantee, chopping is often ineffective because for large real-world program, often chop removes very few or no basic blocks in the program. Hence, very few program paths can be pruned by chopping and often the target search itself manage to keep the search focused to chopped region without explicitly chopping the CFG.

ESD uses *critical edge* concept to introduce sub-goals concept in targeted search. Critical edges are the control-flow edges that must be taken to reach the target and sub-goals are

the program statements that have to be reached before reaching the target. ESD detects critical edges for the target/goal block and uses reaching definition to find the sub-goal basic blocks that will ensure the execution will take critical edges in future. Exercising sub-goal blocks ensures that the execution will follow the critical edges.

Data-Flow Based Path Pruning. Katch [71] uses CFG pruning also referred to as path pruning technique based on data-flow analysis on the program. Katch tries to detect program branches which if taken by the execution, will not reach the target block. Katch prunes those control-flow edges in CFG which defers symbolic execution to take those branches. Pruned branches are detected based on the *weakest pre-condition* for the target block. Weakest pre-condition is an essential but not sufficient condition that an execution has to satisfy to reach the target. Hence, any program paths that do not satisfy weakest pre-condition can be pruned.

Data-flow based pruning is only possible when high level semantic information about program is available which is unavailable in program binary. Although it is possible to apply data-flow analysis on binary, in practice, the lack of high level information makes the analysis too imprecise to use.

Program Structure Based Path Pruning. Hercules [73] extracts Module Dependency Graph (MDG) of the program (modules are shared libraries and executable binaries) and use this information to prune program paths that do not lead to crashing module. Essentially Hercules uses chopping on MDG to prune program paths. Hercules also uses chopping on CFG to prune program paths.

Pruning Error Program Paths. Document Aided Symbolic Execution (DASE) [87] first introduced the concept of using documentation of a program or specification of program input to assist symbolic exploration. DASE extracts constraints for valid input by mining the program documentation and input specification and later uses the input constraints to detect error paths (paths that are followed by execution if invalid input is provided) in the program. By pruning the error program paths, DASE helps the search strategy to focus on deep program states and execute uncovered lines of code. Similar idea can be applied to targeted search strategies to detect whether the target can only executed by following some error program paths or vice-versa. In such case, the search strategy can only focus on error program paths.

Any path pruning technique based on program CFG is unsound, i.e., a feasible path that will reach the target can get pruned because of incompleteness of CFG.

3.3 Tackling Path Explosion by Loops in Symbolic Exploration

Loops in the programs are notoriously known for causing path explosion and unbounded loops cause unbounded number of program paths. Moreover, loops cause "memory explosion" for online execution engines and make the search space for targeted exploration unbounded. Despite the cost of loops, many online symbolic execution engines are loop oblivious [39] [44]. In other words, they do not employ any specific heuristic to tackle the explosion caused by loops.

Most of the techniques for handling loops fall into three categories: (1) bounded iteration, (2) search-guiding heuristics, and (3) loop summarization [88].

Bounded Iteration. *Bounded iteration*, the most commonly used technique, unrolls loops bounded number of times in run-time. In other words, number of iterations of a loop is kept bounded in run-time. It is especially useful for coverage-optimized search strategy where repeated execution of loops does not increase code coverage. Bounded iteration for loops has also been applied in targeted search [73]. Although bounded iteration is useful in coverage-optimized search strategy, some program behaviors are only observed when loops are executed repeatedly. This is especially true for security vulnerabilities like buffer overflow and integer overflow. Bounded loop iteration can make revealing loop induced vulnerabilities impossible.

Loop Summarization. *Loop summarization* summarizes a loop into a set of logical formulas which abstracts every iteration of loops. Therefore, the engine do not get stuck in executing loops. Although summarization addresses the path explosion problem, but it requires finding loop invariants and loop induction variables which for many loops are not feasible. Also, queries generated by loop summarization are often hard for underlying constraint solvers.

Search-guiding Heuristics. Search-guiding heuristics attempt to guide symbolic exploration to focus on program paths in loops that are more likely to execute the target instruction. Babic et al. [28] have focused on finding a pattern of branches in loop(s) that are essential to cause an overflow. Dowser [58] focuses on finding loop induced buffer overruns. It takes a feasible program path to target instruction. Dowser executes that path and ranks each branch in loops according to the probability of leading to *interesting* pointer manipulation and hence the buffer overrun. Although Dowser's approach is interesting, its branch ranking heuristic needs a feasible path as an input which limits its applicability.

Although both [28] [58] present loop-aware targeted heuristic that focus on branch prioritization in loops, none of them focus on mitigating path explosion, hence, memory explosion for online execution engines caused by loops.

3.4 Use of Regression Test Suite

Search Based Software Testing. Using existing regression test cases to generate new test cases have been explored in *Search Based Software Testing (SBST)*. In *SBST* [51] [89], meta-heuristics based search algorithms (e.g., simulated annealing, tabu search, genetic algorithms) are used to generate new test cases. Branch distance is also used in *SBST* as a metric in search space for estimating distance between two program instructions where the search process tries to find a solution in search space with minimal cost e.g., branch distance.

Extracting Program Structure. Regression test suite is also used extensively in symbolic execution based testing to extract structure and information of the program. One very common use of regression test suite is resolving memory/register-indirect jumps in the program to recover as many edges in the CFG as possible [28] [71] [73].

Seed Input for Targeted Exploration. Katch [71] uses regression test cases to find seed input for future symbolic exploration.

Dowser [71] also uses regression test-suite to rank which conditional branches in loops may lead to "interesting" pointer manipulation/array accesses because Dowser is only focused loop based vulnerabilities. Later Dowser uses this information to guide the targeted search.

ZESTI [70] uses existing regression test suite to find sensitive instructions in the program paths executed by regression test suite. ZESTI later tries to execute the sensitive instructions through another program path. The assumption of ZESTI is that program paths exercised by regression test suite may not reveal bug in sensitive instructions, but another program path may expose the bugs.

Finding Relevant Input Bytes. Regression test suite has also been used to find relevant input bytes to the target [53] [58] [73] [81]. Only relevant input bytes are marked as symbolic in targeted exploration to make it more efficient because less amount of symbolic data in program allows more concrete execution.

3.5 Memory Management

In addition to path explosion, online symbolic execution engines suffer from memory explosion problem. Because online engines forks a new execution state for each feasible symbolic branch target. Usually number of forked execution states grow exponentially with the number of executed symbolic branches. Exponential growth of execution states quickly strains system memory which is known as memory explosion. Offline symbolic execution engines do not suffer from memory explosion problem because they execute only one execution state.

To our best knowledge, only KLEE [39] and Mayhem [42] engines are memory usage aware. KLEE terminates execution states randomly when memory usage reaches the memory limit. Although it helps to mitigate memory explosion and useful in coverage-optimized search strategy, terminating states in random manner in targeted search can be fatal - an execution that is about to execute target instruction can be terminated randomly.

Mayhem introduces the concept of *hybrid symbolic execution* that combines both online and offline symbolic execution in the same program analysis session. Mayhem begins symbolic exploration in multi-path mode. When the memory usage reaches memory limit, Mayhem selects an execution state to checkpoint, i.e., persist the state to disk and continues executing the remaining execution states in memory. When Mayhem finishes all execution states in memory, it selects one of the execution states persisted in disk based on some heuristics and resumes online symbolic execution. In this manner, Mayhem never terminates a state per se, it just suspends execution states to disk and later resumes execution of the suspended states from disk.

3.6 Other

Researchers have also used software model checkers to solve the program statement reachability problem by specifying the target instruction as the target program state in the model. Since program lines can be guarded by conditionals that check arbitrary properties of the current program state, this problem is equivalent to the very general problem of finding a path that causes the program to enter a particular state. Similar to targeted search in symbolic execution, directed model checking [48] focuses on scheduling heuristics to quickly discover the target state. Several heuristics have been based on minimizing the number of transitions from the current program state to the target state in the model defined by a finite-state automata [50] or Buchi automata [49].

Chapter 4

Targeted Search

In this chapter, we will present an overview of targeted search strategy of WatSym, how WatSym mitigates "path explosion", and a new execution state life-cycle scheme to mitigate "memory explosion" problem of online symbolic execution engines.

4.1 Overview

Automatically finding a feasible program path, i.e., an input that will execute a specific program instruction is in general an undecidable problem. Even though the theoretical limitation, it is possible that a well-tuned, domain specific heuristic can synthesize a concrete input that will execute a target instruction with the values needed to reveal a bug in reasonable time.

Symbolic execution suffers from the notorious "path explosion" problem [39]. WatSym incorporates a number of techniques to cope with the large number of execution states (each execution state represent a feasible program path/sub-path) that get forked during symbolic execution. The foremost of these techniques is the use of a distance estimation heuristic to guide symbolic execution to those paths that are most likely to reach the target. WatSym introduces the concept of "target loop", and "target function" in targeted search. Using "target loop", "target function", and context-sensitive CFG, WatSym estimates the effort needed to execute the target from currently executing basic block. Using this estimate, the exploration of paths is steered toward choices that are more likely to execute the target, thus enabling WatSym to find a suitable path considerably faster than mere symbolic execution.

To mitigate "path explosion", WatSym introduces two different heuristics for terminating execution states that are less likely to execute the target. To mitigate "memory explosion" in targeted exploration, WatSym has introduced a new execution state management scheme.

WatSym's targeted search can be divided into three phases:

1. The static binary analysis phase
2. Regression test suite execution phase
3. Targeted symbolic exploration phase

In the first phase, WatSym uses static analysis on program binary to extract program structure e.g., control-flow graph (CFG), functions, and basic blocks in loops. In the second phase, WatSym executes the regression test suite of the target program to find the seed input for targeted exploration. Many online symbolic execution engines are capable of accepting concrete values (in other words, a test-case) for respective symbolic input values; the concrete values are known as seed input. Seed input is important because it determined the starting point in the search space for the search strategy. In targeted exploration phase, WatSym tries to find a feasible program path, i.e., an input that goes through the target instruction using targeted search heuristics.

4.2 Targeted Search

In this section, we describe first how we extract CFG in section 4.2.1, select seed input for symbolic execution in section 4.2.2, and finally the targeted search heuristic in section 4.2.3.

```
1  int main(char** argv, int argc) {
2
3    if (0) {
4      target_function(argc);
5    }
6    else {
7
8    }
9    return 0;
10 }
11
12 void f(int* argc) {
13   for ( int i=0; i<10; i++) {
14     if (*argc < 10) {
15       *argc++;
16     }
17     else {
18
19     }
20   }
21 }
22
23 void target_function(int argc) {
24
25   f(&argc);
26
27   if (argc > 10) {
28     int a[10];
29     for ( int i=argc; i <= (argc+10); i++) {
30
31       if (argc > 15) {
32         a[i-argc] = argc; // target program statement
33       }
34
35     }
36   }
37   else {
38
39   }
40
41 }
```

Listing 4.1: A running example to demonstrate our technique. Line 32 is the target statement because of potential buffer overrun.

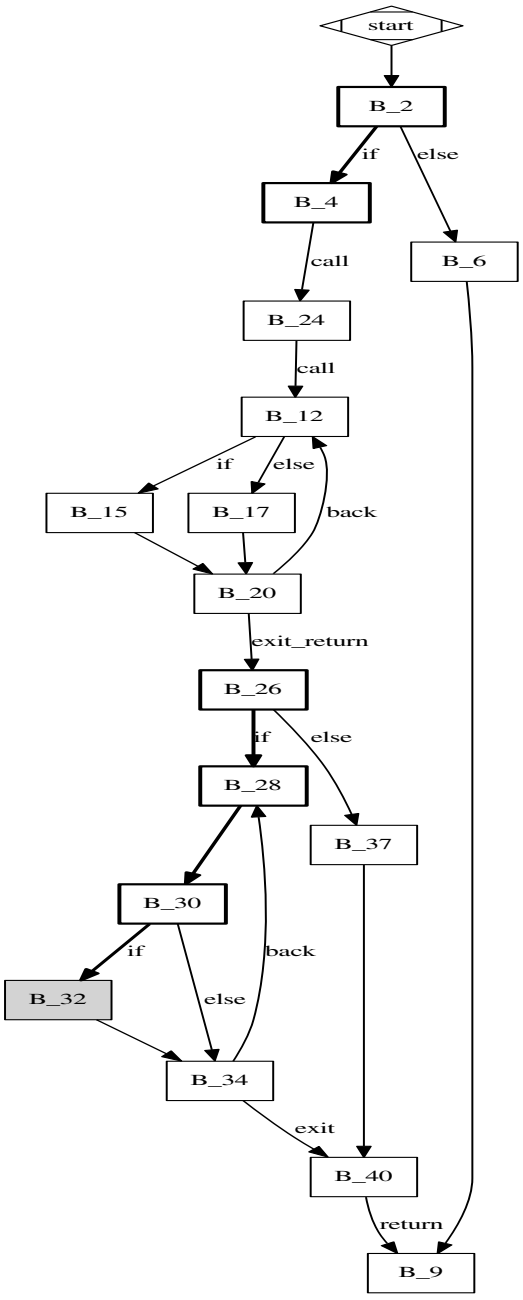

Figure 4.1: Control-Flow Graph of the running example in code 4.1. Each node represents a basic block, each basic block is named that is suffixed with the source line number where the block starts e.g. the target block is `B_32` . Basic blocks with bold borders represent dominator basic blocks of target block. Critical edges are represented by bold edges.

4.2.1 Control-Flow Graph Extraction, Repair, and Distance Estimation

For targeted search, WatSym uses inter-procedural control-flow graph (CFG) for estimating distance. Therefore, it is essential to have complete inter-procedural CFG of the program binary. But extracting CFG of a program by static analysis is undecidable because of register or memory-indirect jumps in program binaries.

Program paths enumerated from context-insensitive CFG can be not well-formed. A well-formed program path has the restriction that the callee function has to return to its caller function. But in context-insensitive CFG a callee function can return to another function that is not necessarily its caller function [28]. Attempting to follow an infeasible path that can be detected infeasible through static analysis is counter-productive.

To circumvent this problem, WatSym extracts 1-context sensitive CFG as opposed to context insensitive CFG. One of the benefits of context sensitive CFG is the paths on context sensitive CFG is well-formed (matches function calls and returns) up-to the given k context sensitivity level. WatSym extracts 1-context sensitive CFG where it is guaranteed that the callee function will return to its caller function. WatSym can also extract CFG with context sensitivity level higher than one, but extracting CFG with higher context sensitivity level is highly expensive and in our experiences we have found 1-context sensitive CFG is to be good enough. Incomplete CFG makes the distance estimation heuristics more imprecise and even can make reaching a target instruction impossible as the search heuristics may keep looking at wrong region of search space.

WatSym also repairs the extracted CFG using dynamic analysis during regression test-suite. During regression test-suite execution and symbolic exploration, WatSym monitors the control-flow edges followed by the program and recovers the missing edges in CFG if execution(s) follows any missing control-flow edge.

Distance Estimation. In targeted search phase, WatSym needs to prioritizely execute the states that are more likely to execute the target instruction. WatSym estimates the effort needed for each execution state to execute the target (if the execution state is able to execute). To estimate the distance between an execution state (currently executing instruction) to target instruction, WatSym needs a distance estimation heuristic that will ideally find the distance of a feasible program path from current instruction to target.

WatSym uses a variation of CFG/branch distance [38] as distance estimation heuristics. The vanilla branch distance is the shortest path on the weighted CFG where conditional branches are assigned weight *one* and unconditional branches are assigned weight *zero*.

Intuitively, the effort for finding a feasible path to target instruction is choosing the right branch.

Impreciseness of the distance estimation heuristics will only make the targeted search slower but will not affect its correctness.

4.2.2 Seed Input Selection

In the second phase, WatSym selects the seed input from existing regression test cases to current target instruction for targeted symbolic exploration. The benefit of using regression test suite is that test cases often exercise interesting program regions e.g., a region which is bug-prone or a region that is semantically important. Bug-prone program regions usually cause more warnings to be generated by static analysis tools. Using regression test case as seed input can help WatSym to find a feasible path closer to the target instruction.

Using seed input has its own benefit. Seed inputs are particularly beneficial for testing programs that process highly structured input (e.g., compilers, program interpreters). Programs that take highly structured inputs impose challenges because of not only the path explosion due to large number of input parsing branches [87], but also executions often get trapped in the input parsing program regions [55] [87]. Therefore, symbolic exploration has to get pass through input parsing regions to execute deep interesting program states. A valid input that can pass through the input parsing regions helps targeted exploration to escape input parsing regions and mitigate path explosion caused by input parsing program regions.

After executing the regression tests, WatSym chooses the nearest regression test case as the seed input for targeted symbolic exploration. The nearest test case is the test case for which one of the forked feasible execution states is the nearest to the target instruction among all the forked execution states by all the test cases. In the next section, we describe WatSym's distance estimation heuristics.

4.2.3 Targeted Search

We define terms target loop and target function as:

Definition 4.2.1. Target Function: The function that contains the target instruction is called target function.

Definition 4.2.2. Target Loop: The loop(s) that contain the target instruction is called target loop(s). If the target instruction is inside nested loops then all the outer loops are regarded as target loops.

WatSym uses online targeted forward symbolic exploration to search for a feasible path that reaches the target. Therefore, at any instance, WatSym usually maintains more than one execution state. To select an execution state to process, WatSym uses a heuristic to estimate how long it would take each execution state to reach the target, and then processes the execution state that would take least amount of effort to reach the target according to the heuristic. To estimate distance, in other words, effort needed to reach the target, of an execution state, WatSym categorizes execution states and deploys different distance estimation heuristics for different categories. The overall estimation of effort needed has to be as precise as possible and should require low overhead computation.

WatSym categorizes execution states into three categories:

1. Execution states that are currently executing inside target loops (if there is any)
2. Execution states that have the target function stack frame in its call stack
3. Execution states that do not have the target function stack frame its call stack

Selecting States That Have Reached The Target Loop(s). If there is any execution state that is executing inside target loop(s), WatSym executes that state until the execution leaves the target loop(s). Loop induced vulnerabilities can only be detected when the loop(s) is executed more than certain number of times. Therefore, WatSym executes that execution state until it reveals the vulnerability or the execution state has left the target loop(s).

In our example, loop in line 29 is our target loop. The vulnerability inside the loop at line 32 can only be revealed if the target loop is executed enough times. WatSym will find this vulnerability because once an execution state reaches the target loop, it will keep executing the state until that state exits the target loop. CFG-directed search [38] may reach the target statement, but without the concept of target loop, it will fail to reveal the actual bug because it will execute the target first time but will not find any vulnerability in that line.

Selecting States That Have Reached The Target Function.

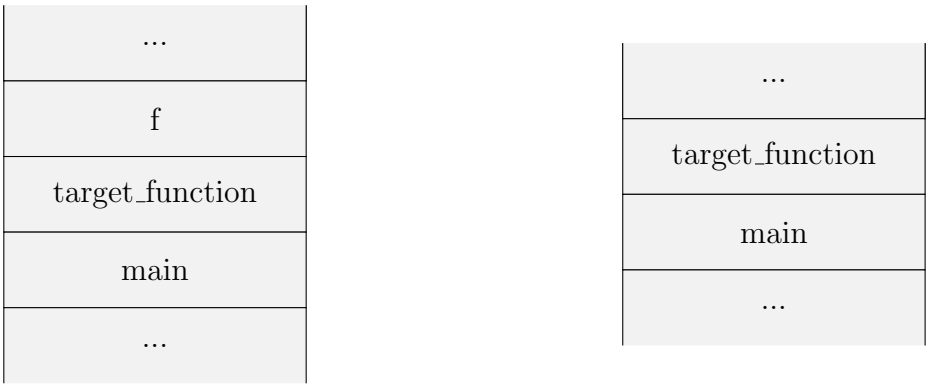

Figure 4.2: Two execution states that have target function in their call stacks. WatSym selects the execution state on right which has target function in the lowest call depth.

In our example, if the execution is currently at line 24, the branches in the loop at line 13, conditional branch at line 14 in function `f` will be included in branch distance computation. But this is imprecise because no matter what branches taken inside `f`, execution will always return back to the target function. Therefore, the execution states that have reached the target function but currently executing in other functions should be prioritized instead of their respective branch distance. Because no matter which branches are taken in functions that are called subsequently from the target function, we will return back to the target function. This is based on the assumption that the called function will always return to its caller function with the exception of program exiting functions like `abort` and asynchronous programs.

If there is no execution state that is executing inside target loop(s), WatSym selects from the states that have target function in call stack if there is any. The idea is that to execute the target the targeted exploration has to reach the function first. Intuitively, we should give higher priority to the states that have already reached the target function than the states that are yet to reach the target function. Therefore, if an execution state has target function in its call stack, then currently executing function may return to its caller and so on to return back to the target function.

If there are more than one such state that have target function in call stack, WatSym selects the state that has the target function in the lowest depth in call stack (shown in fig. 4.2), i.e., the number of functions to return to return back to the target function. The heuristic estimates number of instructions has to be executed before returning to the target function in a very cheap manner. In case of multiple stack frames of target function in call stack, WatSym measures distance by the frame that is found in the lowest depth. In our

example, if we have two execution states that have `target_function` in their call stacks, one is executing inside the target function and another is executing inside the `f`. In such case, WatSym will prioritize the state that is currently executing inside `target_function` because it needs minimum number of returns to back at `target_function` which is zero in this case. More precise but potentially more expensive heuristics can be used to estimate number of instructions [91] for returning to target function.

The idea of prioritizing the execution states that have target function in their call stack is inspired by ESD [91] heuristic. Although WatSym uses a simpler heuristic to choose from the states that have target function in their call stacks. WatSym also introduces the concept of target loop which is very important for finding security-sensitive bugs.

Detecting Execution States That Have Bypassed The Target Basic Block. There is one caveat to selecting from execution states containing target function stack frame in call stack. An execution state may have bypassed the target instruction but the target function can be still in the call stack.

```
1   ...
2   if (argc > 10) {
3     int a[10];
4     for ( int i=argc; i <= (argc+10); i++) {
5
6       if (argc > 15) {
7         a[i-argc] = argc;
8       }
9
10    }
11  }
12  else {
13    // currently executing statement
14  }
15  ...
```

Listing 4.2: The execution is currently inside target function and the execution has bypassed the target function.

For instance, in our running example, the target is inside true branch, but the execution has gone through the false branch, i.e., bypassed the target in CFG by taking an alternative branch (shown in code 4.2). In such case, the targeted search prioritizes a state that has already bypassed the target instruction. To detect such states, WatSym follows similar idea of *chopped CFG*. Chop of intra-procedural CFG contains the sub-graph of CFG where each node can reach (in graph theoretic terms) the target node. To compute chop, an edge is added from target basic block `B_32` to the function entry block `B_26` and strongly connected component (SCC) of target function CFG is computed. The SCC that contains the target block is the chop. If a node that is outside the chop, execution can not reach the target node in current invocation of target function, but execution can reach the target block in future invocations. For example, `B_37` and `B_40` are outside the chop, so if an execution reaches one of these blocks, it will never reach the target in current invocation of the function. If WatSym detects that a state is executing inside such basic block, WatSym declared that state has bypassed the target instruction and suspends that execution state.

Proximity Based Search. If there are no execution states that have target function in call stack, WatSym uses proximity based search. The goal of the proximity based search [38] is using statically extracted program structure to guide the dynamic search to find a feasible path to target. WatSym selects the nearest execution state according to branch distance heuristic. The distance of an execution state is estimated as the shortest distance from execution state's currently executed basic block to the target basic block in the program's inter-procedural CFG. It is a greedy heuristics, always selects the execution state with minimal distance to the target. Intuitively, the effort of targeted exploration lies in an execution state following the "right" branch to target. Therefore, WatSym computes this distance in terms of the number of conditional branch instructions between current block and target block. All the conditional branches are assigned weight one and non-conditional branches (including function calls and returns) are assigned weight zero.

Local Minima in Search Space. One major concern in heuristics driven search algorithms is that the search can be trapped inside a local minima in search space. Local minima is dangerous for search processes that can not backtrack because without backtracking escaping local minima is very hard. Targeted search in offline symbolic execution engine [71] is vulnerable to local minima since the search process can not readily backtrack and the search process has to be restarted.

Targeted search in multi-path symbolic execution are not prone to this phenomenon because the search heuristics can always backtrack and selects another execution state to proceed provided that such an execution state is available. If one of the execution states is trapped in a local minima, WatSym can always backtrack to higher up in the symbolic

execution tree and select another execution state. Hence, the targeted search of WatSym avoids the danger of getting trapped in local minima.

4.2.4 Finding the Least Useful Execution States

WatSym introduces two different heuristics described in section 4.3.1 and section 4.3.2 for terminating states to tackle path and memory explosion. Unfortunately, these heuristics are often not adequate enough to tackle exponential growth of forked execution states which very quickly overwhelms memory. Because when the memory limit is reached, often times terminating suspended states is not enough to limit memory usage. In such case, WatSym has to terminate least useful active execution states at that instance to limit memory usage. Currently WatSym assumes the execution states that are the most distant at that instance as the least useful execution states. WatSym terminate the most distant states to limit memory usage. This is a greedy heuristic and together with targeted search heuristic which itself is another greedy heuristic, the juxtaposition of these two greedy heuristics can result in sub-optimal performance.

4.3 Tackling Path Explosion in Targeted Search

Offline symbolic execution engines usually have very low resource requirement because they only work on single execution state of a program. But they have the cost of executing same instructions multiple times redundantly. To avoid this redundant executions of same instructions, online symbolic execution engine forks a new execution state at each symbolic branch if the both branches are feasible. But forking new execution states also strain the memory quickly, causing the system thrashing because all the execution states are kept in memory. Because of thrashing, the number of explored paths decreases dramatically with the increasing memory usage and in the worst case, online symbolic execution executes slower than its counterpart offline symbolic execution shown in fig. 4.3 [42].

To mitigate memory explosion, state-of-the-art online engines use optimizations like copy-on-write, shares common data between execution states. Nonetheless, due to exponential growth of number of execution states, eventually all online execution engines will reach the memory limit. Memory usage can be reduced by terminating execution states. However, terminating execution states has to be done very carefully. Otherwise a state that executes the target, can be forcefully terminated.

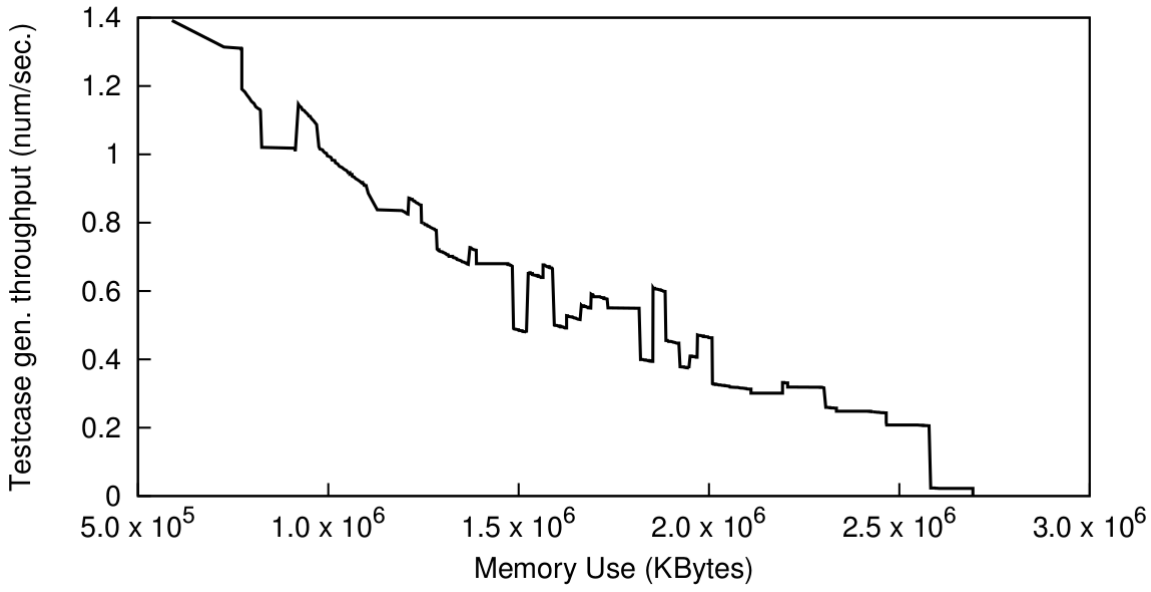

Figure 4.3: Number of path explored vs. memory use [42]

To keep the targeted search efficient, WatSym has to make a trade-off between memory usage and number of execution states at any instance. To keep memory usage in check, WatSym should carefully terminate execution states that are less likely to execute the target instruction. It also has to maintain a healthy population of execution states so that even if the nearest state right now does not lead to a feasible program path to the target instruction, there are other execution states that WatSym can process to find a feasible path.

In the following subsections, first we describe how WatSym tackles path explosion problem caused by loops in section 4.3.1, and then how WatSym finds execution states that are unlikely to execute the target in section 4.3.2, finally we describe WatSym's new execution state management scheme in section 4.4.

4.3.1 Tackling Path Explosion Caused by Loops

Natural loops (e.g., *for*, *while* constructs) are notoriously known for causing path explosion in symbolic exploration. Loops induce path explosion in two ways:

1. Symbolic loop conditions fork new execution state in each iteration of the loop
2. Symbolic branch conditions inside loops fork new execution states (if both branches are feasible) in each iteration of the loop

Loop condition branches have two edges in the CFG: exit edge that exits the loop, and back edge that re-iterates the loop. If an execution state is forked over a loop condition, WatSym suspends the execution state that exits the loop, i.e., WatSym only keeps the execution states that iterate the loops to the fullest. This is very unlike to coverage-optimized search where loops are iterated as few times as possible and execution states that exit the loops early (akin to bounded iteration) are favored over execution states that keep iterating the loops. Although this idea may seem counter-intuitive at first, it is important for finding loop induced vulnerabilities.

Many buffer and integer overflows are directly or indirectly induced by loops, so it is essential to iterate the loops to the fullest to detect loop induced bugs. Also, in our experience, if the loop exiting execution states are favored over loop iterating states, exiting states tend to generate test-cases are mostly rejected by the program as invalid input. One of the reasons behind program rejecting test-cases generated by bounded iteration is that often inputs are provided as an array (e.g., command line parameters are essentially array of arrays of characters) and program needs to process each element of the input array in a loop. Early loop exiting execution states indicate that only some elements of the input array have been processed and other elements are not processed which in turn generates invalid test cases [58].

In our example, each time the target loop is iterated, two different execution states are forked at the loop condition at `B_34`. In such case, WatSym only keeps the state that follows the back edge `B_34` $\rightarrow$ `B_28` because it will re-execute the loop and terminates the states that follow loop exit edge `B_34` $\rightarrow$ `B_40`. This helps WatSym to mitigate path explosion caused by loops and only keep the states that are more likely to reveal the vulnerability.

For execution states that have been forked on same symbolic branches inside loops, WatSym only keeps one execution state and terminates other execution states. At any point of targeted exploration, if WatSym finds more than one execution state such that their last followed conditional edges are same, WatSym only keeps the oldest execution state and terminates other states. WatSym measures age of an execution state by the number of executed program basic blocks. The key observation behind this idea is that two different execution states that have been forked along same conditional branch in loops tend to behave same way in the future e.g., they have same branch distance to target instruction. Therefore, WatSym keeps the execution state that has executed maximum

number of basic blocks because it is more likely to satisfy overflow condition than the other states.

In our example, each time the target loop is iterated, two different execution states are forked at the conditional branch at line 31. The target loop can be executed maximum eleven times. For instance, an execution at iteration 11 of the target loop, is forked at line 31: one at true branch of line 31 named `ExecutionState_31_true_11` and another at false branch at line 31 named `ExecutionState_31_false_11`. Right after forking, WatSym examines the current states at that instance, it has found one more execution state that has been forked at true branch of line 31 named `ExecutionState_31_true_9` in loop iteration number 9 and another one that has been forked at false branch named `ExecutionState_31_false_10` in loop iteration number 10. In such case, WatSym terminates `ExecutionState_31_true_9` and keeps `ExecutionState_31_true_11` because it has executed more basic blocks than `ExecutionState_31_true_9` and more likely to satisfy overflow condition.

We have also experimented with bounded iteration of loops, but it appeared unwieldy because programs parse inputs in loops. Bounded unrolling of input parsing loops often leads the program to reject the input as invalid because some of the bytes of input have not been parsed because of bounded iteration.

4.3.2 Terminating Execution States That Can Not Reach Target

In the worst case, number of program paths that reach a program instruction is unbounded. But many program paths are guaranteed not to reach the target instruction. Exploring these program paths is not only inefficient, but also leads to path explosion because these program paths forks new executes states. To keep the search focused on the target, WatSym has to identify the program paths that do not reach target instruction and terminate execution states that follow those program paths during targeted exploration as early as possible.

Definition 4.3.1. Critical Edge: A *critical edge* is a branch (an edge in CFG) in the program that must be followed to reach a target instruction. Conditional branches have more than one outgoing edge in the CFG for each branch targets. If from input entry points to target instruction, only one of the outgoing edges of a conditional branch can be part of the path to target, then that edge is a critical edge. Any execution state that

does not follow the critical edges can be terminated since the execution will never reach the target instruction. Finding critical edges is undecidable because the soundness of detecting critical edge depends on the completeness of the CFG.

Definition 4.3.2. Dominator Node: A *dominator node* d dominates a node n in a directed graph if every path from entry nodes to n must go through d. A dominator node d is a strict dominator if d is dominator node of n and d, and n are not same node. Similar to *critical edges* finding dominator nodes in CFG is undecidable because the soundness of detecting dominator node depends on the completeness of the CFG.

Dominator basic blocks of target block are important because by definition any program path from any program entry point to target instruction has to go through dominator blocks of target block. Before reaching the target block, an execution has to go through all target block's dominator blocks. Therefore, any control-flow edge that has to be taken to reach the dominator blocks is also a part of any feasible path from program entry point to the target. Edges essential to be followed to reach the dominator blocks of target block are also essential to be followed to reach the target block and hence they are critical edges of target block.

WatSym finds critical edges in manner similar to backward program slicing [84]. Starting from each block of the target block, and its dominator blocks as source block, at each step, the algorithm finds the predecessor block of current block, ensures only one of the edges from the predecessor block can reach the target block and mark that edge as critical edge. As soon as a block with multiple predecessors is found, marking of critical edges stops there for that respective source block and the algorithm proceeds with another basic block as its source.

WatSym improves the critical edge detection technique of ESD by including the dominator basic blocks of the target block as the source of backward slicing [84] and WatSym finds more critical edges than ESD. Unlike ESD, WatSym utilizes *critical edge* concept to prune program path/terminate states that do not follow critical edges and hence do not reach the target block. A more effective but potentially expensive algorithm can find more critical edges.

For example, ESD will only consider target block as the source and it will find `B_28` $\rightarrow$ `B_30` and `B_30` $\rightarrow$ `B_32` critical edges. WatSym finds `B_2`, `B_4`, `B_26`, `B_28`, and `B_30`

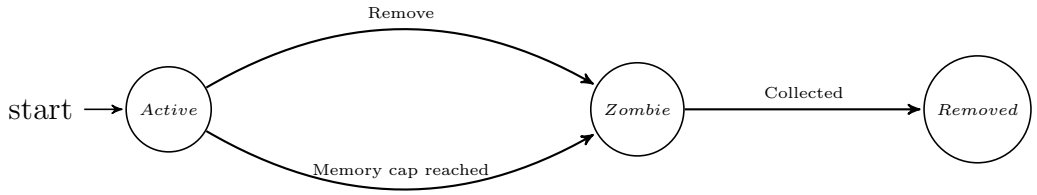

Figure 4.4: State diagram of an execution state in traditional symbolic execution engines like KLEE.

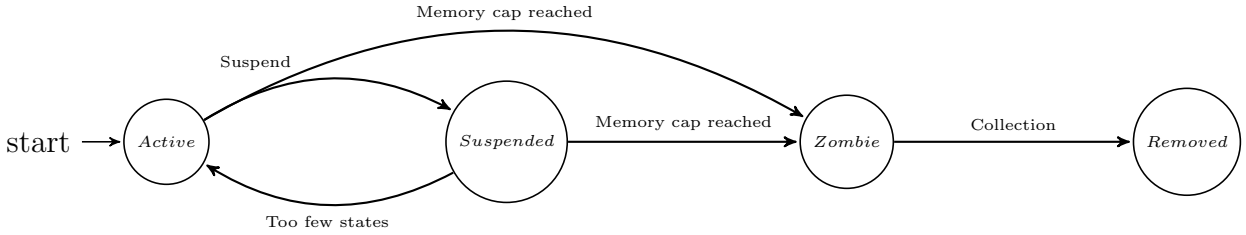

Figure 4.5: State diagram of an execution state in WatSym.

basic blocks that dominator blocks of the target block. WatSym finds critical edges `B_2` → `B_4` and `B_26` → `B_28` in addition to the edges found by ESD.

The soundness of dominator analysis itself depends on completeness of the CFG. Since the extracted CFG is incomplete, critical edge analysis is unsound. Therefore, terminating states that do not follow the critical edges introduces incompleteness (makes reaching target instruction infeasible). Moreover, in presence of cycles in CFG, an execution can execute the target instruction by following a back edge of a cycle and then follow the critical edge to target instruction.

4.4 Lazy Termination of Execution States

Online execution engines (e.g. KLEE [39], DASE [87], and S2E [44]) terminate execution states eagerly, i.e., as soon as an execution state is marked for termination, that state is removed from memory. Terminating states in coverage-optimized search do very negligible

harm to its goal, i.e., improving coverage. But terminating execution states can be fatal in targeted search because an execution state that can execute the target can be forcefully terminated. Forcefully terminating states in targeted search is a necessary evil that must be done when it is absolutely essential, i.e., when memory limit is reached.

To address this issue, we introduce lazy state termination and a new execution state management scheme (depicted in fig. 4.5) in WatSym. As the name suggests, lazy state termination terminates states lazily, i.e., execution states are terminated only when memory usage reaches the limit, but other heuristics are kept under the illusion that the execution state is indeed removed from memory. When an execution state is marked for termination by state termination heuristics, instead of terminating the state right away, WatSym *suspends* that state. A suspended state is removed from active execution states so that the suspended state never gets scheduled by the search heuristics. However, a suspended state is not removed from memory immediately. When memory limit is reached, WatSym heuristically calculates minimum number (size of an execution state can not be precisely measured because execution states share memories in S2E without very expensive bookkeeping) of execution states has to be terminated. WatSym finds least useful suspended execution states at that instance and terminates them. When a state is terminated, underlying engine S2E does not immediately remove that state from memory. Instead, the state is marked as "zombie" and kept in memory for a while to let other parts of the system to clean up resources and wrap up analysis of the zombie state. Eventually the zombie states are removed from memory periodically by S2E engine.

If WatSym reaches memory limit and there is no more suspended state to be terminated, WatSym requests list of execution states that are least useful at that instance from currently deployed search strategy. Upon receiving the list of states, WatSym directly terminates those states directly to keep memory usage in check. When the targeted search strategy is requested for the least useful execution states, the targeted search heuristic finds the least useful states described in section 4.2.4 and returns them to S2E engine to terminate them.

Many times a state termination heuristics is too aggressive that too many execution states are terminated prematurely. In such case, the targeted search is done processing all active execution states without executing the target because too many execution states are forcefully terminated. If WatSym has finished processing all active execution states without executing the target, WatSym resumes the suspended execution states so that the targeted exploration can make progress.

Similar to WatSym, Mayhem [42] also suspends execution states to disk when memory limit is reached and after finishing the active execution states, Mayhem resumes suspended execution states from disk. Similar to Mayhem, WatSym can also use snapshot feature

available in S2E to persist the suspended states to disk and resume later from disk. To our best knowledge there is no existing work in targeted exploration that resumes the suspended states.

All the complexities of managing execution states is kept hidden from the search strategy, the path pruning/state terminating heuristics. The heuristics have access to only the active execution states; therefore, when an execution state is removed from active execution states, to the heuristics, that execution state is "removed" from memory. And when an execution state is resumed from suspended state, the search strategy sees the resumed states as newly added execution states. This enables WatSym's lazy state termination policy can be applied to any search or path pruning strategy without any modification for new state life-cycle.

Chapter 5

Implementation

WatSym is built upon the selective symbolic execution technique implemented in S2E [44]. S2E is based on QEMU [20], an open source hypervisor and KLEE [39], symbolic execution engine for LLVM IR.

The WatSym is consist of around: modified 200 LOC in S2E core engine, modified 300 LOC in core S2E plugins, 9.3 kLOC code for WatSym plugins, 1.3 kLOC python script, 700 LOC pre-processing C++ code, 400 LOC post-processing C++ code.

In addition WatSym provides tools for extracting control-flow graph from binary using open source binary analysis framework *angr* [79], for repairing CFG using existing regression test-suite, and post processing tools for extracting generated test cases generated by symbolic execution including crashing inputs.

5.1 Selective Symbolic Execution (S2E)

To mitigate path explosion caused by uninteresting program regions (e.g., environment), Chupounov et al. has proposed *selective symbolic execution* [44]. Selective symbolic execution is based on the key observation that often only some program paths or regions we are interested to analyze. For example, an user may want to exhaustively explore all paths through a program, but may not want to explore the program paths forked inside the environment (shared libraries and operating system). This means that, selective symbolic

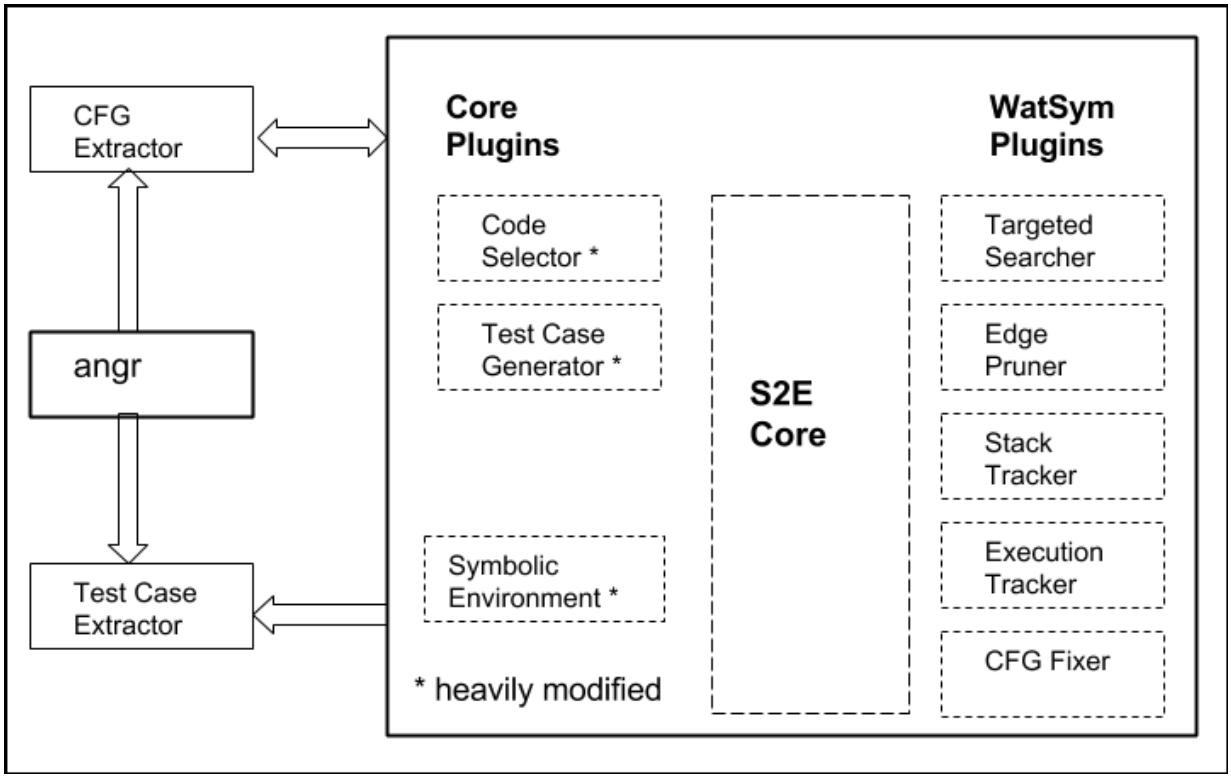

Figure 5.1: Components of WatSym

execution should explore only the program under test in multi-path mode, but whenever the execution leaves the program into some other parts of the system, such as a library, selective symbolic execution should switch from multi-path mode to single-path (shown in fig. 5.2). Selective symbolic execution is named because the online symbolic execution engine selectively execute only some program regions (including the environment itself) in multi-path mode and rest program regions either in single path or concrete mode.

Selective symbolic execution has been implemented in a program analysis platform named S2E. S2E is an hypervisor based on QEMU [20] which uses KLEE [39] for symbolic execution back-end. By combining selective symbolic execution with OS level virtualization, S2E keeps the abstraction of whole system symbolic execution without actually executing whole system symbolically. Also, by allowing concrete execution in the environment, program can be explored without explicitly modeling the environment. Combination of OS level virtualization, executing in real environment, selective symbolic execution gives S2E to analyze real-world system programs e.g., Windows [64], Linux, BSD [74] device drivers.

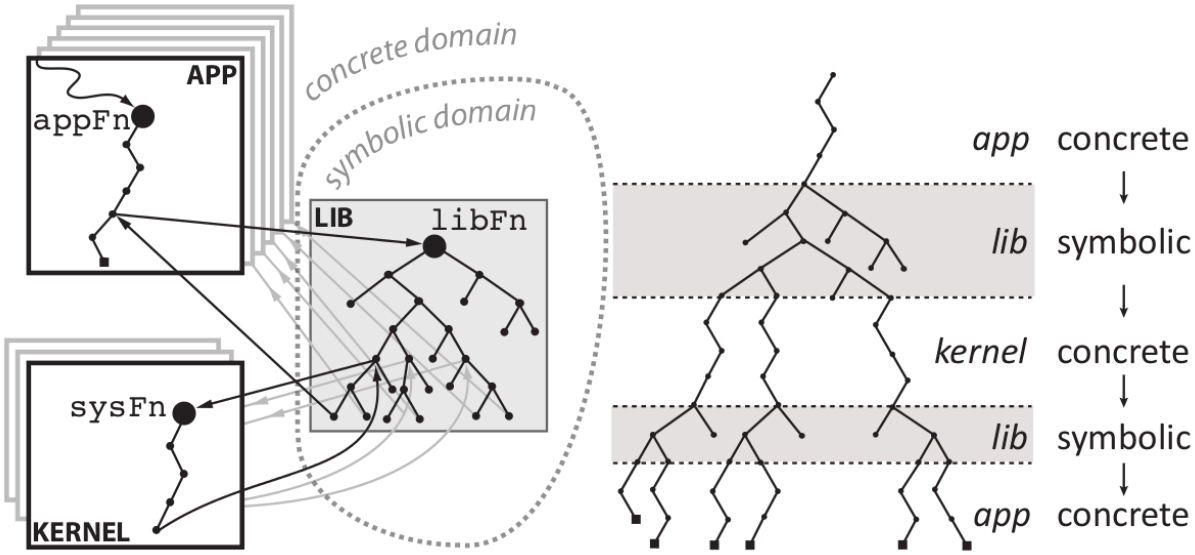

Figure 5.2: Selective Symbolic Execution Multi-path/single-path execution: three different modules (left) and the resulting execution tree (right). Shaded areas represent the multi-path (symbolic) execution domain, while the white areas are single-path [44].

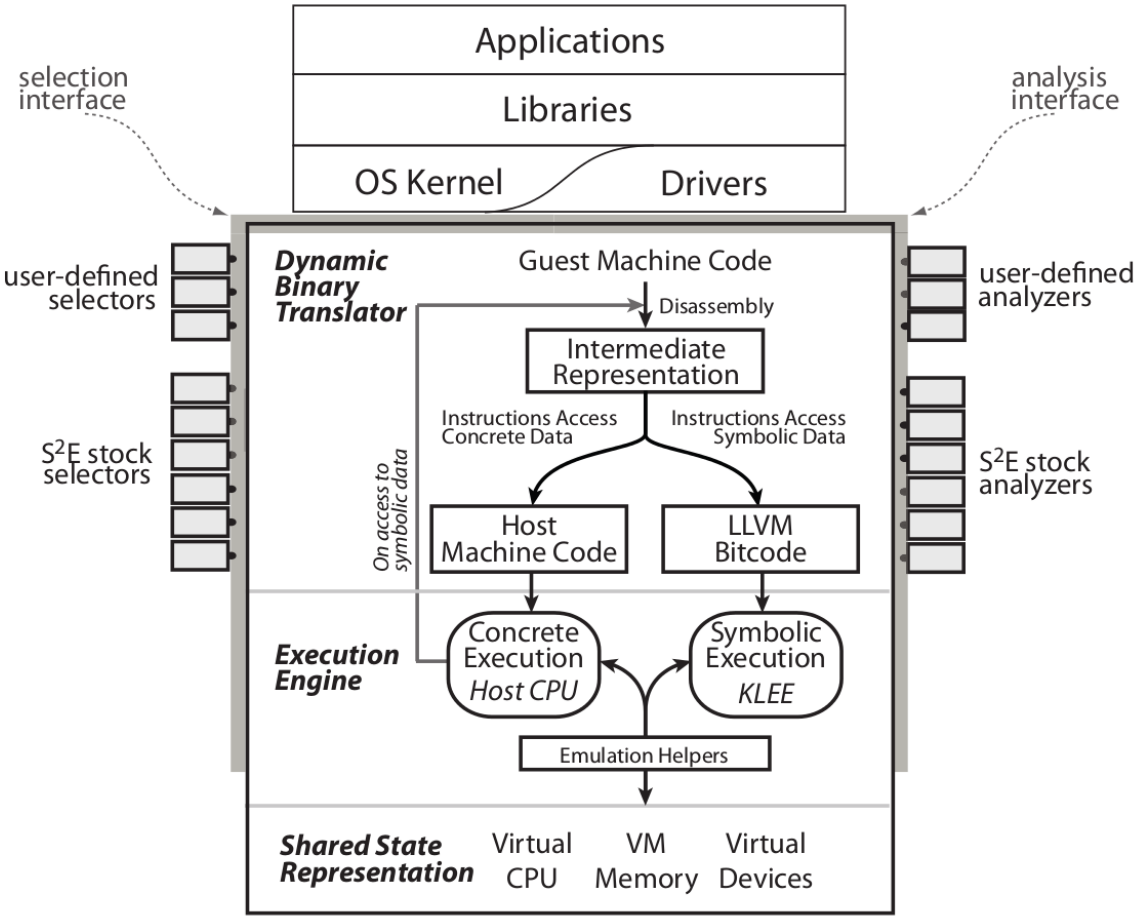

Figure 5.3: Components of S2E [44].

The S2E system diagram is shown in fig. 5.3.

S2E executes only marked program regions (usually the program under test) in multi-path symbolic mode, other program regions and the environment (usually the shared libraries, OS etc.) in single-path symbolic/concrete mode. To maintain the consistency of an execution, when the execution leaves a symbolic program region to a concrete program region, S2E concretizes the symbolic memories accessed in concrete regions by adding *concretization constraints* to the current path constraints. Concretizing symbolic memories enables more concrete execution in future but limits the ability to explore more program paths since an execution can not be forked on a concrete conditional branch.

Even when concrete input for a program path is not available, S2E finds a concrete input by solving current path constraints and uses concrete input to maintain concrete program state in addition to symbolic program state for that path. The benefit is by maintaining concrete state S2E can very efficiently switch to concrete domain from symbolic domain.

5.2 WatSym Implementation

In this section we briefly describe various implementation details of WatSym.

5.2.1 CFG Extraction, Repair and Critical Edge Detection

For extracting CFG from binary, we use binary analysis framework angr [79] which is based on Balakrishnan's work [29] on static analysis of program binary. Recovering CFG through static analysis is undecidable. Kinder et al. [61] explain the "chicken-and-egg" nature of the problem of inferring the control-flow of binaries statically: data-flow analysis is required to infer the control-flow information, and control-flow analysis is required to infer the data-flow information. Although open source binary analysis frameworks [34] [79] can extract CFG from binaries, but in many cases they fail to extract CFG from binaries in reasonable accuracy (too many missing control-flow edges or unable to disassemble basic blocks). Moreover, our evaluation environment is x86 architecture, is a CISC architecture. Distinguishing CISC machine instructions from data through static analysis is an undecidable problem [76].

WatSym also provides a S2E plugin that can repair extracted CFG using existing regression test suite. If any test case exercises any missing control-flow edge in the CFG, WatSym can repair that CFG. WatSym pre-computes the branch distance to the target block for every basic block in the binary.

angr uses NetworkX [19] as graph library. WatSym uses dominator analysis available in NetworkX to detect critical edges, natural loops, chopping. WatSym also provides tools that can detect errors in extracted CFG e.g., overlapping basic blocks, overlapping functions. WatSym also implements a state termination algorithm similar to *chopping* [35].

5.2.2 Extension of S2E Core

We modify S2E to provide an option to users to limit memory usage; we modified S2E core engine to add a new event `onMemoryLimitReached`. S2E engine measures memory usage each time a new execution state is forked and when the user given memory limit is reached, the engine fires `onMemoryLimitReached` event. S2E core engine checks whether any plugin has registered for `onMemoryLimitReached` event. If such a plugin is found, S2E core requests the registered plugin(s) for execution states to terminate. If no such plugin is found, S2E terminates execution states randomly similar to KLEE [39]. Providing interface for specifying which states to be terminated to plugins is important because it is useful to strategies like targeted search where randomly terminating states can be fatal.

By default S2E directly adds concretization constraints to the path constraints. We have modified S2E core to provide API to plugins so that plugins can specify whether concretization constraints should be added to path constraints or ignored. Ignored concretization constraints are stored so that they can be accessed anytime and re-added to path constraints if necessary. We also provide an event which is triggered when a concretization constraint is ignored to notify the plugins.

5.2.3 Plugins

The S2E platform offers a plugin interface for writing custom path analyzers and searchers. S2E plugins can be divided into two key types: the path selection plugins also known as *searcher* plugins, are used to guide the exploration of program paths, and the analysis plugins also known as *analyzers*, are used to collect information (e.g., instruction, branch coverage) on program paths or check properties of program paths (e.g., assertion violations, null pointer dereferences, invalid memory accesses). The benefit of plugin infrastructure is modularity and easier development of plugins. The plugin architecture of S2E is built based on event publish-subscribe pattern. S2E core engine publishes events and other core and custom plugins subscribe to the published events. Plugins themselves can also publish events for other plugins to subscribe. For example, S2E FunctionMonitor plugin publishes events `onFunctionCall`, and `onFunctionReturn` and WatSym's plugin StackTracker subscribe to those events to keep track of target program call stack of each execution state. In addition to modularity, plugins (e.g., FunctionMonitor) can abstract the machine architecture (x86, ARM) specific complexities to other plugins. We develop WatSym using S2E plugin infrastructure to maintain future S2E versions compatibility with exception of modification of core S2E engine and core S2E plugins.

5.2.4 Bug Checkers

To detect occurrence of any overflow in the program, WatSym utilizes run-time error detectors in guest operating system. Run-time error detectors have access to higher level information than symbolic engine does. WatSym can work with any run-time error detector. In this work, we use AddressSanitizer [78], a run-time invalid memory access detector available on both recent versions of GCC and LLVM compiler infrastructure. AddressSanitizer can detect wide range of invalid memory access bugs including use after free (dangling pointer dereference), heap buffer overflow, stack buffer overflow, global buffer overflow, use after return.

We compile programs with `-fsanitize=address` compilation flags to detect invalid memory accesses and `-fsanitize=undefined` to detect integer overflows. Both GCC and Clang supports these compilation flags. We configure the run-time AddressSanitizer library such that if an invalid memory access occurs the program aborts immediately. The shared library that prepares the program for symbolic execution sends this information to S2E engine and subsequently to WatSym plugins.

Chapter 6

Evaluation

In this chapter, we describe the evaluation methodology and result of evaluating WatSym on three real world bugs including a real-world security vulnerability.

6.1 Experimental Setup

Host Machine. We have conducted our experiments on Ubuntu 14.04 64-bit OS on a machine with Intel Core i7-4710HQ @ 2.50GHz CPU and 16 GB RAM.

Guest Machine. S2E and hence WatSym is a hypervisor which allows users to analyze a program or the whole system in real-life environment. Therefore, the programs need to be analyzed in a guest virtual machine run by S2E. The operating system in guest virtual machine is Debian 8.2 [13] 32-bit OS and is allocate 128 MB RAM.

We limit available RAM to 8 GB to both S2E and WatSym using Linux control groups [17]. S2E is built from S2E public Github repository [1] from revision `e4da04762747df14762edbbea36dcabc7a638a64`.

[1] https://github.com/dslab-epfl/s2e

6.2 Evaluated Bugs

We evaluate WatSym on three different invalid memory access bugs from programs: date 8.21, tac 6.10, mkdir 6.10 in Coreutils [4]. For each program we provide a symbolic input specification tailored for each bug, then use WatSym to automatically synthesize inputs that trigger invalid memory access in the programs.

CVE-2014-9471.

```
1  bool
2  parse_datetime (struct timespec *result, char const *p,
3                  struct timespec const *now)
4  {
5      ...
6      for (s = tzbase; *s; s++, tzsize++)
7          ...
8          else if (*s == '"')
9            {
10             ...
11             /* Free tz0, in case this is the 2nd or subsequent time through. */
12             free (tz0);
13             ...
```

Listing 6.1: Double free pointer vulnerability in CVE-2014-9471 [10].

Double free pointer vulnerability occurs in CVE-2014-9471 [10] which affects several programs in Coreutils including date, and touch. The bug occurs when the specified time-zone has more than pair of double quotes (`""`) e.g., `date -d 'TZ="America/Los_Angeles " "00:00 + 1 hour"'`. During parsing, when the second/closing double quote is encountered, memory is freed by `free (tz0)` statement and subsequent double quotes force already freed memory to free again. A remote attacker can exploit this vulnerability to execute arbitrary code by careful crafted the payload.

mkdir 6.10 Crash[2].

```
1  if (scontext && setfscreatecon (scontext) < 0)
2        error (EXIT_FAILURE, errno,
3               _("failed to set default file creation context to %s"),
4              quote (optarg));
5  ...
6  static size_t
7  quotearg_buffer_restyled (char *buffer, size_t buffersize,
8                    char const *arg, size_t argsize,
9                    enum quoting_style quoting_style,
10                   struct quoting_options const *o)
11 {
12       ...
13       for (i = 0; ! (argsize == SIZE_MAX ? arg[i] == '\0' : i == argsize); i++) {
14            ...
```

Listing 6.2: Loop induced invalid memory access bug.

mkdir can create directories in the file-system in different SELinux security contexts. This crash happens when a SELinux security context that does not exist in the system is provided to mkdir e.g., `mkdir -Z non-existent-context directory_name`. When mkdir finds the system does not have any SELinux context with such name, the program prepares an error message by calling `quote (optarg)`; and subsequent invalid memory access occurs in `quotearg buffer restyled` because `optarg` or `arg` in the target function is not set yet.

[2] http://lists.gnu.org/archive/html/bug-coreutils/2008-03/msg00189.html

tac 6.10 Crash[3].

```
1  static bool
2  tac_seekable (int input_fd, const char *file)
3  {
4      ...
5      struct re_registers regs;
6      ...
7      for (;;)
8        {
9          ...
10         if (range == 1
11           || ((ret = re_search (&compiled_separator, G_buffer,
12                                 i, i - 1, range, &regs))
13           == -1))
14         ...
15         else
16           {
17             match_start = G_buffer + regs.start[0];
18             ...
```

Listing 6.3: Loop induced invalid memory access bug.

tac ('cat' spelled backwards) is a program in Coreutils that prints the content of file in reverse order. When provided an input `tac -r file1 file2 ...`, memory access violation occurs in `tac_seekable` function that separates the content of file(s) by user provided separator instead of default separator (newlines). `"-r"` flag forces the program to treat the separator string as RegEx pattern. Memory access violation occurs in `regs` when the pro-

[3] http://lists.gnu.org/archive/html/bug-coreutils/2008-05/msg00018.html

gram tries to separate content by user provided RegEx pattern. The error occurs because the code assumes `regs->num_regs` is initialized when it is not. When tac makes multiple `re_search` calls with the registers on the stack, `re_search` overwrites `regs` in stack with which in turn lead to invalid memory access in subsequent iterations of the target loop.

Program	Target Statement	Error Type	S2E	WatSym
date 8.21	parse-datetime.y:1307	double free pointer	>8 h	5 s
tac 6.10	tac.c:278	invalid memory access	586 s	17 s
mkdir 6.10	quotearg.c:248	invalid memory access	N/A [4]	8 s

Table 6.1: Evaluation: Time taken to reveal bugs by both WatSym and S2E on well-known bugs.

Table 6.1 shows the time needed by WatSym and S2E to reveal the bug. Both CVE-2014-9471, and tac 6.10 bugs are loop induced crashes i.e., the crash happens when the target statement is executed multiple times in loop(s). While other targeted search may be able to find a feasible path to the target, but the memory access violation can only be revealed when the target loop is iterated more than once. WatSym's concept of *target loop* described in section 4.2.3 to find loop induced vulnerabilities, keep iterating the target loop to the fullest even if WatSym has found a feasible path to the target and did not find a memory access violation in the first try. Without the concept of target loop, other targeted search heuristics may have found a feasible path to the target, but would immediately abandon exploring the path further because memory access violation is not found in the target site in the feasible path so far and begin exploring other paths.

6.2.1 Critical Edges

Program	Dominator Blocks	Critical Edges (ESD)	Critical Edges (WatSym)
date	1	1	1
tac	21	1	7
mkdir	19	1	1

Table 6.2: Evaluation: Detected critical edges by WatSym and ESD.

[4] Crashed with assertion violation error after 162 s [42]

In table 6.2, we compare the number of critical edges detected by both ESD and WatSym. By incorporating dominator blocks of target basic blocks WatSym finds more critical edges than ESD. Critical edges help WatSym to prune the search space where finding a feasible program paths to target is very unlikely.

6.3 Other Experiments

We have tried with several known vulnerabilities [6] [7] [8] [9] [12], and check warnings generated by static analysis tool Cppcheck [5], a static C/C++ source code analysis tool; unfortunately as detailed in section 7.1, control-flow graphs we have extracted from respective binaries were not precise enough to conduct evaluation.

Chapter 7

Limitations and Future Work

In this chapter, we summarize both theoretical and practical limitations of WatSym and present several future directions in which WatSym's heuristics including the targeted search heuristic can be improved.

7.1 Limitations

Undecidability of Finding a Feasible Path. Finding a feasible program path that executes a specific instruction is an undecidable problem - solving this problem means solving the halting problem. In programs, unbounded loops make the potential search space infinite. Because of unbounded loops, finding a feasible path is an undecidable problem. Therefore, WatSym may not always be able to reveal a bug.

Theoretical Hardness of Constraint Solving. Symbolic execution has inherent theoretical limitations because of solving complex constraints. Complexity of solving constraints depends on the domain of logic where the constraints are expressed. In bit-vector theory the complexity of solving generated constraints is NP-Complete or harder than NP-Complete. Even though the theoretical hardness of constraint solving recent advancement of constraint solvers has made usage of symbolic execution practically feasible. Regardless of theoretical hardness, modern constraint solvers can solve most of the constraints

generated by symbolic execution in reasonable amount of time [57] [32]. But some constraints generated by symbolic execution can not be solved in reasonable time [32]. In such cases, online symbolic execution engines set a time-out for underlying solver, if the constraints can not be solved before time-out, the execution engines treat the respective path as infeasible and stop processing that execution state. If path constraints generated from program paths that reach the target are hard for underlying solver, WatSym may fail to find a feasible program path.

One such case is path constraints generated by cryptographic hash functions. If there is a target is guarded by `hash_SHA2(m)= 0xf8e56eadb8db9a` condition, it is very likely that WatSym will not be able to find a feasible program path. Because to do so would amount to breaking a cryptographic hash function.

WatSym, based on S2E [44], does not support symbolic evaluation of floating point instructions. Hence, WatSym may fail to find a feasible program path on any program with floating point computation.

Undecidability of Static Disassembly. Although many problems are theoretically hard, practical instances of these problems can be solved in efficient manner e.g., satisfiability, graph isomorphism problem. But the theoretical undecidability of differentiating between data and instructions in CISC instructions through static analysis is found widely in practice. We have experimented with several vulnerabilities [7] [8] [9] [6] [12], where static disassembly did not match the run-time disassembly, i.e., the static disassembly is incorrect. Because of this, we have failed to evaluate WatSym on these vulnerabilities.

Run-time disassembly shown in fig. 7.2, we can see that the crashing instruction starts at address `0x80e658b` [8]. But static disassembly shown in fig. 7.1, we can see there is no instruction that starts at `0x80e658b`. This happens because the run-time disassembly is different from static disassembly of the binary. The static binary analysis framework angr [79] we have used also failed to extract CFG precise enough for targeted search.

Because of undecidability of static disassembly of CISC instructions, we have seen binaries where extracting CFG, even disassembling the target basic block, through static analysis is infeasible. Due to this limitation, we have failed to evaluate WatSym on many binaries.

Requirement of Symbolic Input Specification. Another inherent limitation of targeted search in general applicability is that targeted search needs a symbolic input specification tailored for the target. Symbolic input specification is the number of symbolic input values (because it is also possible to provide concrete input values), types (command line arguments, standard input, file, environment variables, network packets etc.) of symbolic

```
80e65b4:       0f 95 c3                setne  %bl
80e65b7:       89 c1                   mov    %eax,%ecx
80e65b9:       83 e1 07                and    $0x7,%ecx
80e65bc:       83 c1 03                add    $0x3,%ecx
80e65bf:       38 d1                   cmp    %dl,%cl
80e65c1:       0f 9d c2                setge  %dl
80e65c4:       21 da                   and    %ebx,%edx
80e65c6:       84 d2                   test   %dl,%dl
80e65c8:       74 08                   je     80e65d2 <bfd_section_from_shdr+0x3c2c>
80e65ca:       89 04 24                mov    %eax,(%esp)
80e65cd:       e8 3e 2f f6 ff          call   8049510 <__asan_report_load4@plt>
80e65d2:       8b 45 dc                mov    -0x24(%ebp),%eax
80e65d5:       8b 90 a4 00 00 00       mov    0xa4(%eax),%edx
80e65db:       8d 82 8c 00 00 00       lea    0x8c(%edx),%eax
```

Figure 7.1: Static disassembly by objdump tool of crashing program region in strings program for CVE-2014-8485 [8].

```
ASAN:SIGSEGV
=================================================================
==12205== ERROR: AddressSanitizer: SEGV on unknown address 0x41414181 (pc 0x080e658c sp 0xffe140d0 bp 0xffe14198 T0)
AddressSanitizer can not provide additional info.
    #0 0x80e658b in bfd_section_from_shdr /home/riyad/dev-src/test-targets/binutils-2.23/bfd/elf.c:1953
    #1 0x81a51ea in bfd_elf32_object_p /home/riyad/dev-src/test-targets/binutils-2.23/bfd/elfcode.h:808
    #2 0x80668be in bfd_check_format_matches /home/riyad/dev-src/test-targets/binutils-2.23/bfd/format.c:215
    #3 0x80662d1 in bfd_check_format /home/riyad/dev-src/test-targets/binutils-2.23/bfd/format.c:95
    #4 0x804a736 in strings_object_file /home/riyad/dev-src/test-targets/binutils-2.23/binutils/strings.c:376
    #5 0x804a935 in strings_file /home/riyad/dev-src/test-targets/binutils-2.23/binutils/strings.c:419
    #6 0x804a384 in main /home/riyad/dev-src/test-targets/binutils-2.23/binutils/strings.c:286
    #7 0xf5fd4a82 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x19a82)
    #8 0x8049bc0 in _start (/home/riyad/dev-src/test-targets/binutils-2.23/binutils/strings+0x8049bc0)
SUMMARY: AddressSanitizer: SEGV /home/riyad/dev-src/test-targets/binutils-2.23/bfd/elf.c:1953 bfd_section_from_shdr
==12205== ABORTING
```

Figure 7.2: Run-time disassembly by objdump tool of crashing program region in strings program for CVE-2014-8485 [8].

input values. In other words, symbolic input specification consists of how the symbolic input value is given (command line arguments, stdin etc.) to the program and the size of each symbolic input. Coverage-optimized search strategies use a general symbolic input specification to evaluate general applicability of coverage-optimized search for programs. But this general applicability criteria does not apply to targeted search, because targeted search needs a symbolic input specification tailored for the target. Automatically inferring symbolic input specification itself is a hard problem. It is possible to devise a technique that can infer specification for domain specific programs, but this problem still not addressed

yet in existing research.

```
1 int *ptr = malloc(10*sizeof(int));
2 if (ptr == NULL) {
3     target program statement
4 }
```

Listing 7.1: Target program statement (line 3) can only be executed if the environment fails to allocate memory in the run-time.

MergePoint [26] tested 33,248 binaries available in Debian package repositories. A single input specification has been used for MergePoint experiments: 3 symbolic arguments up to 10, 2, and 2 bytes respectively, and symbolic files/stdin up to 24 bytes. MergePoint encountered at least one symbolic branch in 23,731 binaries, i.e., MergePoint did not encounter any symbolic branch for the rest of the binaries. This demonstrates the inherent limitation applicability of general symbolic input specification in targeted search.

Some target statements can only be executed under very special scenarios. For instance, some program regions can only be executed when there is an error in the environment (e.g., the environment fails to allocate memory). For these cases, fault has to be injected [14] in environment to deterministically find a path to the target. In code 7.1, the target program statement can only be executed if the environment fails to allocate memory, in such case not only the environment `malloc` has to be modeled, but the model also has to have fault injection capability. But in other cases where fault injection is not needed to find an execution, fault injection will lead to path explosion.

Symbolic input specification in targeted search determines whether the bug can be revealed at all or not. Wrong symbolic input specification makes the targeted exploration unsuccessful irrelevant of the targeted search heuristic. For instance, CVE-2014-9471 [10] exploit can only be revealed if there are at least two symbolic input values with size of at least 2 and 5 bytes respectively are provided e.g. `date -d 'TZ="America/Los_Angeles " "00:00 + 1 hour"'`. The vulnerable statement can only be executed if `-d` flag and `TZ=.....` data are provided to the program. Therefore, if only one symbolic input is provided, the targeted search can only find either the flag or the data, but not both. Any symbolic input specification with less then two symbolic input values and less than their respective size will make the targeted search unsuccessful. Because the target program

statement can only be executed if a command line flag is provided with the input data. Like any other targeted search tool, WatSym needs a symbolic input specification that will not make targeted search unsuccessful.

Finding Relevant Input Bytes to the Target Instruction. If a program receives input from input file/network packet, usually not all bytes are relevant to finding a feasible program path to the target. Marking all bytes of an input file/network packet symbolic causes path explosion and often prohibitively expensive because usually large amount of input bytes are provided via file/network packet. Therefore, marking only the input bytes relevant to the target symbolic is very important for efficient symbolic execution. Many targeted search tools [73] [58] need a test-case that executes target instruction to find relevant input bytes to the target using dynamic taint analysis [53].

Finding input bytes relevant to the target without using dynamic taint analysis has not been addressed in research yet. Though, it is possible to find relevant input bytes using static backward taint/data-flow analysis [54] on the program binary, but static backward data-flow analysis on binary is a very hard problem. We have tried existing binary analysis frameworks angr [34], BAP [79], but none of them yield backward data-flow analysis precise enough to use in WatSym. Therefore, WatSym will face path explosion for programs that take large amount of input bytes. It is possible to use dynamic taint analysis with existing target instruction executing input to find relevant input bytes and WatSym provides an option to mark only the relevant input bytes symbolic. But requirement of a target instruction exercising test-case limits the general applicability of WatSym.

7.2 Future Work

Execution State Merging in Targeted Symbolic Execution. One related technique to mitigate path explosion is known as *state merging* [26] [31] [37] [65]. State merging is based on the observation that many program paths only differ in few branches taken so far and will explore the same instructions in future. Instead of exploring similar execution states separately, it is often beneficial to merge and explore these execution states simultaneously. State merging introduces disjunctions in path constraints which makes queries harder to solve for constraint solvers. In state merging, the execution engine has to make the trade-off between exploring more simple program paths and exploring fewer complex program paths. Usually states are merged based on some merging conditions when more than one execution states reach same program point following different program paths.

State merging technique has been used for improving code coverage [26] [65]. In the future, we want to investigate state merging for targeted search by merging two almost identical execution states where both are believed to nearest to the target instruction.

String Constraint Solver. String functions (e.g., `strlen`, `strcpy`, `strcat`) are notoriously known for worsening path explosion. String functions are very important because large number of vulnerabilities involve string data. Currently S2E [44], KLEE [39] and hence WatSym models strings as arrays of fixed length bit-vectors. New generation of constraint solvers S3 [85], Z3-str [93] treat string as native primitive and can solve string constraints. By modeling strings in program as native primitive and modeling string functions by using function summaries using string constraints for string solvers can mitigate path explosion problem caused by string functions.

Chapter 8

Conclusion

Symbolic execution is an automated test generation technique that stands out for its soundness, and flexibility of path-based partial verification. This thesis explored finding security-critical bugs such as control-flow hijacks using targeted symbolic execution.

In this thesis, we present a targeted search strategy to check warnings generated by static program analysis tools using symbolic exploration. To detect the execution states that have bypassed the target instruction, we propose a technique to find the basic blocks that an execution state will reach if it bypasses the target based on reachability analysis on intra-procedural control-flow graph of the target function. To mitigate path explosion and keep the search focused on the target instruction, we propose two different execution state termination heuristics. In addition to path explosion, memory explosion is another problem for online symbolic execution engines; we propose a new execution state life-cycle scheme for targeted search to maintain a balanced trade-off between memory usage and healthy population of execution states.

We have built WatSym based on S2E [44] which gives us the ability to analyze large variants of programs including system programs (e.g., Linux kernel) in real environment. We have evaluated WatSym to show that WatSym can find real bugs in program despite all practical and theoretical limitation described in section 7.1.

Although we have implemented and evaluated WatSym as a static analysis generated warning checking tool, WatSym can be easily adapted into other contexts. It can be used as patch testing technique integrated into code review process where WatSym can drive the execution to the program regions modified by the patch. It can also be used as crash

reproducing technique; WatSym can find the crashing instruction from core-dump of the crash and use the targeted search to find a path that will reproduce the crash. WatSym can also be used as triaging tool for warnings generated by static analysis. If the targeted search strategy of WatSym can not find a feasible program path to the target, that means the vulnerability will be hard to exploit by an attacker.

APPENDICES

References

[1] Ariane. the ariane catastrophe. http://www.around.com/ariane.html. [Online; accessed 07-March-2016].

[2] Binutils. https://www.gnu.org/software/binutils/. [Online; accessed 07-March-2016].

[3] Cnn. toyota recall costs: $2 billion. http://money.cnn.com/2010/02/04/news/companies/toyota_earnings.cnnw/index.htm. [Online; accessed 07-March-2016].

[4] Coreutils. http://www.gnu.org/software/coreutils/coreutils.html. [Online; accessed 07-March-2016].

[5] Cppcheck. https://github.com/danmar/cppcheck. [Online; accessed 07-March-2016].

[6] Cve-2013-2174. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2174. [Online; accessed 07-March-2016].

[7] Cve-2014-8484. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8484. [Online; accessed 07-March-2016].

[8] Cve-2014-8485. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8485. [Online; accessed 07-March-2016].

[9] Cve-2014-8738. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8738. [Online; accessed 07-March-2016].

[10] Cve-2014-9471. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9471. [Online; accessed 07-March-2016].

[11] Cve-2015-3144. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3144. [Online; accessed 07-March-2016].

[12] Cve-2015-7696. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7696. [Online; accessed 07-March-2016].

[13] Debian. http://debian.org. [Online; accessed 07-March-2016].

[14] Fault injection. https://en.wikipedia.org/wiki/Fault_injection. [Online; accessed 07-March-2016].

[15] Fuzzing. http://www.fuzzing.org/. [Online; accessed 07-March-2016].

[16] Institute systems sciences ibm. it is 100 times more expensive to fix security bug at production than design. https://www.owasp.org/images/f/f2/Education_Module_Embed_within_SDLC.ppt. [Online; accessed 07-March-2016].

[17] Linux cgroups. http://debian.org. [Online; accessed 07-March-2016].

[18] mkdir crash. http://lists.gnu.org/archive/html/bug-coreutils/2008-03/msg00189.html. [Online; accessed 07-March-2016].

[19] Networkx. https://networkx.github.io/. [Online; accessed 07-March-2016].

[20] Qemu. http://wiki.qemu.org. [Online; accessed 07-March-2016].

[21] tac crash. http://lists.gnu.org/archive/html/bug-coreutils/2008-05/msg00018.html. [Online; accessed 07-March-2016].

[22] Test-suite results for the chromium browser. https://test-results.appspot.com/testfile?testtype=layout-tests. [Online; accessed 07-March-2016].

[23] Saswat Anand, CorinaS. Păsăreanu, and Willem Visser. Jpf–se: A symbolic execution extension to java pathfinder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer Berlin Heidelberg, 2007.

[24] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 265–274, New York, NY, USA, 2010. ACM.

[25] Thanassis Avgerinos. *Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs*. PhD thesis, Carnegie Mellon University, 2014.

[26] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[27] Domagoj Babic and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 211–220, New York, NY, USA, 2008. ACM.

[28] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22, New York, NY, USA, 2011. ACM.

[29] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23, 2010.

[30] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[31] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[32] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[33] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, apr 1975.

[34] David Brumley, Ivan Jager, Thanassis Avgerinos, and EdwardJ. Schwartz. Bap: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 463–469. Springer Berlin Heidelberg, 2011.

[35] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 311–325, Washington, DC, USA, 2007. IEEE Computer Society.

[36] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, pages 183–198. ACM, 2011.

[37] Suhabe Bugrara and Dawson R. Engler. Redundant state detection for dynamic symbolic execution. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 199–211, 2013.

[38] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.

[40] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[41] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[42] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[43] Vitaly Chipounov. *S2E: A Platform for In-vivo Multi-path Analysis of Software Systems*. PhD thesis, École Polytechnique Federale De Lausanne, 2014.

[44] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[45] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[46] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[47] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[48] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International journal on software tools for technology transfer*, 5(2-3):247–267, 2004.

[49] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Int. J. Softw. Tools Technol. Transf.*, 5(2):247–267, mar 2004.

[50] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Trail-directed model checking. *Electronic Notes in Theoretical Computer Science*, 55(3):343 – 356, 2001. Workshop on Software Model Checking (in connection with {CAV} '01).

[51] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 121–130. IEEE, 2012.

[52] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[54] Iván García-Ferreira, Carlos Laorden, Igor Santos, and Pablo García Bringas. *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14: Bilbao, Spain, June 25th-27th, 2014, Proceedings*, chapter A Survey on Static Analysis and Model Checking, pages 443–452. Springer International Publishing, Cham, 2014.

[55] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[56] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[57] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, jan 2012.

[58] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, Washington, D.C., 2013. USENIX.

[59] W.E. Howden. Methodology for the generation of program test data. *Computers, IEEE Transactions on*, C-24(5):554–560, May 1975.

[60] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[61] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In NeilD. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer Berlin Heidelberg, 2009.

[62] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[64] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[65] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[66] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security Privacy, IEEE*, 9(3):49–51, May 2011.

[67] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *Static Analysis*, pages 95–111. Springer, 2011.

[69] Paul Dan Marinescu. *Transparently Improving Regression Testing Using Symbolic Execution*. PhD thesis, Imperial College of Science, Technology and Medicine, London, UK, 2013.

[70] Paul Dan Marinescu and Cristian Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 716–726, Piscataway, NJ, USA, 2012. IEEE Press.

[71] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[72] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[73] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 891–901, Piscataway, NJ, USA, 2015. IEEE Press.

[74] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: Testing drivers without devices. In *OSDI*, volume 1, pages 4–6, 2012.

[75] Caitlin Sadowski and Jaeheon Yi. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 43–51, New York, NY, USA, 2014. ACM.

[76] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 45–, Washington, DC, USA, 2002. IEEE Computer Society.

[77] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, sep 2005.

[78] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX.

[79] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[80] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 43–54. ACM, 2015.

[81] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 473–486, New York, NY, USA, 2015. ACM.

[82] Rahul Telang and Sunil Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Trans. Softw. Eng.*, 33(8):544–557, aug 2007.

[83] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[84] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[85] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA, 2014. ACM.

[86] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with kint. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 163–177, Berkeley, CA, USA, 2012. USENIX Association.

[87] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 620–631, Piscataway, NJ, USA, 2015. IEEE Press.

[88] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, November 2013.

[89] Shin Yoo and Mark Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3):171–201, 2012.

[90] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. [Online; accessed 07-March-2016].

[91] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 321–334, New York, NY, USA, 2010. ACM.

[92] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. Automated debugging for arbitrarily long executions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 20–20, Berkeley, CA, USA, 2013. USENIX Association.

[93] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. Computer aided verification: 27th international conference, cav 2015, san francisco, ca, usa, july 18-24, 2015, proceedings, part i. pages 235–254, Cham, 2015. Springer International Publishing.