Dynamic Test Generation for Large Binary Programs

by

David Alexander Molnar

A.B. (Harvard College) 2003
M.S. (University of California, Berkeley) 2006

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Fall 2009

Dynamic Test Generation for Large Binary Programs

Abstract

Dynamic Test Generation for Large Binary Programs

by

David Alexander Molnar

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Wagner, Chair

This thesis develops new methods for addressing the problem of software security bugs, shows that these methods scale to large commodity software, and lays the foundation for a service that makes automatic, effective security testing available at a modest cost per bug found. We make the following contributions:

- We introduce a new search algorithm for systematic test generation that is optimized for large applications with large input files and exhibiting long execution traces where the search is bound to be incomplete (Chapter 2);
- We introduce optimizations for checking multiple properties of a program simultaneously using dynamic test generation, and we formalize the notion of *active property checking* (Chapter 3);
- We describe the implementation of tools that implement dynamic test generation of large binary programs for Windows and for Linux: SAGE and SmartFuzz. We

explain the engineering choices behind their symbolic execution algorithm and the key optimization techniques enabling both tools to scale to program traces with hundreds of millions of instructions (Chapters 2 and 4);

- We develop methods for coordinating large scale experiments with fuzz testing techniques, including methods to address the defect triage problem (Chapter 4);

Professor David A. Wagner
Dissertation Committee Chair

To my family,

who have always been there,

no matter where we may go.

Contents

List of Figures

List of Tables

Acknowledgments

I could not have done this work without my advisors and mentors David Wagner, Patrice Godefroid, and Michael Levin. I could also not have done this without my PhD committee members Sanjit Seshia and David Aldous. Thank you, all of you.

I am indebted to Chris Marsh and Dennis Jeffries for important contributions to SAGE, and to Hunter Hudson for championing the SAGE project from the very beginning. SAGE builds on the work of the TruScan team, including Andrew Edwards and Jordan Tigani, and the Disolver team, including Youssf Hamadi and Lucas Bordeaux, for which I am grateful. I thank Tom Ball, Manuvir Das and Jim Larus for their support and feedback on this project. Various internal test teams provided valuable feedback during the development of SAGE, including some of the bugs described here, for which I thank them. I thank Derrick Coetzee, Ben Livshits and David Wagner for their comments on drafts of our paper, and Nikolaj Bjorner and Leonardo de Moura for discussions on constraint solving and using the Z3 solver. I thank Chris Walker for helpful discussions regarding security.

I thank Sanjit Seshia, Doug Tygar, and David Aldous for feedback and advice during this project, and for serving on my qualifying exam committee. I am indebted to Vijay Ganesh and David Dill for the use of the STP tool, and for numerous improvements to STP that dramatically increased performance on our test cases. I thank Jason Waddle, Dean McNamee, Ben Laurie, and Daniel Wilkerson for helpful discussions. I thank posters to the `fuzzing@whitestar.linuxbox.org` mailing list, including Jared DeMott, Disco Jonny, and Ari Takanen, for discussions on fuzzing tradeoffs. Martin Vuganoux sent me pointers to his `autodafe` tool and we had helpful discussions.

For the work in Chapter4, I thank Cristian Cadar, Daniel Dunbar, Dawson Engler, Patrice Godefoid, Michael Levin, and Paul Twohey for discussions about their respective systems and dynamic test generation. I thank Paul Twohey for helpful tips on the engineering of test machines. I thank Chris Karlof for reading a draft of our paper on short notice. I thank the SUPERB TRUST 2008 team for their work with Metafuzz and SmartFuzz during Summer 2008. I thank Li-Wen Hsu and Alex Fabrikant for their help with the metafuzz.com web site, and Sushant Shankar, Shiuan-Tzuo Shen, and Mark Winterrowd for their comments. I also thank Erinn Clark, Charlie Miller, Prateek Saxena, Dawn Song, the Berkeley BitBlaze group, and the anonymous referees of Oakland and Usenix Security for feedback on earlier drafts of this work. Chris Kugel served as shepherd for the conference version of this work, for which I am thankful. Xue Cong Li was a joy to work with as an undergraduate research partner and did early work on the infrastructure to power our Amazon EC2 experiments.

In addition, I am pleased to have worked with several great colleagues on projects related to the work in this thesis or which concern the SmartFuzz system but are only tangentially mentioned here. My thanks to Vijay Ganesh, Susmit Jha, Rhishi Limaye, Sushant Shankar, Shiuan-Tzuo Shen, Sanjit Seshia, and Mark Winterrowd.

I have been lucky to receive funding through several different sources over the course of this work. I am thankful to the NSF Graduate Fellowship program, the NSF TRUST center, Microsoft Research, and DARPA for their generous research support.

Without the support of my friends and colleagues, I could not have completed this dissertation. A big thank you to Alex Fabrikant, Nicole Holland, Alex Simma, Jill Blaxton, Karl Chen, Jimmy Su, Jason Waddle, Christine Robson, the Cult of Levitating Plywood, Noisebridge, and Bab5. Thanks to Arvind Narayanan and Adam Bossy for always being there to talk about commercial ideas. My special thanks to Tracy, my lovely girlfriend, who puts up with my odd schedules, and my inexplicable inability to write at any other time than two in the morning. My apologies to the many other people I have likely have forgotten as an oversight.

Finally, I am indebted to my family. Mom, Dad, Susan, I couldn't have made it here without you.

Chapter 1

Introduction

This thesis develops new methods for addressing the problem of software security bugs, shows that these methods scale to large commodity software, and it lays the foundation for a service that helps developers protect their users from the dangers of insecure software. Security bugs are examples of *vulnerabilities* in computer software. A vulnerability is a bug that could potentially be used by an adversary to violate the security policy of a victim's computer system. An *exploit*, in contrast, refers to the actual code that uses the vulnerability to violate the security policy. For example, the vulnerability might be a buffer overflow bug in a piece of code that displays image files. An exploit would be a carefully crafted image file that uses the buffer overflow bug to run arbitrary code on the computer processing the image.

Vulnerabilities exist in all types of software, both "traditional" desktop and server software and in software used to develop Web applications. While vulnerabilities in Web software have grown to become the third most commonly reported vulnerability by vendors in 2005 and 2006 [23], the work of this thesis will focus on vulnerabilities in *file-reading, client-side* software. This category of software includes such software as image processors, decompression libraries, office document readers, and music players.

We focus on such software because the Internet users often view images, play music, or download files from arbitrary sites. For example, vulnerabilities in a web browser plugin, such as Flash, may allow an adversary to take control of a machine should the user surf to the wrong web site. Nevis Labs notes that recent years have seen fewer public disclosures of "remote" vulnerabilities, i.e. those that could be used without needing user interaction. Because such vulnerabilities appear to be harder to find, Nevis suggests that work will shift to finding bugs in client-side code [58].

Furthermore, despite the availability of memory-safe languages, buffer overflows continue to be prominent. MITRE reports that buffer overflows were the number one issue reported by operating system vendors in 2006 [23]. Symantec's 2008 threat report states that memory corruption vulnerabilities accounted for the majority of vulnerabilities in browser plug-ins, with 271 vulnerabilities [24]. MITRE also notes that *integer overflows* jumped from barely in the top 10 issues overall to number 2 for OS vendor advisories in 2006-2007. Later in this thesis, we will develop techniques that are particularly suited for finding integer overflow bugs.

Regardless of the type of software, managing software security bugs requires multiple different approaches. We can organize these approaches using a *bug cycle* as a tool to help understand the relationship of different techniques. In this cycle, we start when when a programmer *writes a bug* into a piece of software. The next stage comes when someone *finds* *the bug*, either through deliberate testing or by accident. After that, we *report the bug* to the developers, who must then *fix the bug* by modifying the software. Future modifications can then introduce new bugs, thereby starting the cycle anew.

Each stage of the bug cycle offers us an opportunity to reduce the severity of software security bugs. For example, there has been intense research into building programs that are correct by construction. This work comes into play at the *write bug* part of the cycle. If we can use these methods, we should, because they allow us to eliminate security bugs before they have an opportunity to affect software users.

Unfortunately, creating software that is correct by construction is not always practical. For example, we may need to work with legacy software or with software libraries not under our control. Therefore, the work of this thesis focuses on the *find* and on the *report* stages of the bug cycle. By reducing the cost for software developers to find security bugs, we make it easier to catch bugs before deployment. In particular, we develop a new method for finding bugs in file-reading client side software that automatically discovers bugs missed by human code review, static analysis, and previous dynamic testing methods.

1.1 Whitebox Fuzz Testing

The starting point for our new method of finding bugs is *fuzz testing*. Since the "Month of Browser Bugs" released a new bug each day of July 2006 [71], fuzz testing has leapt to prominence as a quick and cost-effective method for finding serious security defects in large applications. Fuzz testing is a form of *blackbox random* testing which randomly mutates well-formed inputs and tests the program on the resulting data [35, 79, 1, 8].

In some cases, *grammars* are used to generate the well-formed inputs, which also allows encoding application-specific knowledge and test heuristics.

Although fuzz testing can be remarkably effective, the limitations of blackbox testing approaches are well-known. For instance, the `then` branch of the conditional statement "`if (x==10) then`" has only one in 2^{32} chances of being exercised if `x` is a randomly chosen 32-bit input value. This intuitively explains why random testing usually provides low code coverage [77]. In the security context, these limitations mean that potentially serious security bugs, such as buffer overflows, may be missed because the code that contains the bug is not even exercised.

We propose a conceptually simple but different approach of *whitebox fuzz testing.* This work is inspired by recent advances in systematic dynamic test generation [40, 17]. Starting with a fixed input, our algorithm symbolically executes the program, gathering input constraints from conditional statements encountered along the way. The collected constraints are then systematically negated and solved with a constraint solver, yielding new inputs that exercise different execution paths in the program.

For example, symbolic execution of the above fragment on the input $\texttt{x} = \texttt{0}$ generates the constraint $\texttt{x} \neq \texttt{10}$. Once this constraint is negated and solved, it yields $\texttt{x} = \texttt{10}$, which gives us a new input that causes the program to follow the `then` branch of the given conditional statement. This allows us to exercise and test additional code for security bugs, even without specific knowledge of the input format. Furthermore, this approach automatically discovers and tests "corner cases" where programmers may fail to properly allocate memory or manipulate buffers, leading to security vulnerabilities.

In theory, systematic dynamic test generation can lead to full program path coverage, giving us program verification [40]. In practice, however, the search is incomplete. The incompleteness stems from the fact that the number of execution paths in the program under test is huge, too large to exhaustively search in a reasonable amount of time with the techniques of this thesis. Furthermore, symbolic execution, constraint generation, and constraint solving are necessarily imprecise.

Therefore, we are forced to explore practical tradeoffs. We propose a novel search algorithm, the *generational search*, with a coverage-maximizing heuristic designed to find defects as fast as possible. The generational search responds to an engineering reality we have discovered with large applications, which is that *generating symbolic traces is expensive, but solving constraints is cheap.* Therefore, we want to leverage the expensive work of tracing an application to obtain a path condition over many new test cases. Each test case is a "bite at the apple" that could find a new high value software security bug. In Chapter 2 we describe the generational search, and we show that it is highly effective at finding bugs in commodity software.

1.2 Active Property Checking

The basic dynamic test generation approach creates new test cases which cover different paths through the program. Traditional runtime checking tools like Purify [50], Valgrind [74] and AppVerifier [22], however, check a single program execution against a large set of properties, such as the absence of buffer overflows, use of uninitialized variables or memory leaks. We want a way to generate test cases which violate these properties and exhibit a bug, if any such test cases exist.

Our insight is that these traditional *passive* runtime property checkers can be extended to *actively* search for property violations. Consider the simple program:

```
int divide(int n,int d) { // n and d are inputs
  return (n/d); // division-by-zero error if d==0
}
```

The program `divide` takes two integers `n` and `d` as inputs and computes their division. If the denominator `d` is zero, an error occurs. To catch this error, a traditional runtime checker for division-by-zero would simply check whether the concrete value of `d` satisfies `(d==0)` just before the division is performed in that specific execution, but would not provide any insight or guarantee concerning other executions. Testing this program with random values for `n` and `d` is unlikely to detect the error, as `d` has only one chance out of 2^{32} to be zero if `d` is a 32-bit integer. Static and dynamic test generation techniques that attempt to cover specific or all feasible paths in a program will also likely miss the error since this program has a single program path which is covered no matter what inputs are used. However, the latter techniques could be helpful provided that a test `if (d==0) error()` was inserted before the division `(n/d)`: they could then attempt to generate an input value for `d` that satisfies the constraint `(d==0)`, now present in the program path, and detect the error. This is essentially what *active property checking* does: it injects *at runtime* additional symbolic constraints that, when solvable by a constraint solver, will generate new test inputs leading to property violations.

We introduce *active property checking*, which extends runtime checking by checking whether the property is satisfied by *all* program executions that follow the same program path. This check is performed on a dynamic symbolic execution of the given program path using a constraint solver. If the check fails, the constraint solver generates an alternative program input triggering a new program execution that follows the same program path but exhibits a property violation. We call this "active" checking because a constraint solver is used to "actively" look for inputs that cause a runtime check to fail. Chapter 3 describes our formalism for active property checking, as well as reports on the effectiveness of several active checkers.

A particular class of interest for active property checking involves *integer bugs. Integer bugs* result from a mismatch between machine arithmetic and mathematical arithmetic. For example, machine arithmetic has bounded precision; if an expression has a value greater than the maximum integer that can be represented, the value wraps around to fit in machine precision. This can cause the value stored to be smaller than expected by the programmer. If, for example, a wrapped value is used as an argument to `malloc`, the result is an object that is smaller than expected, which can lead to a buffer overflow later if the programmer is not careful. This kind of bug is often known as an integer overflow bug.

Integer overflow bugs recently became the second most common bug type in security advisories from OS vendors [23]. These kinds of bugs are pervasive and can, in many cases, cause serious security vulnerabilities. Therefore, eliminating such bugs is important for improving software security. Unfortunately, traditional static and dynamic analysis techniques are poorly suited to detecting integer-related bugs. Static analysis yields false positives due to the difficulty of reasoning precisely about integer values for variables, and dynamic analysis may kill program executions that exhibit harmless integer overflows. Dynamic test generation is better suited to finding such bugs, in contrast, because we can use active property checking for integer bugs as a way to guide the search but notify a human being only when a test case that crashes the program is found.

We develop new methods for finding a broad class of integer bugs with dynamic test generation. Of special note are *signed/unsigned conversion errors*, which are bugs that require reasoning about the entire trace of the program execution. This reasoning goes beyond the scope of our formalism for active property checking in Chapter 3 and requires new ideas. Therefore, we devise methods for performing such reasoning in a memory efficient way that allows us to scale the inference to traces of millions of instructions. In Chapter 4 we describe these methods and report on their effectiveness.

1.3 Putting It All Together

We have implemented whitebox fuzz testing with active property checking in two different systems, both of which scale to applications with millions of instructions in an execution trace. The first is SAGE, short for *Scalable, Automated, Guided Execution*, which a whole-program whitebox file fuzzing tool for x86 Windows applications. The second is SmartFuzz, which is a whole-program file fuzzing tool for Linux applications using the Valgrind framework. Both systems focus on file reading programs such as media players or image conversion tools, but the techniques in this thesis could also apply to networked or distributed applications. In addition to the practical benefits of having the technique available on multiple platforms, the performance of whitebox fuzz testing on different platforms with different implementations gives us information about the technique's effectiveness in itself, as opposed to the effectiveness of a particular implementation.

Our results show that dynamic test generation is remarkably effective in finding new defects in large applications that were previously well-tested. In particular, Whitebox fuzzing is capable of finding bugs that are beyond the reach of blackbox fuzzers. For instance, without any format-specific knowledge, SAGE detects the critical MS07-017 ANI vulnerability, which was missed by extensive blackbox fuzzing and static analysis.

Furthermore both whitebox and blackbox fuzz testing find so many test cases exhibiting bugs that we must address the *bug triage problem*. That is, given a test case, is the bug it exhibits new, and if it is new, how serious is it? The bug triage problem is common in static program analysis and blackbox fuzzing, but has not been faced until now in the context of dynamic test generation [40, 17, 81, 70, 57, 46].

To help us prioritize and manage these bug reports and streamline the process of reporting them to developers, we built *Metafuzz*, a web service for tracking test cases and bugs. Metafuzz helps minimize the amount of human time required to find high-quality bugs and report them to developers, which is important because human time is the most expensive resource in a testing framework. We also develop new methods for *bug bucketing*, the process of grouping test cases by the bugs they exhibit. We used Metafuzz to coordinate large scale experiments with SmartFuzz and with the `zzuf` blackbox fuzz testing tool, running on the Amazon Elastic Compute Cloud.

Our goal is a service to run dynamic test generation continuously on a wide range of software, automatically finding high-value bugs and reporting them with a minimum of human effort. Our work shows this goal is within reach.

1.4 SMT Solvers and Security

Our work makes use of solvers for *satisfiability modulo theories*, also known as *SMT solvers*. These solvers take sentences that consist of statements in some underlying theory combined with first-order logic connectives. The output of the solver is a model that makes the sentence true. Many recent SMT solvers make use of an underlying solver for Boolean satisfiability, taking advantage of the rapid recent progress in such solvers.

While these kinds of decision procedures have a long and distinguished history, their application to software security is much more recent. For example, Ganapathy et al. created a model for the *sprintf* function, then used the UCLID solver [13] to find *format string vulnerabilities* in the function, in which a specially crafted format string causes a memory corruption error. Chirstodorescu et al. also used a decision procedure to identify "semantic no-op" code patterns and defeat code obfuscation [21]. Xie and Aiken use a SAT solver directly to perform static analysis of the entire Linux kernel looking for violations of lock discipline and memory safety errors, which can lead to security vulnerabilities [88]. Such SAT solvers have also found their way into traditional interactive verification tasks [29]. More recently, Heelan has begun applying SMT solvers to the question of generating exploits for known vulnerabilities [51].

Most closely related to our work, DART, EXE, and KLEE also use SMT solvers for creating test cases using dynamic analysis of the code under test [40, 17, 18]. In fact, EXE and KLEE use the same solver, STP, as we do in our SmartFuzz tool. Both DART and the version of the SAGE tool described in this thesis, on the other hand, use a linear equation solver that does not understand the full theory of bitvector arithmetic; non-linear equations are “concretized” using the values for variables observed on a program run.

All of this work depends on the existence of solvers that can quickly handle large instances arising from program analysis. The SMT-COMP competition has been a key factor in spurring advances in such solvers [5]. One of the early fast solvers was Yices, which won the 2006 SMT-COMP competition [32]. More recently, solvers such as STP, z3, and Beaver have been developed specifically for use in software analysis applications [36] [28] [54]. Malik and Zhang survey recent advances in SAT solvers, while Barrett et al. survey the current state of SMT solvers [66, 4].

1.5 Contributions and Statement on Joint Work

This thesis makes the following contributions:

- We introduce a new search algorithm for systematic test generation that is optimized for large applications with large input files and exhibiting long execution traces where the search is bound to be incomplete (Chapter 2);
- We introduce optimizations for checking multiple properties of a program simultaneously using dynamic test generation, and we formalize the notion of *active property checking* (Chapter 3);
- We describe the implementation of SAGE and SmartFuzz: the engineering choices behind their symbolic execution algorithm and the key optimization techniques enabling both tools to scale to program traces with hundreds of millions of instructions (Chapters 2 and 4);

- We develop methods for coordinating large scale experiments with fuzz testing techniques, including methods to address the defect triage problem (Chapter 4);
- We report on extensive experiments with these tools, including examples of discovered defects, the effectiveness of active property checking techniques, the performance of the constraint solvers used, and the cost per bug when running fuzz testing with the Amazon Elastic Compute Cloud framework (Chapters 2, 3, and 4).

Chapter 2 is joint work with Patrice Godefroid and Michael Y. Levin, and it appeared at the 2008 Network Distributed Security Systems Conference [43]. Chapter 3 is joint work with Patrice Godefroid and Michael Y. Levin, and it appeared at the 2008 ACM EMSOFT conference [41]. Chapter 4 is joint work with Xue Cong Li and David Wagner, and it appeared at the Usenix Security 2009 conference [69].

The development of the SAGE system was led by Michael Y. Levin. In addition to David Molnar, contributors include Patrice Godefroid, Dennis Jeffries, and Adam Kiezun. The SAGE system builds on the iDNA/Time Travel Debugging and Nirvana infrastructure produced by the Microsoft Center for Software Excellence [7].

The development of the SmartFuzz system was led by David Molnar. In addition to David Molnar, contributors include Xue Cong Li, Sushant Shankar, Shiuan-Tzuo Shen, David Wagner, and Mark Winterrowd. The SmartFuzz system builds on the Valgrind infrastructure produced by Nicholas Nethercote and Julian Seward [74].

Chapter 2

Automated Whitebox Fuzz Testing

2.1 A Whitebox Fuzzing Algorithm

Consider the program shown in Figure 2.1. This program takes 4 bytes as input and contains an error when the value of the variable `cnt` is greater than or equal to `3` at the end of the function `top`. Running the program with random values for the 4 input bytes is unlikely to discover the error: there are 5 values leading to the error out of $2^{(8*4)}$ possible values for 4 bytes, i.e., a probability of about $1/2^{30}$ to hit the error with random testing, including blackbox fuzzing. This problem is typical of random testing: it is difficult to generate input values that will drive the program through all its possible execution paths.

In contrast, whitebox *dynamic test generation* can easily find the error in this program: it consists in executing the program starting with some initial inputs, performing a dynamic symbolic execution to collect constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next executions of the program towards alternative

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

Figure 2.1: Example of program.

program branches. This process is repeated until a given specific program statement or path is executed [57, 46], or until all (or many) feasible program paths of the program are exercised [40, 17].

For the example above, assume we start running the function `top` with the initial 4-letters string `good`. Figure 2.2 shows the set of all feasible program paths for the function `top`. The leftmost path represents the first run of the program on input `good` and corresponds to the program path ρ including all 4 else-branches of all conditional if-statements in the program. The leaf for that path is labeled with `0` to denote the value of the variable `cnt` at the end of the run. Intertwined with the normal execution, a symbolic execution collects the predicates $i_0 \neq \texttt{b}$, $i_1 \neq \texttt{a}$, $i_2 \neq \texttt{d}$ and $i_3 \neq !$ according to how the conditionals evaluate, and where i_0, i_1, i_2 and i_3 are *symbolic variables* that represent the values of the memory locations of the input variables `input[0]`, `input[1]`, `input[2]` and `input[3]`,

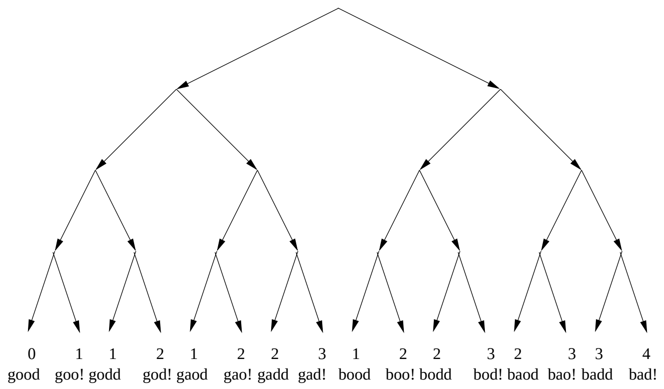


Figure 2.2: Search space for the example of Figure 2.1 with the value of the variable `cnt` at the end of each run and the corresponding input string.

respectively.

The path constraint $\phi_\rho = \langle i_0 \neq \texttt{b}, i_1 \neq \texttt{a}, i_2 \neq \texttt{d}, i_3 \neq ! \rangle$ represents an equivalence class of input vectors, namely all the input vectors that drive the program through the path that was just executed. To force the program through a different equivalence class, one can calculate a solution to a different path constraint, say, $\langle i_0 \neq \texttt{b}, i_1 \neq \texttt{a}, i_2 \neq \texttt{d}, i_3 = ! \rangle$ obtained by negating the last predicate of the current path constraint. A solution to this path constraint is $(i_0 = \texttt{g}, i_1 = \texttt{o}, i_2 = \texttt{o}, i_3 = !)$. Running the program `top` with this new input `goo!` exercises a new program path depicted by the second leftmost path in Figure 2.2. By repeating this process, the set of all 16 possible execution paths of this program can be exercised. If this systematic search is performed in depth-first order, these 16 executions are explored from left to right on the Figure. The error is then reached for the first time with `cnt==3` during the 8th run, and full branch/block coverage is achieved after the 9th run.

2.1.1 Limitations

Systematic dynamic test generation [40, 17] as briefly described above has two main limitations.

Path explosion: systematically executing all feasible program paths does not scale to large, realistic programs. Path explosion can be alleviated by performing dynamic test generation *compositionally* [38], by testing functions in isolation, encoding test results as *function summaries* expressed using function input preconditions and output postconditions, and then re-using those summaries when testing higher-level functions. Although the use of summaries in software testing seems promising, achieving full path coverage when testing large applications with hundreds of millions of instructions is still problematic within a limited search period, say, one night, even when using summaries.

Imperfect symbolic execution: symbolic execution of large programs is bound to be imprecise due to complex program statements (pointer manipulations, arithmetic operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision at a reasonable cost. Whenever symbolic execution is not possible, concrete values can be used to simplify constraints and carry on with a simplified, partial symbolic execution [40]. Randomization can also help by suggesting concrete values whenever automated reasoning is difficult. Whenever an actual execution path does not match the program path predicted by symbolic execution for a given input vector, we say that a *divergence* has occurred. A divergence can be detected by recording a predicted execution path as a bit vector (one bit for each conditional branch outcome) and checking that the expected path is actually taken in the subsequent test run.

2.1.2 Generational Search

We now present a new search algorithm that is designed to address these fundamental practical limitations. Specifically, our algorithm has the following prominent features:

- it is designed to systematically yet partially explore the state spaces of large applications executed with large inputs (thousands of symbolic variables) and with very deep paths (hundreds of millions of instructions);
- it maximizes the number of new tests generated from each symbolic execution (which are long and expensive in our context) while avoiding any redundancy in the search;
- it uses heuristics to maximize code coverage as quickly as possible, with the goal of finding bugs faster;
- it is resilient to divergences: whenever divergences occur, the search is able to recover and continue.

This new search algorithm is presented in two parts in Figures 2.3 and 2.4. The main `Search` procedure of Figure 2.3 is mostly standard. It places the initial input `inputSeed` in a `workList` (line 3) and runs the program to check whether any bugs are detected during the first execution (line 4). The inputs in the `workList` are then processed (line 5) by selecting an element (line 6) and expanding it (line 7) to generate new inputs with the function `ExpandExecution` described later in Figure 2.4. For each of those `childInputs`, the program under test is run with that input. This execution is checked for errors (line 10) and is assigned a `Score` (line 11), as discussed below, before being added to

```
1  Search(inputSeed){
2    inputSeed.bound = 0;
3    workList = {inputSeed};
4    Run&Check(inputSeed);
5    while (workList not empty) {//new children
6      input = PickFirstItem(workList);
7      childInputs = ExpandExecution(input);
8      while (childInputs not empty) {
9        newInput = PickOneItem(childInputs);
10       Run&Check(newInput);
11       Score(newInput);
12       workList = workList + newInput;
13     }
14   }
15 }
```

Figure 2.3: Search algorithm.

the `workList` (line 12) which is sorted by those scores.

The main originality of our search algorithm is in the way children are expanded as shown in Figure 2.4. Given an `input` (line 1), the function `ExpandExecution` symbolically executes the program under test with that `input` and generates a *path constraint* `PC` (line 4) as defined earlier. `PC` is a conjunction of `|PC|` constraints, each corresponding to a conditional statement in the program and expressed using symbolic variables representing values of input parameters. Then, our algorithm attempts to *expand every* constraint in

```
1  ExpandExecution(input) {
2    childInputs = {};
3    // symbolically execute (program,input)
4    PC = ComputePathConstraint(input);
5    for (j=input.bound; j < |PC|; j++) {
6      if((PC[0..(j-1)] and not(PC[j]))
                       has a solution I){
7        newInput = input + I;
8        newInput.bound = j;
9        childInputs = childInputs + newInput;
10   }
11   return childInputs;
12 }
```

Figure 2.4: Computing new children.

the path constraint (at a position `j` greater or equal to a parameter called `input.bound` which is initially `0`). This is done by checking whether the conjunction of the part of the path constraint prior to the `j`th constraint `PC[0..(j-1)]` and of the negation of the `j`th constraint `not(PC[j])` is satisfiable. If so, a solution `I` to this new path constraint is used to update the previous solution `input` while values of input parameters not involved in the path constraint are preserved (this update is denoted by `input + I` on line 7). The resulting new input value is saved for future evaluation (line 9).

In other words, starting with an initial input `inputSeed` and initial path constraint `PC`, the new search algorithm depicted in Figures 2.3 and 2.4 will attempt to expand *all*

`|PC|` constraints in `PC`, instead of just the last one with a *depth-first search*, or the first one with a *breadth-first search*. To prevent these child sub-searches from redundantly exploring overlapping parts of the search space, a parameter `bound` is used to limit the backtracking of each sub-search above the branch where the sub-search started off its parent. Because each execution is typically expanded with many children, we call such a search order a *generational search*.

Consider again the program shown in Figure 2.1. Assuming the initial input is the 4-letters string `good`, the leftmost path in the tree of Figure 2.2 represents the first run of the program on that input. From this *parent* run, a generational search generates four *first-generation* children which correspond to the four paths whose leafs are labeled with `1`. Indeed, those four paths each correspond to negating *one* constraint in the original path constraint of the leftmost parent run. Each of those first generation execution paths can in turn be expanded by the procedure of Figure 2.4 to generate (zero or more) *second-generation* children. There are six of those and each one is depicted with a leaf label of `2` to the right of their (first-generation) parent in Figure 2.2. By repeating this process, all feasible execution paths of the function `top` are eventually generated exactly once. For this example, the value of the variable `cnt` denotes exactly the generation number of each run.

Since the procedure `ExpandExecution` of Figure 2.4 expands all constraints in the current path constraint (below the current `bound`) instead of just one, it maximizes the number of new test inputs generated from each symbolic execution. Although this optimization is perhaps not significant when exhaustively exploring all execution paths of small programs like the one of Figure 2.1, it is important when *symbolic execution takes a* *long time*, as is the case for large applications where exercising all execution paths is virtually hopeless anyway. This point will be further discussed in Section 2.2 and illustrated with the experiments reported in Section 2.3.

In this scenario, we want to exploit as much as possible the first symbolic execution performed with an initial input and to systematically explore all its first-generation children. This search strategy works best if that initial input is *well formed*. Indeed, it will be more likely to exercise more of the program's code and hence generate more constraints to be negated, thus more children, as will be shown with experiments in Section 2.3. The importance given to the first input is similar to what is done with traditional, blackbox fuzz testing, hence our use of the term *whitebox fuzzing* for the search technique introduced in this paper.

The expansion of the children of the first parent run is itself prioritized by using a heuristic to attempt to maximize block coverage as quickly as possible, with the hope of finding more bugs faster. The function `Score` (line 11 of Figure 2.3) computes the incremental block coverage obtained by executing the `newInput` compared to all previous runs. For instance, a `newInput` that triggers an execution uncovering 100 new blocks would be assigned a score of 100. Next, (line 12), the `newInput` is inserted into the `workList` according to its score, with the highest scores placed at the head of the list. Note that all children compete with each other to be expanded next, regardless of their generation number.

Our block-coverage heuristic is related to the "Best-First Search" of EXE [17]. However, the overall search strategy is different: while EXE uses a depth-first search that occasionally picks the next child to explore using a block-coverage heuristic, a generational search tests all children of each expanded execution, and scores their entire runs before picking the best one from the resulting `workList`.

The block-coverage heuristics computed with the function `Score` also helps dealing with *divergences* as defined in the previous section, i.e., executions diverging from the expected path constraint to be taken next. The occurrence of a single divergence compromises the completeness of the search, but this is not the main issue in practice since the search is bound to be incomplete for very large search spaces anyway. A more worrisome issue is that divergences may prevent the search from making any progress. For instance, a depth-first search which diverges from a path p to a previously explored path p' would cycle forever between that path p' and the subsequent divergent run p. In contrast, our generational search tolerates divergences and can recover from this pathological case. Indeed, each run spawns many children, instead of a single one as with a depth-first search, and, if a child run p divergences to a previous one p', that child p will have a zero score and hence be placed at the end of the `workList` without hampering the expansion of other, non-divergent children. Dealing with divergences is another important feature of our algorithm for handling large applications for which symbolic execution is bound to be imperfect/incomplete, as will be demonstrated in Section 2.3.

Finally, we note that a generational search parallelizes well, since children can be checked and scored independently; only the work list and overall block coverage need to be shared.

2.2 The SAGE System

The generational search algorithm presented in the previous section has been implemented in a new tool named SAGE, which stands for *Scalable, Automated, Guided Execution*. SAGE can test any file-reading program running on Windows by treating bytes read from files as symbolic inputs. Another key novelty of SAGE is that it performs symbolic execution of program traces at the x86 binary level. This section justifies this design choice by arguing how it allows SAGE to handle a wide variety of large production applications. This design decision raises challenges that are different from those faced by source-code level symbolic execution. We describe these challenges and show how they are addressed in our implementation. Finally, we outline key optimizations that are crucial in scaling to large programs.

2.2.1 System Architecture

SAGE performs a generational search by repeating four different types of tasks. The `Tester` task implements the function `Run&Check` by executing a program under test on a test input and looking for unusual events such as access violation exceptions and extreme memory consumption. The subsequent tasks proceed only if the `Tester` task did not encounter any such errors. If `Tester` detects an error, it saves the test case and performs automated triage as discussed in Section 2.3.

The `Tracer` task runs the target program on the same input file again, this time recording a log of the run which will be used by the following tasks to replay the program execution offline. This task uses the `iDNA` framework [7] to collect complete execution traces at the machine-instruction level.

The CoverageCollector task replays the recorded execution to compute which basic blocks were executed during the run. SAGE uses this information to implement the function Score discussed in the previous section.

Lastly, the SymbolicExecutor task implements the function ExpandExecution of Section 2.1.2 by replaying the recorded execution once again, this time to collect input-related constraints and generate new inputs using the constraint solver Disolver [49].

Both the CoverageCollector and SymbolicExecutor tasks are built on top of the trace replay framework TruScan [73] which consumes trace files generated by iDNA and virtually re-executes the recorded runs. TruScan offers several features that substantially simplify symbolic execution. These include instruction decoding, providing an interface to program symbol information, monitoring various input/output system calls, keeping track of heap and stack frame allocations, and tracking the flow of data through the program structures.

2.2.2 Trace-based x86 Constraint Generation

SAGE's constraint generation differs from previous dynamic test generation implementations [40, 81, 17] in two main ways. First, instead of a source-based instrumentation, SAGE adopts a *machine-code-based* approach for three main reasons:

Multitude of languages and build processes. Source-based instrumentation must support the specific language, compiler, and build process for the program under test. There is a large upfront cost for adapting the instrumentation to a new language, compiler, or build tool. Covering many applications developed in a large company with a variety of incompatible build processes and compiler versions is a logistical nightmare. In contrast, a machine-code based symbolic-execution engine, while complicated, need be implemented only once per architecture. As we will see in Section 2.3, this choice has let us apply SAGE to a large spectrum of production software applications.

Compiler and post-build transformations. By performing symbolic execution on the binary code that actually ships, SAGE makes it possible to catch bugs not only in the target program but also in the compilation and post-processing tools, such as code obfuscators and basic block transformers, that may introduce subtle differences between the semantics of the source and the final product.

Unavailability of source. It might be difficult to obtain source code of third-party components, or even components from different groups of the same organization. Source-based instrumentation may also be difficult for self-modifying or JITed code. SAGE avoids these issues by working at the machine-code level. While source code does have information about types and structure not immediately visible at the machine code level, we do not need this information for SAGE's path exploration.

Second, instead of an online instrumentation, SAGE adopts an *offline trace-based* constraint generation. With online generation, constraints are generated as the program is executed either by statically injected instrumentation code or with the help of dynamic binary instrumentation tools such as Nirvana [7] or `Valgrind` [74] (`Catchconv` is an example of the latter approach [70].) SAGE adopts offline trace-based constraint generation for two reasons. First, a single program may involve a large number of binary components some of which may be protected by the operating system or obfuscated, making it hard to replace them with instrumented versions. Second, inherent nondeterminism in large target programs makes debugging online constraint generation difficult. If something goes wrong in the constraint generation engine, we are unlikely to reproduce the environment leading to the problem. In contrast, constraint generation in SAGE is completely deterministic because it works from the execution trace that captures the outcome of all nondeterministic events encountered during the recorded run.

2.2.3 Constraint Generation

SAGE maintains the concrete and symbolic state of the program represented by a pair of stores associating every memory locations and registers to a byte-sized value and a *symbolic tag* respectively. A symbolic tag is an expression representing either an input value or a function of some input value. SAGE supports several kinds of tags: $\texttt{input}(m)$ represents the mth byte of the input; c represents a constant; t_1 *op* t_2 denotes the result of some arithmetic or bitwise operation *op* on the values represented by the tags t_1 and t_2; the sequence tag $\langle t_0 \ldots t_n \rangle$ where $n = 1$ or $n = 3$ describes a word- or double-word-sized value obtained by grouping byte-sized values represented by tags $t_0 \ldots t_m$ together; $\texttt{subtag}(t, i)$ where $i \in \{0 \ldots 3\}$ corresponds to the i-th byte in the word- or double-word-sized value represented by t. Note that SAGE does not currently reason about symbolic pointer dereferences. SAGE defines a fresh symbolic variable for each non-constant symbolic tag. Provided there is no confusion, we do not distinguish a tag from its associated symbolic variable in the rest of this section.

As SAGE replays the recorded program trace, it updates the concrete and symbolic stores according to the semantics of each visited instruction.

In addition to performing symbolic tag propagation, SAGE also generates *constraints* on input values. Constraints are relations over *symbolic variables*; for example, given a variable x that corresponds to the tag `input`(4), the constraint $x < 10$ denotes the fact that the fifth byte of the input is less than 10.

When the algorithm encounters an input-dependent conditional jump, it creates a constraint modeling the outcome of the branch and adds it to the path constraint composed of the constraints encountered so far.

The following simple example illustrates the process of tracking symbolic tags and collecting constraints.

```
# read 10 byte file into a
# buffer beginning at address 1000
mov ebx, 1005
mov al, byte [ebx]
dec al              # Decrement al
jz LabelForIfZero   # Jump if al == 0
```

The beginning of this fragment uses a system call to read a 10 byte file into the memory range starting from address 1000. For brevity, we omit the actual instruction sequence. As a result of replaying these instructions, SAGE updates the symbolic store by associating addresses 1000 ... 1009 with symbolic tags `input`(0) ... `input`(9) respectively. The two `mov` instructions have the effect of loading the fifth input byte into register `al`. After replaying these instructions, SAGE updates the symbolic store with a mapping of `al` to `input`(5). The effect of the last two instructions is to decrement `al` and to make a conditional jump to `LabelForIfZero` if the decremented value is 0. As a result of replaying these instructions, depending on the outcome of the branch, SAGE will add one of two constraints $t = 0$ or $t \neq 0$ where $t = \texttt{input}(5) - 1$. The former constraint is added if the branch is taken; the latter if the branch is not taken.

This leads us to one of the key difficulties in generating constraints from a stream of x86 machine instructions—dealing with the two-stage nature of conditional expressions. When a comparison is made, it is not known how it will be used until a conditional jump instruction is executed later. The processor has a special register EFLAGS that packs a collection of status flags such as CF, SF, AF, PF, OF, and ZF. How these flags are set is determined by the outcome of various instructions. For example, CF—the first bit of EFLAGS—is the carry flag that is influenced by various arithmetic operations. In particular, it is set to 1 by a subtraction instruction whose first argument is less than the second. ZF is the zero flag located at the seventh bit of EFLAGS; it is set by a subtraction instruction if its arguments are equal. Complicating matters even further, some instructions such as `sete` and `pushf` access EFLAGS directly.

For sound handling of EFLAGS, SAGE defines bitvector tags of the form $\langle f_0 \ldots f_{n-1} \rangle$ describing an n-bit value whose bits are set according to the constraints $f_0 \ldots f_{n-1}$. In the example above, when SAGE replays the `dec` instruction, it updates the symbolic store mapping for `al` and for EFLAGS. The former becomes mapped to $\texttt{input}(5) - 1$; the latter—to the bitvector tag $\langle t < 0 \ldots t = 0 \ldots \rangle$ where $t = \texttt{input}(5) - 1$ and the two shown constraints are located at offsets 0 and 6 of the bitvector—the offsets corresponding to the positions of CF and ZF in the EFLAGS register.

Another pervasive x86 practice involves casting between byte, word, and double word objects. Even if the main code of the program under test does not contain explicit casts, it will invariably invoke some run-time library function such as `atol`, `malloc`, or `memcpy` that does.

SAGE implements sound handling of casts with the help of subtag and sequence tags. This is illustrated by the following example.

```
mov ch, byte [...]

mov cl, byte [...]

inc cx              # Increment cx
```

Let us assume that the two `mov` instructions read addresses associated with the symbolic tags t_1 and t_2. After SAGE replays these instructions, it updates the symbolic store with the mappings $\texttt{cl} \mapsto t_1$ and $\texttt{ch} \mapsto t_2$. The next instruction increments `cx`—the 16-bit register containing `cl` and `ch` as the low and high bytes respectively. Right before the increment, the contents of `cx` can be represented by the sequence tag $\langle t_1, t_2 \rangle$. The result of the increment then is the word-sized tag $t = (\langle t_1, t_2 \rangle + 1)$. To finalize the effect of the `inc` instruction, SAGE updates the symbolic store with the byte-sized mappings $\texttt{cl} \mapsto \texttt{subtag}(t, 0)$ and $\texttt{ch} \mapsto \texttt{subtag}(t, 1)$. SAGE encodes the subtag relation by the constraint $x = x' + 256 * x''$ where the word-sized symbolic variable x corresponds to t and the two byte-sized symbolic variables x' and x'' correspond to $\texttt{subtag}(t, 0)$ and $\texttt{subtag}(t, 1)$ respectively.

2.2.4 Constraint Optimization

SAGE employs a number of optimization techniques whose goal is to improve the speed and memory usage of constraint generation: *tag caching* ensures that structurally equivalent tags are mapped to the same physical object; *unrelated constraint elimination* reduces the size of constraint solver queries by removing the constraints which do not share symbolic variables with the negated constraint; *local constraint caching* skips a constraint if it has already been added to the path constraint; *flip count limit* establishes the maximum number of times a constraint generated from a particular program instruction can be flipped; *concretization* reduces the symbolic tags involving bitwise and multiplicative operators into their corresponding concrete values.

These optimizations are fairly standard in dynamic test generation. The rest of this section describes *constraint subsumption*, an optimization we found particularly useful for analyzing structured-file parsing applications.

The constraint subsumption optimization keeps track of the constraints generated from a given branch instruction. When a new constraint f is created, SAGE uses a fast syntactic check to determine whether f definitely implies or is definitely implied by another constraint generated from the same instruction. If this is the case, the implied constraint is removed from the path constraint.

The subsumption optimization has a critical impact on many programs processing structured files such as various image parsers and media players. For example, in one of the Media 2 searches described in Section 2.3, we have observed a ten-fold decrease in the number of constraints because of subsumption. Without this optimization, SAGE runs out of memory and overwhelms the constraint solver with a huge number of redundant queries.

Let us look at the details of the constraint subsumption optimization with the help of the following example:

```
mov cl, byte [...]

dec cl              # Decrement cl

ja 2                # Jump if cl > 0
```

This code fragment loads a byte into `cl` and decrements it in a loop until it becomes 0. Assuming that the byte read by the `mov` instruction is mapped to a symbolic tag t_0, the algorithm outlined in Section 2.2.3 will generate constraints $t_1 > 0, \ldots, t_{k-1} > 0$, and $t_k \leq 0$ where k is the concrete value of the loaded byte and $t_{i+1} = t_i - 1$ for $i \in \{1 \ldots k\}$. Here, the memory cost is linear in the number of loop iterations because each iteration produces a new constraint and a new symbolic tag.

The subsumption technique allows us to remove the first $k-2$ constraints because they are implied by the following constraints. We still have to hold on to a linear number of symbolic tags because each one is defined in terms of the preceding tag. To achieve constant space behavior, constraint subsumption must be performed in conjunction with *constant folding* during tag creation: $(t - c) - 1 = t - (c + 1)$. The net effect of the algorithm with constraint subsumption and constant folding on the above fragment is the path constraint with two constraints $t_0 - (k - 1) > 0$ and $t_0 - k \leq 0$.

Another hurdle arises from multi-byte tags. Consider the following loop which is similar to the loop above except that the byte-sized register `cl` is replaced by the word-sized register `cx`.

```
mov cx, word [...]

dec cx              # Decrement cx

ja 2                # Jump if cx > 0
```

Assuming that the two bytes read by the `mov` instruction are mapped to tags t'_0 and t''_0, this fragment yields constraints $s_1 > 0$, …, $s_{k-1} > 0$, and $s_k \leq 0$ where $s_{i+1} = \langle t'_i, t''_i \rangle - 1$ with $t'_i = \texttt{subtag}(s_i, 0)$ and $t''_i = \texttt{subtag}(s_i, 1)$ for $i \in \{1 \ldots k\}$. Constant folding becomes hard because each loop iteration introduces syntactically unique but semantically redundant word-size sequence tags. SAGE solves this with the help of *sequence tag simplification* which rewrites $\langle \texttt{subtag}(t, 0), \texttt{subtag}(t, 1) \rangle$ into t avoiding duplicating equivalent tags and enabling constant folding.

Constraint subsumption, constant folding, and sequence tag simplification are sufficient to guarantee constant space replay of the above fragment generating constraints $\langle t'_0, t''_0 \rangle - (k-1) > 0$ and $\langle t'_0, t''_0 \rangle - k \leq 0$. More generally, these three simple techniques enable SAGE to effectively fuzz real-world structured-file-parsing applications in which the input-bound loop pattern is pervasive.

2.3 Experiments

We first describe our initial experiences with SAGE, including several bugs found by SAGE that were missed by blackbox fuzzing efforts. Inspired by these experiences, we pursue a more systematic study of SAGE's behavior on two media-parsing applications. In particular, we focus on the importance of the starting input file for the search, the effect of our generational search vs. depth-first search, and the impact of our block coverage heuristic. In some cases, we withold details concerning the exact application tested because the bugs are still in the process of being fixed.

2.3.1 Initial Experiences

MS07-017. On 3 April 2007, Microsoft released an out of band critical security patch for code that parses ANI format animated cursors. The vulnerability was originally reported to Microsoft in December 2006 by Alex Sotirov of Determina Security Research, then made public after exploit code appeared in the wild [83]. This was only the third such out-of-band patch released by Microsoft since January 2006, indicating the seriousness of the bug. The Microsoft SDL Policy Weblog states that extensive blackbox fuzz testing of this code failed to uncover the bug, and that existing static analysis tools are not capable of finding the bug without excessive false positives [53]. SAGE, in contrast, synthesizes a new input file exhibiting the bug within hours of starting from a well-formed ANI file.

In more detail, the vulnerability results from an incomplete patch to MS05-006, which also concerned ANI parsing code. The root cause of this bug was a failure to validate a size parameter read from an `anih` record in an ANI file. Unfortunately, the patch for MS05-006 is incomplete. Only the length of the *first* `anih` record is checked. If a file has an initial `anih` record of 36 bytes or less, the check is satisfied but then an icon loading function is called on all `anih` records. The length fields of the second and subsequent records are not checked, so any of these records can trigger memory corruption.

Therefore, a test case needs at least two `anih` records to trigger the MS07-017 bug. The SDL Policy Weblog attributes the failure of blackbox fuzz testing to find MS07-017 to the fact that all of the seed files used for blackbox testing had only one `anih` record, and so none of the test cases generated would break the MS05-006 patch. While of course one could write a grammar that generates such test cases for blackbox fuzzing, this requires

```
RIFF...ACONLIST          RIFF...ACONB
B...INFOINAM....         B...INFOINAM....
3D Blue Alternat         3D Blue Alternat
e v1.1..IART....         e v1.1..IART....
................         ................
1996..anih$...$.         1996..anih$...$.
................         ................
................         ................
..rate..........         ..rate..........
..........seq ..         ..........seq ..
................         ................
..LIST....framic         ..anih....framic
on.........  ..          on.........  ..
```

Figure 2.5: On the left, an ASCII rendering of a prefix of the seed ANI file used for our search. On the right, the SAGE-generated crash for MS07-017. Note how the SAGE test case changes the `LIST` to an additional `anih` record on the next-to-last line.

effort and does not generalize beyond the single ANI format.

In contrast, SAGE can generate a crash exhibiting MS07-017 starting from a well-formed ANI file with one `anih` record, despite having no knowledge of the ANI format. Our seed file was picked arbitrarily from a library of well-formed ANI files, and we used a small test driver that called `user32.dll` to parse test case ANI files. The initial test case generated a path constraint with 341 branch constraints after parsing 1279939 total instructions over 10072 symbolic input bytes. SAGE then created a crashing ANI file at depth 72 after 7 hours 36 minutes of search and 7706 test cases, using one core of a 2 GHz AMD Opteron 270 dual-core processor running 32-bit Windows Vista with 4 GB of RAM. Figure 2.5 shows a prefix of our seed file side by side with the crashing SAGE-generated test case. Figure 2.6 shows further statistics from this test run.

Compressed File Format. We released an alpha version of SAGE to an internal testing team to look for bugs in code that handles a compressed file format. The parsing code for

Test	# SymExec	SymExecT	Init. |PC|	# Tests	Mean Depth	Mean # Instr.	Mean Size
ANI	808	19099	341	11468	178	2066087	5400
Media 1	564	5625	71	6890	73	3409376	65536
Media 2	3	3457	3202	1045	1100	271432489	27335
Media 3	17	3117	1666	2266	608	54644652	30833
Media 4	7	3108	1598	909	883	133685240	22209
Compressed File	47	1495	111	1527	65	480435	634
OfficeApp	1	3108	15745	3008	6502	923731248	45064

Figure 2.6: Statistics from 10-hour searches on seven test applications, each seeded with a well-formed input file. We report the number of `SymbolicExecutor` tasks during the search, the total time spent in all `SymbolicExecutor` tasks in seconds, the number of constraints generated from the seed file, the total number of test cases generated, the mean depth per test case in number of constraints, the mean number of instructions executed after reading the input file, and the mean size of the symbolic input in bytes.

this file format had been extensively tested with blackbox fuzzing tools, yet SAGE found two serious new bugs. The first bug was a stack overflow. The second bug was an infinite loop that caused the processing application to consume nearly 100% of the CPU. Both bugs were fixed within a week of filing, showing that the product team considered these bugs important. Figure 2.6 shows statistics from a SAGE run on this test code, seeded with a well-formed compressed file. SAGE also uncovered two separate crashes due to read access violations while parsing malformed files of a different format tested by the same team; the corresponding bugs were also fixed within a week of filing.

Media File Parsing. We applied SAGE to parsers for four widely used media file formats, which we will refer to as “Media 1,” “Media 2,” “Media 3,” and “Media 4.” Through several testing sessions, SAGE discovered crashes in each of these media files that resulted in nine distinct bug reports. For example, SAGE discovered a read violation due to the program copying zero bytes into a buffer and then reading from a non-zero offset. In addition, starting from a seed file of 100 zero bytes, SAGE synthesized a crashing Media 1 test case after 1403 test cases, demonstrating the power of SAGE to infer file structure from code. Figure 2.6 shows statistics on the size of the SAGE search for each of these parsers, when starting from a well-formed file.

Office 2007 Application. We have used SAGE to successfully synthesize crashing test cases for a large application shipped as part of Office 2007. Over the course of two 10-hour searches seeded with two different well-formed files, SAGE generated 4548 test cases, of which 43 crashed the application. The crashes we have investigated so far are NULL pointer dereference errors, and they show how SAGE can successfully reason about programs on a large scale. Figure 2.6 shows statistics from the SAGE search on one of the well-formed files.

Image Parsing. We used SAGE to exercise the image parsing code in a media player included with a variety of other applications. While our initial run did not find crashes, we used an internal tool to scan traces from SAGE-generated test cases and found several uninitialized value use errors. We reported these errors to the testing team, who expanded the result into a reproducible crash. This experience shows that SAGE can uncover serious bugs that do not immediately lead to crashes.

2.3.2 Experiment Setup

Test Plan. We focused on the Media 1 and Media 2 parsers because they are widely used. We ran a SAGE search for the Media 1 parser with five "well-formed" media files, chosen from a library of test media files. We also tested Media 1 with five "bogus" files : `bogus-1` consisting of 100 zero bytes, `bogus-2` consisting of 800 zero bytes, `bogus-3` consisting of 25600 zero bytes, `bogus-4` consisting of 100 randomly generated bytes, and `bogus-5` consisting of 800 randomly generated bytes. For each of these 10 files, we ran a 10-hour SAGE search seeded with the file to establish a baseline number of crashes found by SAGE. If a task was in progress at the end of 10 hours, we allowed it to finish, leading to search times slightly longer than 10 hours in some cases. For searches that found crashes, we then re-ran the SAGE search for 10 hours, but disabled our block coverage heuristic. We repeated the process for the Media 2 parser with five "well-formed" Media 2 files and the `bogus-1` file.

Each SAGE search used AppVerifier [22] configured to check for heap memory errors. Whenever such an error occurs, AppVerifier forces a "crash" in the application under test. We then collected crashing test cases, the absolute number of code blocks covered by the seed input, and the number of code blocks added over the course of the search. We performed our experiments on four machines, each with two dual-core AMD Opteron 270 processors running at 2 GHz. During our experiments, however, we used only one core to reduce the effect of nondeterministic task scheduling on the search results. Each machine ran 32-bit Windows Vista, with 4 GB of RAM and a 250 GB hard drive.

Triage. Because a SAGE search can generate many different test cases that exhibit the same bug, we "bucket" crashing files by the *stack hash* of the crash, which includes the address of the faulting instruction. It is possible for the same bug to be reachable by program paths with different stack hashes for the same root cause. Our experiments always report the distinct stack hashes.

Nondeterminism in Coverage Results. As part of our experiments, we measured the absolute number of blocks covered during a test run. We observed that running the same input on the same program can lead to slightly different initial coverage, even on the same machine. We believe this is due to nondeterminism associated with loading and initializing DLLs used by our test applications.

2.3.3 Results and Observations

The Appendix shows a table of results from our experiments. Here we comment on some general observations. We stress that these observations are from a limited sample size of two applications and should be taken with caution.

Symbolic execution is slow. We measured the total amount of time spent performing symbolic execution during each search. We observe that a single symbolic execution task is many times slower than testing or tracing a program. For example, the mean time for a symbolic execution task in the Media 2 search seeded with `wff-3` was 25 minutes 30 seconds, while testing a Media 2 file took seconds. At the same time, we can also observe that only a small portion of the search time was spent performing symbolic execution, because each task generated many test cases; in the Media 2 `wff-3` case, only 25% of the search time was spent in symbolic execution. This shows how a generational search effectively leverages the expensive symbolic execution task. This also shows the benefit of separating the `Tester` task from the more expensive `SymbolicExecutor` task.

Generational search is better than depth-first search. We performed several runs with depth-first search. First, we discovered that the SAGE search on Media 1 when seeded with the `bogus-1` file exhibited a pathological divergence (see Section 2) leading to premature termination of the search after 18 minutes. Upon further inspection, this divergence proved to be due to concretizing an AND operator in the path constraint. We did observe depth-first search runs for 10 hours for Media 2 searches seeded with `wff-2` and `wff-3`. Neither depth-first searches found crashes. In contrast, while a generational search seeded with `wff-2` found no crashes, a generational search seeded with `wff-3` found 15 crashing files in 4 buckets. Furthermore, the depth-first searches were inferior to the generational searches in code coverage: the `wff-2` generational search started at 51217 blocks and added 12329, while the depth-first search started with 51476 and added only 398. For `wff-3`, a generational search started at 41726 blocks and added 9564, while the depth-first search started at 41703 blocks and added 244. These different initial block coverages stem from the nondeterminism noted above, but the difference in blocks added is much larger than the difference in starting coverage. The limitations of depth-first search regarding code coverage are well known (e.g., [65]) and are due to the search being too localized. In contrast, a generational search explores alternative execution branches at all depths, simultaneously exploring all the layers of the program. Finally, we saw that a much larger percentage of the search time is spent in symbolic execution for depth-first search than for generational search, because each test case requires a new symbolic execution task. For example, for the Media 2 search seeded with `wff-3`, a depth-first search spent 10 hours and 27 minutes in symbolic execution for 18 test cases generated, out of a total of 10 hours and 35 minutes. Note that any other search algorithm that generates a single new test from each symbolic execution (like a breadth-first search) has a similar execution profile where expensive symbolic executions are poorly leveraged, hence resulting in relatively few tests being executed given a fixed time budget.

Divergences are common. Our basic test setup did not measure divergences, so we ran several instrumented test cases to measure the divergence rate. In these cases, we often observed divergence rates of over 60%. This may be due to several reasons: in our experimental setup, we concretize all non-linear operations (such as multiplication, division, and bitwise arithmetic) for efficiency, there are several x86 instructions we still do not emulate, we do not model symbolic dereferences of pointers, tracking symbolic variables may be incomplete, and we do not control all sources of nondeterminism as mentioned above. Despite this, SAGE was able to find many bugs in real applications, showing that our search technique is tolerant of such divergences.

Bogus files find few bugs. We collected crash data from our well-formed and bogus seeded SAGE searches. The bugs found by each seed file are shown, bucketed by stack hash, in Figure 2.7. Out of the 10 files used as seeds for SAGE searches on Media 1, 6 found at least one crashing test case during the search, and 5 of these 6 seeds were well-formed. Furthermore, all the bugs found in the search seeded with `bogus-1` were also found by at least one well-formed file. For SAGE searches on Media 2, out of the 6 seed files tested, 4 found at least one crashing test case, and all were well-formed. Hence, the conventional wisdom that well-formed files should be used as a starting point for fuzz testing applies to our whitebox approach as well.

Different files find different bugs. Furthermore, we observed that no single well-formed file found all distinct bugs for either Media 1 or Media 2. This suggests that using a wide variety of well-formed files is important for finding distinct bugs as each search is incomplete.

Bugs found are shallow. For each seed file, we collected the maximum generation reached by the search. We then looked at which generation the search found the last of its unique crash buckets. For the Media 1 searches, crash-finding searches seeded with well-formed files found all unique bugs within 4 generations, with a maximum number of generations between 5 and 7. Therefore, most of the bugs found by these searches are *shallow* — they are reachable in a small number of generations. The crash-finding Media 2 searches reached a maximum generation of 3, so we did not observe a trend here.

Figure 2.8 shows histograms of both crashing and non-crashing ("NoIssues") test cases by generation for Media 1 seeded with `wff-4`. We can see that most tests executed were of generations 4 to 6, yet all unique bugs can be found in generations 1 to 4. The number of test cases tested with no issues in later generations is high, but these new test cases do not discover distinct new bugs. This behavior was consistently observed in almost all our experiments, especially the "bell curve" shown in the histograms. This generational search did not go beyond generation 7 since it still has many candidate input tests to expand in smaller generations and since many tests in later generations have lower incremental-coverage scores.

No clear correlation between coverage and crashes. We measured the absolute number of blocks covered after running each test, and we compared this with the locations of the first test case to exhibit each distinct stack hash for a crash. Figure 2.9 shows the result for a Media 1 search seeded with `wff-4`; the vertical bars mark where in the search crashes with new stack hashes were discovered. While this graph suggests that an increase in coverage correlates with finding new bugs, we did not observe this universally. Several other searches follow the trends shown by the graph for `wff-2`: they found all unique bugs early on, even if code coverage increased later. We found this surprising, because we expected there to be a consistent correlation between new code explored and new bugs discovered. In both cases, the last unique bug is found partway through the search, even though crashing test cases continue to be generated.

Effect of block coverage heuristic. We compared the number of blocks added during the search between test runs that used our block coverage heuristic to pick the next child from the pool, and runs that did not. We observed only a weak trend in favor of the heuristic. For example, the Media 2 `wff-1` search added 10407 blocks starting from 48494 blocks covered, while the non-heuristic case started with 48486 blocks and added 10633, almost a dead heat. In contrast, the Media 1 `wff-1` search started with 27659 blocks and added 701, while the non-heuristic case started with 26962 blocks and added only 50. Out of 10 total search pairs, in 3 cases the heuristic added many more blocks, while in the others the numbers are close enough to be almost a tie. As noted above, however, this data is noisy due to nondeterminism observed with code coverage.

2.4 Other Related Work

Other extensions of fuzz testing have recently been developed. Most of those consist of using *grammars* for representing sets of possible inputs [79, 84]. Probabilistic weights can be assigned to production rules and used as heuristics for random test input generation. Those weights can also be defined or modified automatically using coverage data collected using lightweight dynamic program instrumentation [86]. These grammars can also include rules for corner cases to test for common pitfalls in input validation code (such as very long strings, zero values, etc.). The use of input grammars makes it possible to encode *application-specific knowledge* about the application under test, as well as *testing guidelines* to favor testing specific areas of the input space compared to others. In practice, they are often key to enable blackbox fuzzing to find interesting bugs, since the probability of finding those using pure random testing is usually very small. But writing grammars manually is tedious, expensive and scales poorly. In contrast, our whitebox fuzzing approach does not require an input grammar specification to be effective. However, the experiments of the previous section highlight the importance of the initial seed file for a given search. Those seed files could be generated using grammars used for blackbox fuzzing to increase their diversity. Also, note that blackbox fuzzing can generate and run new tests faster than whitebox fuzzing due to the cost of symbolic execution and constraint solving. As a result, it may be able to expose new paths that would not be exercised with whitebox fuzzing because of the imprecision of symbolic execution.

As previously discussed, our approach builds upon recent work on systematic dynamic test generation, introduced in [40, 16] and extended in [39, 81, 17, 38, 78]. The main differences are that we use a generational search algorithm using heuristics to find bugs as fast as possible in an incomplete search, and that we test large applications instead of unit test small ones, the latter being enabled by a trace-based x86-binary symbolic execution instead of a source-based approach. Those differences may explain why we have found more bugs than previously reported with dynamic test generation.

Our work also differs from tools such as [31], which are based on dynamic taint analysis that do not generate or solve constraints, but instead simply force branches to be taken or not taken without regard to the program state. While useful for a human auditor, this can lead to false positives in the form of spurious program crashes with data that "can't happen" in a real execution. Symbolic execution is also a key component of *static program analysis*, which has been applied to x86 binaries [3]. Static analysis is usually more efficient but less precise than dynamic analysis and testing, and their complementarity is well known [33, 39]. They can also be combined [39, 44]. *Static test generation* [56] consists of analyzing a program statically to attempt to compute input values to drive it along specific program paths *without ever executing the program.* In contrast, *dynamic* test generation extends static test generation with additional runtime information, and is therefore more general and powerful [40, 38]. Symbolic execution has also been proposed in the context of generating vulnerability signatures, either statically [11] or dynamically [25].

2.5 Conclusion

We introduced a new search algorithm, the *generational search*, for dynamic test generation that tolerates divergences and better leverages expensive symbolic execution tasks. Our system, SAGE, applied this search algorithm to find bugs in a variety of production x86 machine-code programs running on Windows. We then ran experiments to better understand the behavior of SAGE on two media parsing applications. We found that using a wide variety of well-formed input files is important for finding distinct bugs. We also observed that the number of generations explored is a better predictor than block coverage of whether a test case will find a unique new bug. In particular, most unique bugs found are found within a small number of generations.

While these observations must be treated with caution, coming from a limited sample size, they suggest a new search strategy: instead of running for a set number of hours, one could systematically search a small number of generations starting from an initial seed file and, once these test cases are exhausted, move on to a new seed file. The promise of this strategy is that it may cut off the "tail" of a generational search that only finds new instances of previously seen bugs, and thus might find more distinct bugs in the same amount of time. Future work should experiment with this search method, possibly combining it with our block-coverage heuristic applied over different seed files to avoid re-exploring the same code multiple times. The key point to investigate is whether generation depth combined with code coverage is a better indicator of when to stop testing than code coverage alone.

2.6 Additional Search Statistics

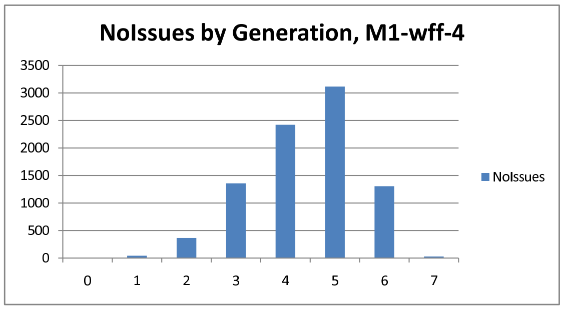
For each search, we report the number of crashes of each type: the first number is the number of distinct buckets, while the number in parentheses is the total number of crashing test cases. We also report the total search time (SearchTime), the total time spent in symbolic execution (AnalysisTime), the number of symbolic execution tasks (AnalysisTasks), blocks covered by the initial file (BlocksAtStart), new blocks discovered during the search (BlocksAdded), the total number of tests (NumTests), the test at which the last crash was found (TestsToLastCrash), the test at which the last unique bucket was found (TestsToLastUnique), the maximum generation reached (MaxGen), the generation at which the last unique bucket was found (GenToLastUnique), and the mean number of file positions changed for each generated test case (Mean Changes).

In the crash rows, the first number is the number of distinct buckets, while the number in parentheses is the total number of crashing test cases. Divergence percentages are calculated using the total number of AnalysisTasks minus one, because the first task cannot diverge.

stack hash	wff-1	wff-2	wff-3	wff-4	wff-5	bogus-1
1867196225	×	×	×	×	×	
2031962117	×	×	×	×	×	
612334691		×	×			
1061959981			×	×		
1212954973			×			×
1011628381			×	×		×
842674295				×		
1246509355			×	×		×
1527393075				×		
1277839407					×	
1392730167					×	
1951025690			×			

stack hash	wff-1	wff-3	wff-4	wff-5
790577684	×	×	×	×
825233195	×	×		×
795945252	×	×	×	×
1060863579	×	×	×	×
1043337003			×	
808455977				×
1162567688				×

Figure 2.7: SAGE found 12 distinct stack hashes (shown left) from 357 Media 1 crashing files and 7 distinct stack hashes (shown right) from 88 Media 2 crashing files.



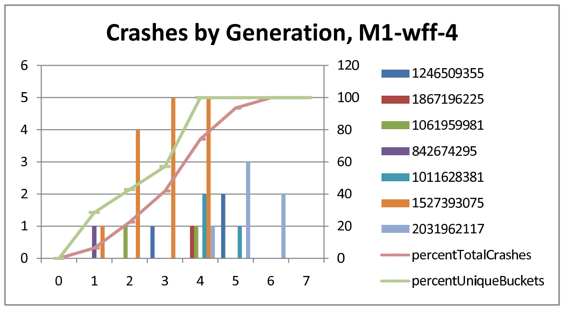
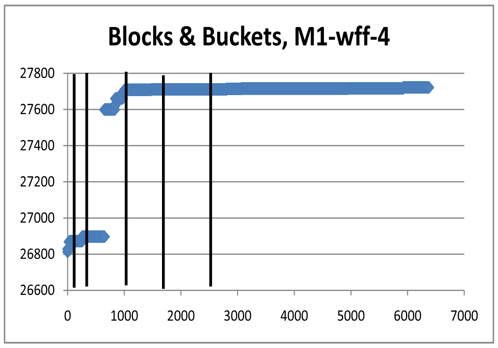


Figure 2.8: Histograms of test cases and of crashes by generation for Media 1 seeded with `wff-4`.



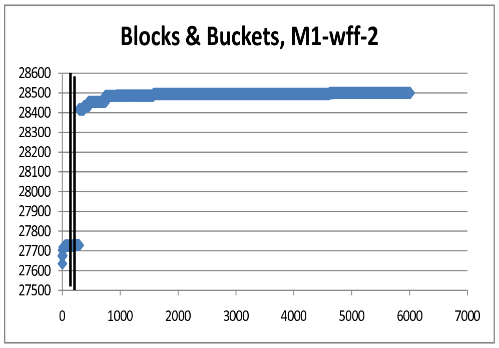


Figure 2.9: Coverage and initial discovery of stack hashes for Media 1 seeded with `wff-4` and `wff-2`. The leftmost bar represents multiple distinct crashes found early in the search; all other bars represent a single distinct crash first found at this position in the search.

Media 1:	wff-1	wff-1nh	wff-2	wff-2nh	wff-3	wff-3nh	wff-4	wff-4nh
NULL	1 (46)	1 (32)	1(23)	1(12)	1(32)	1(26)	1(13)	1(1)
ReadAV	1 (40)	1 (16)	2(32)	2(13)	7(94)	4(74)	6(15)	5(45)
WriteAV	0	0	0	0	0	1(1)	1(3)	1(1)
SearchTime	10h7s	10h11s	10h4s	10h20s	10h7s	10h12s	10h34s	9h29m2s
AnalysisTime(s)	5625	4388	16565	11729	5082	6794	5545	7671
AnalysisTasks	564	545	519	982	505	752	674	878
BlocksAtStart	27659	26962	27635	26955	27626	27588	26812	26955
BlocksAdded	701	50	865	111	96	804	910	96
NumTests	6890	7252	6091	14400	6573	10669	8668	15280
TestsToLastCrash	6845	7242	5315	13616	6571	10563	6847	15279
TestsToLastUnique	168	5860	266	13516	5488	2850	2759	1132
MaxGen	6	6	6	8	6	7	7	8
GenToLastUnique	3 (50%)	5 (83%)	2 (33%)	7 (87.5%)	4 (66%)	3 (43%)	4 (57%)	3 (37.5%)
Mean Changes	1	1	1	1	1	1	1	1

Media 1:	wff-5	wff-5nh	bogus-1	bogus-1nh	bogus-2	bogus-3	bogus-4	bogus-5
NULL	1(25)	1(15)	0	0	0	0	0	0
ReadAV	3(44)	3(56)	3(3)	1(1)	0	0	0	0
WriteAV	0	0	0	0	0	0	0	0
SearchTime	10h8s	10h4s	10h8s	10h14s	10h29s	9h47m15s	5m23s	5m39s
AnalysisTime(s)	21614	22005	11640	13156	3885	4480	214	234
AnalysisTasks	515	394	1546	1852	502	495	35	35
BlocksAtStart	27913	27680	27010	26965	27021	27022	24691	24692
BlocksAdded	109	113	130	60	61	74	57	41
NumTests	4186	2994	12190	15594	13945	13180	35	35
TestsToLastCrash	4175	2942	1403	11474	NA	NA	NA	NA
TestsToLastUnique	1504	704	1403	11474	NA	NA	NA	NA
MaxGen	5	4	14	13	8	9	9	9
GenToLastUnique	3 (60%)	3 (75%)	10 (71%)	11 (84%)	NA	NA	NA	NA

Media 2:	wff-1	wff-1nh	wff-2	wff-3	wff-3nh	wff-4	wff-4nh	wff-5	wff-5nh	bogus1
NULL	0	0	0	0	0	0	0	0	0	0
ReadAV	4(9)	4(9)	0	4(15)	4(14)	4(6)	3(3)	5(14)	4(12)	0
WriteAV	0	0	0	0	0	0	0	1(1)	0	0
SearchTime	10h12s	10h5s	10h6s	10h17s	10h1s	10h3s	10h7s	10h3s	10h6s	10h13s
AnalysisTime(s)	3457	3564	1517	9182	8513	1510	2195	10522	14386	14454
AnalysisTasks	3	3	1	6	7	2	2	6	6	1352
BlocksAtStart	48494	48486	51217	41726	41746	48729	48778	41917	42041	20008
BlocksAdded	10407	10633	12329	9564	8643	10379	10022	8980	8746	14743
NumTests	1045	1014	777	1253	1343	1174	948	1360	980	4165
TestsToLastCrash	1042	989	NA	1143	1231	1148	576	1202	877	NA
TestsToLastUnique	461	402	NA	625	969	658	576	619	877	NA
MaxGen	2	2	1	3	2	2	2	3	2	14
GenToLastUnique	2 (100%)	2 (100%)	NA	2 (66%)	2 (100%)	2 (100%)	1 (50%)	2	2	NA
Mean Changes	3	3	4	4	3.5	5	5.5	4	4	2.9

	wff-1	wff-1-nh	wff-2	wff-3	wff-3-nh	wff-4	wff-4-nh	wff-5	wff-5-nh	bogus-1
NULL	0	0	0	0	0	0	0	0	0	0
ReadAV	4(9)	4(9)	0	4(15)	4(14)	4(6)	3(3)	5(14)	4(12)	0
WriteAV	0	0	0	0	0	0	0	1(1)	0	0
SearchTime	10h12s	10h5s	10h6s	10h17s	10h1s	10h3s	10h7s	10h3s	10h7s	10h13s
AnalysisTime	3457	3564	1517	9182	8513	1510	2195	10522	14386	14454
AnalysisTasks	3	3	1	6	7	2	2	6	6	1352
Divergences	2 (100%)	2 (100%)	0	2 (40%)	6 (100%)	1 (100%)	1 (100%)	3 (60%)	5 (83 %)	1083 (83 %)
BlocksAtStart	48494	48486	51217	41726	41746	48729	48778	41917	42041	20008
BlocksAdded	10407	10633	12329	9564	8643	10379	10022	8980	8746	14743
NumTests	1045	1014	777	1253	1343	1174	948	1360	14386	4165
TestsToLastCrash	1042	909	NA	1143	1231	1148	576	1202	877	NA
TestsToLastUnique	461	402	NA	625	969	658	576	619	877	NA
MaxGen	2	2	1	3	2	2	2	3	2	14
GenToLastUnique	2	2	NA	2	2	2	1	2	2	NA
Mean Changes/Test	3	3	4	4	3.5	5	5.5	4	4	2.9

Figure 2.11: Search statistics for Media 2.

Chapter 3

Active Property Checking

3.1 Introduction

Code inspection is one of the primary ways serious security bugs can be found and fixed before deployment, but manual code inspection is extremely expensive. During the last decade, code inspection for standard programming errors has largely been automated with static code analysis. Commercial static program analysis tools (e.g., [14, 48]) are now routinely used in many software development organizations. These tools are popular because they find many real software bugs, thanks to three main ingredients: they are *automatic*, they are *scalable*, and they check *many properties*. Intuitively, any tool that is able to check automatically (with good enough precision) millions of lines of code against hundreds of coding rules and properties is bound to find on average, say, one bug every thousand lines of code.

Our research goal is to automate, as much as possible, an even more expensive part of the software development process, namely *software testing*, which usually accounts for about 50% of the R&D budget of software development organizations. In particular, we want to automate *test generation* by leveraging recent advances in program analysis, automated constraint solving, and the increasing computation power available on modern computers. Compared to static analysis, test generation has a key benefit: the test case itself demonstrates the presence of a bug; there are no "false positives." To replicate the success of static program analysis in the testing space, we need the same key ingredients: automation, scalability and the ability to check many properties.

Automating test generation from program analysis is an old idea [56, 72], and work in this area can roughly be partitioned into two groups: static versus dynamic test geneneration. *Static test generation* [56, 6, 85, 26] consists of analyzing a program statically to attempt to compute input values to drive its executions along specific program paths. In contrast, *dynamic test generation* [57, 40, 17, 43] consists in executing the program, typically starting with some random inputs, while simultaneously performing a symbolic execution to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer program executions along alternative program paths. Since dynamic test generation extends static test generation with additional runtime information, it can be more powerful [40]. Scalability in this context has been recently addressed in [38, 43]. The motivation of the present work is to address the third challenge, which has been largely unexplored so far: how to dynamically check many properties simultaneously, thoroughly and efficiently, in order to maximize the chances of finding bugs during an automated testing session?

Traditional runtime checking tools like Purify [50], Valgrind [74] and AppVerifier [22] check a *single* program execution against a set of properties (such as the absence of buffer overflows, uninitialized variables or memory leaks). We show in this paper how such traditional *passive* runtime property checkers can be extended to *actively* search for property violations. Consider the simple program:

```
int divide(int n,int d) { // n and d are inputs
  return (n/d); // division-by-zero error if d==0
}
```

The program `divide` takes two integers `n` and `d` as inputs and computes their division. If the denominator `d` is zero, an error occurs. To catch this error, a traditional runtime checker for division-by-zero would simply check whether the concrete value of `d` satisfies `(d==0)` just before the division is performed in that specific execution, but would not provide any insight or guarantee concerning other executions. Testing this program with random values for `n` and `d` is unlikely to detect the error, as `d` has only one chance out of 2^{32} to be zero if `d` is a 32-bit integer. Static and dynamic test generation techniques that attempt to cover specific or all feasible paths in a program will also likely miss the error since this program has a single program path which is covered no matter what inputs are used. However, the latter techniques could be helpful provided that a test `if (d==0) error()` was inserted before the division `(n/d)`: they could then attempt to generate an input value for `d` that satisfies the constraint `(d==0)`, now present in the program path, and detect the error. This is essentially what *active property checking* does: it injects *at runtime* additional symbolic constraints that, when solvable by a constraint solver, will generate new test inputs leading to property violations.

In other words, *active property checking* extends runtime checking by checking whether the property is satisfied by *all* program executions that follow the same program path. This check is performed on a dynamic symbolic execution of the given program path using a constraint solver. If the check fails, the constraint solver generates an alternative program input triggering a new program execution that follows the same program path but exhibits a property violation. We call this "active" checking because a constraint solver is used to "actively" look for inputs that cause a runtime check to fail. Combined with systematic dynamic test generation, which attempts to exercise all feasible paths in a program, active property checking defines a new form of program verification for terminating programs.

Closely Related Work. Checking properties at runtime on a dynamic symbolic execution of the program was suggested in [60], but may return false alarms whenever symbolic execution is imprecise, which is often the case in practice. Active property checking extends the idea of [60] by combining it with constraint solving and test generation in order to further check using a new test input whether the property is actually violated as predicted by the prior imperfect symbolic execution. This way, no false alarms are ever reported.

Although ideas similar to active property checking have been independently mentioned briefly in previous work [17, 55], prior work on dynamic test generation provides no study of active property checking, no formalization, no connections with static and dynamic type checking, no optimizations, and no evaluation with a representative set of checkers. [17] reports several bugs, but does not specify which were found using active property checking (which is only alluded to in one paragraph on page 3). While [55] discusses adding assertions to check properties in the program, it focuses on "predicting" property violating executions, not on test case generation, and their experiments report only "predicted" errors, as done in [60], which we specifically aim to extend.

Contributions. Our work provides first comprehensive study of this simple, general, yet unexplored, idea of active property checking. Specifically, our work makes several contributions:

- We formalize active property checking semantically in Section 3.3 and show how it can provide a form of program verification when combined with (sound and complete) systematic dynamic test generation (recalled in Section 3.2). The technical report associated with this work in addition shows how active type checking connects with traditional static and dynamic type checking [42].
- Section 3.4 discusses how to design a *type system* that combines static, dynamic and active checking for a simple imperative language. This clarifies the connection, difference and complementarity between *active type checking* and traditional static and dynamic type checking.
- Section 3.5 discusses how to implement active checking efficiently by minimizing the number of calls to the constraint solver, minimizing formula sizes and using two constraint caching schemes.
- Section 3.6 describes our implementation of 13 active checkers for security testing of x86 binaries of media playing Windows applications. We stress that while our implementation targets x86, the main ideas apply to any instruction set and architecture. Results of experiments searching for bugs in these applications are discussed in Section 3.7. Media playing is a key feature for some classes of embedded systems, and

media may be obtained from arbitrary, possibly adversarial, sources. For example, a flaw in the Adobe Windows Flash player was exploited during the "PWN2OWN 2008" contest to take control of a machine running Windows Vista; similar bugs may exist in mobile versions of Flash. Active property checking was able to detect several new bugs in these media playing applications.

Note that dynamic test generation is complementary to static analysis because it can analyze code that current static analysis does not handle well (e.g., due to function pointers or inline assembly). The new bugs found in codec applications studied in our paper had been missed by static analysis. Active property checking also requires no programmer intervention or annotations.

3.2 Systematic Dynamic Test Generation

Dynamic test generation (see [40] for further details) consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables x and expressed in terms of input parameters α. Side-by-side concrete and symbolic executions are performed using a concrete store Δ and a symbolic store Σ, which are mappings from program variables to concrete and symbolic values respectively. A *symbolic value* is any expression sv in some theory $\mathtt{T}$ where all free variables are exclusively input parameters α. For any variable x, $\Delta(x)$ denotes the *concrete value* of x in Δ, while $\Sigma(x)$ denotes the symbolic value of x in Σ. The judgment $\Delta \vdash e \rightarrow v$ means an expression e reduces to a concrete value v, and similarly $\Sigma \vdash e \rightarrow sv$ means that e reduces to a symbolic value sv. For notational convenience, we assume that

$\Sigma(x)$ is always defined and is simply $\Delta(x)$ by default if no expression in terms of inputs is associated with x. The notation $\Delta(x \mapsto c)$ denotes updating the mapping Δ so that x maps to c.

The program P manipulates the memory (concrete and symbolic stores) through statements, or *commands*, that are abstractions of the machine instructions actually executed. A command can be an *assignment* of the form $x := e$ (where x is a program variable and e is an expression), a *conditional statement* of the form `if` e `then` C `else` C' where e denotes a boolean expression, and C and C' are *continuations* denoting the unique next statement to be evaluated (programs considered here are thus sequential and deterministic), or `stop` corresponding to a program error or normal termination.

Given an input vector $\vec{\alpha}$ assigning a value to every input parameter α, the evaluation of a program defines a unique finite *program execution* $s_0 \xrightarrow{C_1} s_1 \ldots \xrightarrow{C_n} s_n$ that executes the finite sequence $C_1 \ldots C_n$ of commands and goes through the finite sequence $s_1 \ldots s_n$ of program states. Each *program state* is a tuple $\langle C, \Delta, \Sigma, pc\rangle$ where C is the next command to be evaluated, and pc is a special meta-variable that represents the current path constraint. For a finite sequence w of statements (i.e., a control path w), a *path constraint* pc_w is a formula of theory `T` that characterizes the input assignments for which the program executes along w. To simplify the presentation, we assume that all the program variables have some default initial concrete value in the initial concrete store Δ_0, and that the initial symbolic store Σ_0 identifies the program variables v whose values are program inputs (for all those, we have $\Sigma_0(v) = \alpha$ where α is some input parameter). We also assume that all program executions eventually terminate. Initially, pc is defined to `true`.

Systematic dynamic test generation [40] consists of systematically exploring all feasible program paths of the program under test by using path constraints and a constraint solver. By construction, a path constraint represents conditions on inputs that need be satisfied for the current program path to be executed. Given a program state $\langle C, \Delta, \Sigma, pc \rangle$ and a constraint solver for theory `T`, if C is a conditional statement of the form `if` e `then` C `else` C', any satisfying assignment to the formula $pc \wedge sv$ (respectively $pc \wedge \neg sv$) defines program inputs that will lead the program to execute the `then` (resp. `else`) branch of the conditional statement. By systematically repeating this process, such a *directed search* can enumerate all possible path constraints and eventually execute all feasible program paths.

The search is exhaustive provided that the generation of the path constraint (including the underlying symbolic execution) and the constraint solver for the given theory `T` are both *sound and complete*, that is, for all program paths w, the constraint solver returns a satisfying assignment for the path constraint pc_w *if and only if* the path is feasible (i.e., there exists some input assignment leading to its execution). In this case, in addition to finding errors such as the reachability of bad program statements (like `assert(0)`), a directed search can also prove their absence, and therefore obtain a form of program *verification.*

Theorem 1 *(adapted from [40]) Given a program P as defined above, a directed search using a path constraint generation and a constraint solver that are both sound and complete exercises all feasible program paths exactly once.*

In this case, if a program statement has not been executed when the search is over, this statement is not executable in any context.

```
1 int buggy(int x) { // x is an input
2   int buf[20];
3   buf[30]=0; // buffer overflow independent of x
4   if (x > 20)
5     return 0;
6   else
7     return buf[x]; // buffer overflow if x==20
8 }
```

Figure 3.1: Example of program with buffer overflows.

In practice, path constraint generation and constraint solving are usually not sound and complete. When a program expression cannot be expressed in the given theory `T` decided by the constraint solver, it can be simplified using concrete values of sub-expressions, or replaced by the concrete value of the entire expression. For example, if the solver handles only linear arithmetic, symbolic sub-expressions involving multiplications can be replaced by their concrete values.

3.3 Active Checkers

Even when sound and complete, a directed search based on path exploration alone can miss errors that are not path invariants, i.e., that are not violated by *all* concrete executions executing the same program path, or errors that are not caught by the program's run-time environment. This is illustrated by the simple program shown in Figure 3.1.

This program takes as (untrusted) input an integer value stored in variable `x`. A buffer overflow in line 3 will be detected at run-time only if a runtime checker monitors buffer accesses. Such a runtime checker would thus check whether any array access of the form `a[x]` satisfies the condition $0 \leq \Delta(\mathtt{x}) < b$ where $\Delta(\mathtt{x})$ is the concrete value of array index `x` and b denotes the bound of the array `a` (b is 20 for the array `buf` in the example of Figure 3.1). Let us call such a traditional runtime checker for concrete values a *passive checker*.

Moreover, a buffer overflow is also possible in line 7 provided `x==20`, yet a directed search focused on path exploration alone may miss this error. The reason is that the only condition that will appear in a path constraint for this program is `x > 20` and its negation. Since most input values for `x` that satisfy $\neg$`(x > 20)` do not cause the buffer overflow, the error will likely be undetected with a directed search as defined in the previous section.

In order to catch the buffer overflow on line 7, the program should be extended with a *symbolic* test $0 \leq \Sigma(\texttt{x}) < b$ (where $\Sigma(\texttt{x})$ denotes the symbolic value of array index `x`) just before the buffer access `buf[x]` on line 7. This will force the condition $0 \leq \texttt{x} < 20$ to appear in the path constraint of the program in order to refine the partitioning of its input values. An *active checker* for array bounds can be viewed as systematically adding such symbolic tests before all array accesses.

Formally, we define *passive checkers* and *active checkers* as follows.

Definition 1 *A* passive checker *for a property π is a function that takes as input a finite program execution w, and returns* `fail` $_\pi$ *iff the property π is violated by w.*

Because we assume all program executions terminate, properties considered here are *safety* properties. Runtime property checkers like Purify [50], Valgrind [74] and AppVerifier [22] are examples of tools implementing passive checkers.

Definition 2 *Let pc_w denote the path constraint of a finite program execution w. An active checker for a property π is a function that takes as input a finite program execution w, and* *returns a formula ϕ_C such that the formula $pc_w \wedge \neg\phi_C$ is satisfiable iff there exists a finite program execution w' violating property π and such that $pc_{w'} = pc_w$.*

Active checkers can be implemented in various ways, for instance using property monitors/automata, program rewrite rules or type checking. They can use private memory to record past events (leading to the current program state), but they are not allowed any side effect on the program. Section 3.6 discusses detailed examples of *how* active checkers can be implemented. Here are examples of specifications of active checkers.

Example 1 ***Division By Zero.*** *Given a program state where the next statement involves a division by a denominator* `d` *which depends on an input (i.e., such that $\Sigma(\texttt{d}) \neq \Delta(\texttt{d})$), an active checker for division by zero outputs the constraint $\phi_{Div} = (\Sigma(\texttt{d}) \neq 0)$.*

Example 2 ***Array Bounds.*** *Given a program state where the next statement involves an array access* `a[x]` *where* `x` *depends on an input (i.e., is such that $\Sigma(\texttt{x}) \neq \Delta(\texttt{x})$), an active checker for array bounds outputs the constraint $\phi_{Buf} = (0 \leq \Sigma(\texttt{x}) < b)$ where b denotes the bound of the array* `a`.

Example 3 ***NULL Pointer Dereference.*** *Consider a program expressed in a language where pointer dereferences are allowed. Given a program state where the next statement involves a pointer dereference* `*p` *where* `p` *depends on an input (i.e., such that $\Sigma(\texttt{p}) \neq \Delta(\texttt{p})$), an active checker for NULL pointer dereference generates the constraint $\phi_{NULL} = (\Sigma(\texttt{p}) \neq \texttt{NULL})$.*

Multiple active checkers can be used simultaneously by simply considering separately the constraints they inject in a given path constraint. Such a way, they are guaranteed not to interfere with each other (since they have no side effects). We will discuss how to combine active checkers to maximize performance in Section 3.5.

By applying an active checker for a property π to all feasible paths of a program P, we can obtain a form of *verification* for this property, that is stronger than Theorem 1.

Theorem 2 *Given a program P as defined above, if a directed search (1) uses a path constraint generation and constraint solvers that are both sound and complete,* and *(2) uses both a passive and an active checker for a property π in all program paths visited during the search, then the search reports $\texttt{fail}_\pi$ iff there exists a program input that leads to a finite execution violating ϕ.*

Proof 1 Proof Sketch: *Assume there is an input assignment that leads to a finite execution w of P violating π. Let pc_w be the path constraint for the execution path w. Since path constraint generation and constraint solving are both sound and complete, we know by Theorem 1 that w will eventually be exercised with some concrete input assignment $\vec{\alpha}'$. If the passive checker for π returns $\texttt{fail}_\pi$ for the execution of P obtained from input $\vec{\alpha}'$ (for instance, if $\vec{\alpha}' = \vec{\alpha}$), the proof is finished. Otherwise, the active checker for π will generate a formula ϕ_C and call the constraint solver with the query $pc_w \wedge \neg\phi_C$. The existence of $\vec{\alpha}'$ implies that this query is satisfiable, and the constraint solver will return a satisfying assignment from which a new input assignment $\vec{\alpha}''$ is generated ($\vec{\alpha}''$ could be $\vec{\alpha}$ itself). By construction, running the passive checker for π on the execution obtained from that new input $\vec{\alpha}''$ will return $\texttt{fail}_\pi$.*

Note that both passive checking and active checking are required in order to obtain this result, as illustrated earlier with the example of Figure 3.1. In practice, however, symbolic execution, path constraint generation, constraint solving, passive and active property checking are typically not sound and complete, and therefore active property checking reduces to testing.

3.4 Active Type Checking

Now we describe a framework for specifying active checkers that illustrates their complementarity with traditional static and dynamic checking. Our approach is inspired by the "hybrid type checking" of Flanagan [34]. In hybrid type checking, we first attempt to statically type-check a program against a type environment Γ. Flanagan defines a language λ^H for which type-checking is undecidable in general, but which is parameterized with an *algorithmic* subtyping judgment $\Gamma \vdash_{alg} S <: T$ that attempts to decide whether a type S is a subtype of a type T. This judgment may either terminate or "time out." The key property is that if the judgment terminates and answers that S is a subtype of T or that S is not a subtype of T, then this answer correct. If, on the other hand, the subtyping judgment times out, we add a *run-time* membership check for the type T to the program.

Our key addition is that we perform *symbolic membership checks* for the type T. A check that an expression e is of type T is compiled to a symbolic expression e_T. Then, a satisfying assignment to $pc \wedge \neg e_T$ yields a program input that will fail the run-time check. We call this *active type checking*, because we use a constraint solver to "actively" look for inputs that cause run-time checks to fail. Here, the run-time type membership check is the *passive check* we discussed in the previous section.

While Flanagan mentions dynamic test generation in the context of showing that

Variables	x
Values (v)	$0 \mid 1 \mid n \mid \texttt{true} \mid \texttt{false}$
Input Parameters	α
Guard Set (G)	$\emptyset \mid \{(x_1, T_1), \ldots, (x_n, T_n), (\texttt{res}, S)\}$
Commands (C)	$(G)x := e, C \mid (G)\texttt{if}\ \ e\ \ \texttt{then}\ \ C\ \ \texttt{else}\ \ C'$
	(G)$\texttt{stop}$
Operands (o)	$v \mid x$
Expressions (e)	$o \mid \texttt{op}(o_1, \ldots, o_n)$
SymExpr (se)	$\alpha \mid e$
`op`	$+ \mid - \mid * \mid \texttt{div}$
BoolExp (e, t)	$\texttt{true} \mid \texttt{false} \mid e \wedge e \mid e \vee e \mid e == e \mid e < e$
	$\mid\ \texttt{note}$
Base Types (B)	$\texttt{Bool} \mid \texttt{Int}$
Types (S, T)	$\{x : B \mid t\}, S \to T$
TyEnv (Γ)	$\emptyset \mid \Gamma, x$

Figure 3.2: Syntactic categories and grammar for guarded `SimpL`.

1. $(\emptyset)x_1 := \alpha$	$(\emptyset)x_1 := \alpha$	$pc = \texttt{true}, ch = \emptyset$
2. $(\emptyset)x_2 := 52$	$(\emptyset)x_2 := 52$	$pc = \texttt{true}, ch = \emptyset$
3. if $(x_1 > -5)$	if $(x_1 > -5)$	$pc = \alpha > -5, ch = \emptyset$
4. then $(\emptyset)x_3 := \texttt{div}(x_2, x_1)$	then $((x_1, \texttt{notzero}))W := \texttt{div}(x_2, x_1)$	$pc = \alpha > -5, ch = \{\alpha \neq 0\}$

Figure 3.3: A small program with an active check for division by zero. The left column shows the original program. The middle column shows the program with a type cast inserted as an assignment guard. The right column shows the values of the path condition *pc* and the set of checker expressions *ch* for the input $\alpha = 2$.

a given program fails to type check, active checking can *also* provide a form of *verification*. As we argued in the previous section, we can combine active type checking with systematic path exploration to verify for bounded search spaces that no inputs exist which will cause a run-time check to fail, relative to the completeness of our underlying constraint solver. In general, active type checking is *complementary* to static checking and run-time checking: improved static checking reduces the need for both run-time and active checks, while active checking may cover cases where a particular static analysis fails.

3.4.1 Judgments and Rules

We now describe active type checking for an extension of the SimpL language introduced above, which we call *guarded* SimpL. The key additions to SimpL are a set of *checker expressions ch* in symbolic evaluation, and a set of *type cast guards* before assignments. Figure 3.2 shows the syntax, and Figure 3.4 shows the list of judgments used in our system.

Evaluation. We have a symbolic command evaluation judgment $\rightarrow$ that takes a command C, a continuation C', a concrete store Δ, and a symbolic store Σ, then returns C' and

$\langle C, \Delta, \Sigma, pc, ch \rangle \rightarrow \langle C', \Delta', \Sigma', pc', ch' \rangle$	Symbolic Evaluation
$\langle C, \Delta \rangle \rightarrow \langle C', \Delta' \rangle$	Concrete Evaluation
$\Gamma \vdash C$	Well-typed program
$\Gamma \vdash e : T$	Well-typed expression
$v \in T$	Value of type T
$\Delta \in \Gamma$	Typed concrete state
$\Delta, G \vdash e$	Run-time type check
$\Sigma \in \Gamma$	Typed symbolic state
$\Sigma, G, ch \vdash ch'$	Update active checkers
$\Sigma \vdash e \rightarrow v$	Symbolic evaluation
$\Delta \vdash e \rightarrow v$	Concrete evaluation
$\Sigma[\alpha_i = v_i] \rightarrow \Delta$	Input substitution
$\Gamma \vdash C \Rightarrow C'$	Type guard insertion
$\Gamma \vdash s \Rightarrow t$	Semantic implication
$\Gamma \vdash_{alg} s \Rightarrow t$	Algorithmic implication
$\Gamma \models \sigma$	Closing substitution

Figure 3.4: Judgments.

updated stores Σ' and Δ'. This judgment also updates the *path condition pc*, and how Σ maps variables to expressions in symbolic input parameters α.

We also have a concrete evaluation judgment $\langle C, \Delta\rangle \rightarrow \langle C', \Delta'\rangle$ For passive type checking, each command in guarded `SimpL` has a set of *type cast guards*, which may be the empty set. Each type cast guard is a map from variables to types. If $(x, T) \in G$, this represents an assertion that a variable x is a member of the type T. For a command C, we define $\texttt{guards}(C)$ to be the guard set G associated with the command. The judgment $\Delta, G \vdash e$ represents a run-time check that the values

We also add an additional meta-variable *ch* to the symbolic evaluation relation. The meta-variable *ch* is a *set* of symbolic expressions obtained from casts, which we call a set of *active checker expressions*. The key idea is that each symbolic expression $\texttt{e} \in ch$ is a *symbolic* type membership check corresponding to a type assertion. At each guarded command, if the run-time check of the concrete value succeeds, then we add a symbolic version of the check to the set *ch*. As evaluation progresses, we add more and more symbolic checks to *ch*, such that *ch* contains symbolic versions of all the type checks that have occurred on the program path so far.

Because we build the symbolic expressions in the set *ch* from the type casts, if there exists a set of inputs $v_1, \ldots, v_n$ to the program that satisfies a path constraint, but falsifies $\texttt{e} \in ch$, then these inputs will fail the run-time type membership check. This allows us to check for typing failures by querying a constraint solver with the negation of an expression in *ch* together with the path condition *pc*. We state this more formally below.

We also make use of a *concrete expression evaluation* judgment $\Delta \vdash e \rightarrow v$. This judgment represents substituting concrete values for variables in the expression e and then reducing the resulting expression to obtain a value v. Similarly, the *symbolic expression evaluation* judgment $\Sigma \vdash e \rightarrow sv$ substitutes symbolic values for variables in e and reduces.

Instead of fixing a single set of operations on values in the operational semantics, we instead specify operators `op` in a modular way without allowing for full functions. Each operator `op` of arity k is given by a *type* `op` $: S_1 \rightarrow \cdots \rightarrow S_k \rightarrow T$ and a *semantic implementation function* $\tilde{op}$. We assume that $\tilde{op}$ is a partial function that is defined only on values that are members of its type signature. For concrete evaluation we compile expressions to values and call the semantic implementation function. For symbolic evaluation, we simply record a symbolic expression. This approach allows us to add additional operators without needing to re-prove our progress and type preservation theorems from scratch. We assume that the type of each operator is known globally without need to reference a type environment explicitly.

Typing Judgments. We have a set of typing judgments, one each for commands, expressions, and stores. Each typing judgment takes a *type environment* Γ. The typing judgment for expressions assigns a *type* to each expression, while the judgments for commands and stores represent whether these are *well-typed* or not.

Types in guarded `SimpL` may be either *base types* or *refinement types.* The base types are `Int` and `Bool`. Refinement types are defined using arbitrary predicates applied to existing types. For example, we can define a type `notzero` for integers that are not zero as as `notzero` : $\{x : \texttt{Int} | x \neq 0\}$. Later, we will see how this type can be used in active checking for division by zero errors. Our basic type rules are shown in 3.7, and the basic

$$\text{WT-}\textsc{ConcStr}\ \frac{\forall a.\Delta(a) \in \Gamma(a)}{\Delta \in \Gamma}$$

$$\text{WT-}\textsc{SymOp}\ \frac{\Sigma \vdash v_{s_i} \in T_i \qquad \texttt{op} : T_1 \ldots T_n \to T}{\Sigma \vdash \texttt{op}(v_{s_1}, \ldots, v_{a_n}) \in T}$$

$$\text{WT-}\textsc{SymStr}\ \frac{\forall a.\Sigma(a) \in \Gamma(a)}{\Sigma \in \Gamma}$$

Figure 3.5: Typing judgments for concrete and symbolic stores.

$$\text{TC-}\textsc{Check}\ \frac{G \vdash e_1 : T_1, \ldots, e_n : T_n \qquad T_i = \{x_i : B_i | t_i\} \qquad \forall i.\Delta \vdash t_i[x_i := e_i] \to \texttt{true}}{\Delta, G \vdash e_1, \ldots, e_n}$$

$$\text{TC-}\textsc{Upd}\ \frac{G \vdash e_1 : T_1, \ldots, e_n : T_n \qquad T_i = \{x_i : B_i | t_i\} \qquad ch' = ch \cup_i \{t_i[x_i := e_i]\}}{G, ch \vdash ch'}$$

Figure 3.6: Rules for checking type guards and updating checkers.

type rule for refinement types is WT-Base in Figure 3.8. Of special interest is the type rule T-Op. This rule states that, given a function type for the operator, each argument is of the appropriate type, and the result type is compatible with the destination of the assignment. We write $v \in T$ to mean that the value v is a member of type T, and we assume that membership for types is always decidable.

The type rules for stores are shown in Figure 3.5. Intuitively, these rules say that a concrete is well-typed if and only if all values agree with the types of the variables in Γ.

For symbolic variables, we associate each α with the type of the variable which initially points to α. We also define a judgment $\Sigma[\alpha_i = v_i] \to \Delta$ that represents substituting values v_i for the input parameters α_i and reducing each entry in Σ to obtain a concrete store Δ. This judgment will be useful later on when we discuss the soundness and completeness of the symbolic expressions built by our active checkers. The key property of this judgment is that if $\Sigma \in \Gamma$, and if for all α_i we have that $v_i \in \Gamma(\alpha_i)$, then the reduction of Σ after substitution cannot get stuck.

Subtyping. The key feature of the type system, following Flanagan's λ^H, is that it defines subtyping by logical implication between predicates for refinement types of the same base type. We first define this using a *semantic implication* judgment $\Gamma \vdash s \Rightarrow t$, shown in Figure 3.8. This judgment states that s implies t if and only if all *consistent substitutions* for variables in s and t cause the implication to hold. A consistent substitution is the substitution of a variable of a given type with a value that is a member of the same type. The rules CS-Base and CS-Ext define a judgment $\Gamma \models \sigma$ that states a substitution σ is consistent with Γ.

Unfortuantely, in general it is undecidable to evaluate this semantic implication. As a consequence, subtyping is also undecidable. Therefore, we add an *algorithmic implication* judgment $\Gamma \vdash^{a}_{alg} s \Rightarrow t$, where $a \in \{ok, \times, ?\}$. This represents an algorithm that attempts to decide statically whether the predicate s implies the predicate t. If it succeeds, then either $a = ok$ and $\Gamma \vdash s \Rightarrow t$, or $a = \times$ and $\Gamma \not\vdash s \Rightarrow t$. The judgment may also *time out* and return a "don't know" answer. In this case, our typecast insertion judgment will add a cast, which by the rule T-Cast will cause static type checking to proceed.

$$\text{T-VAR}\ \frac{(v : T) \in \Gamma}{\Gamma \vdash v : T}$$

$$\text{T-CONST}\ \frac{}{\Gamma \vdash c : ty(c)}$$

$$\text{T-INT}\ \frac{\Gamma \vdash e : S = \{x_1 : B|s\} \qquad G \vdash e : T = \{x_2 : B|t\} \qquad T' = \{x_3 : B|t \wedge s\}}{\Gamma \cap G \vdash e : T'}$$

$$\text{T-SUB}\ \frac{\Gamma \vdash e : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e : T}$$

$$\text{T-OP}\ \frac{\Gamma \vdash e_1 : S_1 \ldots \Gamma \vdash e_n : S_n \qquad \Gamma \vdash \mathtt{op} : x_1 : S_1 \rightarrow x_2 : S_2 \rightarrow \cdots \rightarrow x_n : S_n \rightarrow T}{\Gamma \vdash \mathtt{op}(e_1, \ldots, e_n) : T}$$

$$\text{T-IF}\ \frac{\Gamma \cap G \vdash e : \mathtt{Bool} \qquad \Gamma \vdash C \qquad \Gamma \vdash C'}{\Gamma \vdash (G)\mathtt{if}\ \ e\ \ \mathtt{then}\ \ C\ \ \mathtt{else}\ \ C'}$$

$$\text{T-ASN}\ \frac{\Gamma \cap G \vdash e : T \qquad \Gamma \cap G \vdash x : T \qquad \Gamma \vdash C}{\Gamma \vdash (G)x := e, C}$$

$$\text{T-STOP}\ \frac{}{\Gamma \vdash \mathtt{stop}}$$

Figure 3.7: Type rules.

$$\textsc{WT-Base}\ \frac{\Gamma, x : B \vdash t : \texttt{Bool}}{\Gamma \vdash \{x : B | t\}}$$

$$\textsc{S-Base}\ \frac{\Gamma, x : B \vdash s \Rightarrow t}{\Gamma \vdash \{x : B | s\} <: \{x : B | t\}}$$

$$\textsc{SA-Base}\ \frac{\Gamma, x : B \vdash^{a}_{alg} s \Rightarrow t \qquad a \in \{ok, \times, ?\}}{\Gamma \vdash^{a}_{alg} \{x : B | s\} <: \{x : B | t\}}$$

$$\textsc{Imp}\ \frac{\forall \sigma.(\Gamma \models \sigma \ \texttt{and}\ \sigma(s) \rightarrow^{*} \texttt{true implies}\ \sigma(t) \rightarrow^{*} \texttt{true})}{\Gamma \vdash s \Rightarrow t}$$

$$\textsc{CS-Empty}\ \frac{}{\emptyset \models \emptyset}$$

$$\textsc{CS-Ext}\ \frac{\emptyset \vdash t : T \qquad (x := t)\Gamma \models \sigma}{x : T, \Gamma \models (x := t, \sigma)}$$

Figure 3.8: Rules for consistent substitution and subtyping.

Cast Insertion. The next judgment we consider is that of *compiling with typecast insertion.* This judgment $\Gamma \vdash C \Rightarrow C'$ maps a command C to a command C'. Intuitively, this judgment adds casts to C precisely where algorithmic subtyping times out; these additional casts make C' well-typed. Figure 3.10 shows the defining rules for this judgment. For example, in the TI-OK-ASN-VAR rule, we first check that the identifier x has the type T in the type environment Γ, and that the variable x' has the type S. If the algorithmic subtyping judgment terminates and shows $\Gamma \vdash^{ok}_{alg} S <: T$, then no cast insertion is needed; the assignment maps to itself. Note that by our syntax, expressions are either simple values, single variables, or an operator; we do not allow compound expressions.

In contrast, if the algorithmic subtyping judgment times out, we have $\Gamma \vdash^{?}_{alg} S <: T$ and then the rule TI-CHK-ASN-VAR applies. The assignment is mapped to a new assignment with an assertion that x' is of type T that added to its guard set. Note that if the algorithmic subtyping judgment concludes that $\Gamma \vdash^{\times}_{alg} S <: T$, then we can conclude statically that the assignment fails type checking and reject the program outright. The rule for operators is similar, except that we also potentially add guards for operands, and a guard for a special variable `res` that holds the result of the operator evaluation.

Cast Evaluation. Finally, we discuss how casts are checked at run-time and the list of active checker expressions updated. In addition to the basic assignment rule for assignments with an empty guard set, evaluation includes a rule for *guarded assignment evaluation.*

To process a type cast for an expression e and a type T, we first we parse the type T as a refinement type $\{x : B|t\}$ to extract the Boolean expression t which defines the type. Next, we check $\Delta \vdash t[x := e] \rightarrow \texttt{true}$, which is the run-time membership check with the

$$\textsc{E-Stop}\ \frac{}{\langle (G)\texttt{stop}, \Delta, \Sigma, pc, ch\rangle \rightarrow \langle \Delta, \Sigma, pc, ch\rangle}$$

$$\textsc{E-IFTrue}\ \frac{\Delta, G \vdash e \qquad G, ch \vdash ch' \qquad \Delta \vdash e \rightarrow \ \texttt{true} \qquad \Sigma \vdash e \rightarrow sv_e \qquad pc' = pc \wedge sv_e}{\langle (G)\texttt{if}\ \ e\ \ \texttt{then}\ \ C\ \ \texttt{else}\ \ C', \Delta, \Sigma, pc, ch\rangle \rightarrow \langle C, \Delta', \Sigma', pc', ch'\rangle}$$

$$\textsc{E-IFFalse}\ \frac{\begin{array}{c}\Delta, G \vdash e \\ G, ch \vdash ch' \qquad \Delta \vdash e \rightarrow \ \texttt{false} \qquad \Sigma \vdash e \rightarrow sv_e \qquad pc' = pc \wedge \ \texttt{not}\ \ sv_e\end{array}}{\langle (G)\texttt{if}\ \ e\ \ \texttt{then}\ \ C\ \ \texttt{else}\ \ C', \Delta, \Sigma, pc, ch\rangle \rightarrow \langle C', \Delta', \Sigma', pc', ch'\rangle}$$

$$\textsc{E-Op}\ \frac{\forall i. \Delta e_i \vdash v_i \qquad z = \tilde{op}(v_1, \ldots, v_n)}{\Delta \vdash \texttt{op}(e_1, \ldots, e_n) \rightarrow z}$$

$$\textsc{E-CHK-Asn}\ \frac{\Delta, G \vdash e \qquad G, ch \vdash ch' \qquad \Delta \vdash e \rightarrow v \qquad \Delta' = \Delta(x \mapsto v)}{\langle (G)x := e, C', \Delta, \Sigma, pc, ch\rangle \rightarrow \langle C', \Delta', \Sigma', pc, ch'\rangle}$$

Figure 3.9: Evaluation rules. The E-IFTrue and E-IFFalse rules use the concrete state to specify how to update the path condition pc and check guards during the evaluation of the condition e. The E-CHK-Asn rule shows evaluation of type casts. During evaluation, if a cast $\langle T_i \rangle e$ fails, evaluation stops. Otherwise, each cast updates the set of checker constraints ch with a symbolic membership check for the type $\langle T_i \rangle$.

concrete program values. This is shown in the rule TC-Check for the judgment $\Delta, G \vdash e$. If this check fails, the E-CHK-Asn rule does not apply and the program evaluation will become stuck. On the other hand, if the check succeeds, then we use the rule TC-Upd to update the set of checkers ch with a symbolic type membership check to obtain ch'.

Example: Division by Zero. As an example, we show an active checker for division by zero errors. First, we define the refinement type `notzero` as the set of values x of type `Int` such that $x \neq 0$. Then, we can define the type of the division operator `div` as taking an `Int` and a `notzero` argument. Finally, we need a semantic implementation function for division,

$$\text{C-OK-A{\scriptsize SN}}\;\frac{\Gamma \cap G \vdash x : T \qquad \Gamma \cap G \vdash e : S \qquad \Gamma \vdash^{ok}_{alg} S <: T}{\Gamma \vdash \langle (G)x := e\rangle \Rightarrow \langle (G)x := e\rangle}$$

$$\text{C-CHK-A{\scriptsize SN}-V{\scriptsize AR}}\;\frac{\Gamma \cap G \vdash x : T \qquad \Gamma \cap G \vdash x' : S \qquad \Gamma \vdash^{?}_{alg} S <: T \qquad G' = G(x' \mapsto T);}{\Gamma \vdash \langle (G)x := x'\rangle \Rightarrow \langle (G')x := x'\rangle}$$

$$\text{C-CHK-A{\scriptsize SN}-O{\scriptsize P}}\;\frac{\begin{array}{c}\Gamma \cap G \vdash e_1 : S'_1 \ldots \Gamma \cap G \vdash e_n : S'_n \\ \Gamma \cap G \vdash \mathtt{op} : x_1 : S_1 \to \cdots \to x_n : S_n \to T \qquad \Gamma \vdash x : T' \\ \forall i \Gamma \vdash^{?}_{alg} S'_i <: S_i \qquad \Gamma \vdash^{?}_{alg} T <: T' \qquad G' = G(\mathtt{res} \mapsto T', o_1 \mapsto S_1, \ldots, o_n \mapsto S_n)\end{array}}{\langle (G)x := \mathtt{op}(e_1, \ldots, e_n)\rangle \Rightarrow \langle (G')x := \mathtt{op}(o_1, \ldots, o_n)\rangle}$$

Figure 3.10: Rules for compilation with type cast insertion. Casts are inserted on assignments, operator arguments, and on operator return values. The special variable `res` represents the result of an operator evaluation. While we show only the rules for when all arguments or no arguments require casts, we allow any subset of arguments to have casts inserted.

$\texttt{notzero} : \{x : \texttt{Int} | x \neq 0\}$

$\texttt{div} : \texttt{Int} \rightarrow \texttt{notzero} \rightarrow \texttt{Int}$

$\tilde{div}(n_1, n_2) = n_1/n_2$

Figure 3.11: Type and implementation for the division operator.

which here is the standard division function for integers. These are shown in Figure 3.11.

Now, in Figure 3.3 we show an example of active checking a small $\texttt{SimpL}^A$ program for division by zero errors. On the left, we see the original program before type-checking and compilation. Now we wish to perform static type checking and typecast insertion. Because `div` is an operator, at line 4) by the type rule T-Op, we must show that the type of v_1 is a subtype of `notzero` to prove that the program is well-typed. Therefore, during typecast insertion, we apply the rule TI-OK-OpAp, the rule TI-CHK-OpAp, or else statically reject this program as ill-typed. For the purposes of this example, we assume that the algorithmic implication judgment times out, and so we apply TI-CHK-OpAp to insert a cast to the type $\langle\texttt{notzero}\rangle$. The middle column shows the resulting code after cast insertion. Finally, suppose the code is run on the concrete input $\alpha = 2$. The column on the right shows *pc* and *ch* after each command is evaluated. After the first run, we then extract $\alpha \neq 0$ from *ch* and query $(\alpha > -5) \wedge \neg(\alpha \neq 0)$ to a constraint solver to actively check for an input that causes division by zero.

Example: Integer overflow. We can also perform active type checking for *integer overflow* in our framework. Integer overflow and related machine arithmetic errors are a frequent cause of security-critical bugs; the May 2007 MITRE survey of security bugs reported notes that integer overflows were the second most common bug in operating system vendor advi-

$$
\begin{array}{lcl}
\texttt{Z} & = & \{x : \texttt{Int} | -\infty < x < \infty\} \\
\texttt{uint64\_t} & = & \{x : \texttt{Int} | 0 \leq x < 2^{64}\} \\
\texttt{int64\_t} & = & \{x : \texttt{Int} | -2^{63} \leq x < 2^{63}\} \\
\texttt{uint32\_t} & = & \{x : \texttt{Int} | 0 \leq x < 2^{32}\} \\
\texttt{int32\_t} & = & \{x : \texttt{Int} | -2^{31} \leq x < 2^{31}\} \\
\texttt{uint16\_t} & = & \{x : \texttt{Int} | 0 \leq x < 2^{16}\} \\
\texttt{int16\_t} & = & \{x : \texttt{Int} | -2^{15} \leq x < 2^{15}\} \\
\texttt{uint8\_t} & = & \{x : \texttt{Int} | 0 \leq x < 2^{8}\} \\
\texttt{int8\_t} & = & \{x : \texttt{Int} | -2^{7} \leq x < 2^{7}\}
\end{array}
$$

Figure 3.12: Types for an integer overflow checker.

sories [23]. Brumley et al. describe a system of dynamic checks called RICH that catches machine arithmetic errors at run-time and abort program execution before the error can propagate further [11]. The basic idea of their approach is to define upper and lower bounds for signed and unsigned integer types, then insert a dynamic check whenever a potentially *unsafe assignment* is carried out, i.e. when a value outside the bounds of an integer type could be stored into a variable of that type.

We can capture the RICH checks with the refinement types in Figure 3.12. Given an annotation of variables with these types, the $\texttt{SimpL}^A$ type rules automatically insert casts for assignments between variables of different types. For example, suppose that a variable x has type `uint8_t` . Then an assignment of an expression e to x may insert a cast $\langle \texttt{uint8\_t} \rangle$ if the algorithmic implication judgment cannot decide statically whether e falls within the bounds of `uint8_t` or not. At run-time, the cast $\langle \texttt{uint8\_t} \rangle e$ checks that the concrete value of the expression e is between 0 and 2^8, just as in the RICH system. If the run-time check succeeds, then we symbolically evaluate e to obtain a symbolic expression sz_e, then add the expression $(0 \leq sz_e < 2^8)$ to the set of checkers ch. We can then query $pc \wedge \neg(0 \leq sz_e < 2^8)$ to a constraint solver and solve for an input that violates the type bounds.

Properties of Active Type Checking.

We now state several important properties of active type checking. We begin with basic type preservation and progress theorems.

Theorem 3 *(Type preservation for expressions.) Suppose that $\Gamma \vdash e : T$, $\Gamma \vdash \Delta$, and $\Delta \vdash e \rightarrow z$. Then $\Gamma \vdash z : T$.*

Suppose that $\Gamma \vdash C$, $\Delta \in \Gamma$, and $\langle C\Delta, \rangle \rightarrow \langle C', \Delta' \rangle$. Then $\Gamma \vdash C'$

Definition 3 *(Failed cast.) We say that an expression $\langle T \rangle e$ is a* failed cast *under Δ if $\Delta \vdash e \rightarrow z$ and $\emptyset \not\vdash z : T$. We say a command C* contains a failed cast *under Δ if there exists $\langle T_i \rangle e \in \texttt{guards}(C)$ such that $\langle T_i \rangle e$ is a failed cast under Δ.*

Theorem 4 *(Progress.) Suppose that $\Gamma \vdash C$, $\Delta \in \Sigma$, and $\Delta \in \Gamma$. Then either $\langle C, \Delta \rangle \rightarrow \langle C', \Delta' \rangle$, or C contains a failed cast under Δ.*

We then want to show that a program is always well-typed after compilation.

Theorem 5 *(Well-typed compilation.) Suppose that $\Gamma \vdash C \Rightarrow C'$. Then $\Gamma \vdash C'$.*

Proof 2 *By induction on the derivation. The main observation is that compilation adds casts precisely where the algorithmic subtyping judgment cannot decide subtyping.*

Next, we argue that unless a cast fails, compilation does not change how the concrete state of the program is updated on each command. This rules out the trivial compilation procedure $\Gamma C \Rightarrow C'$ that sets C' to `nil`.

Theorem 6 *(Semantic preservation.) Suppose $\Gamma \vdash C \Rightarrow C'$. For all finite k, if $\langle C, \Delta \rangle \rightarrow^k \langle C_2, \Delta' \rangle$, then either $\langle C', \Delta \rangle \rightarrow^k \langle C_2', \Delta' \rangle$, or there exists an i with $0 \leq i \leq k$ such that $\langle C', \Delta \rangle \rightarrow^i \langle C''', \Delta'' \rangle$ and C''' contains a failed cast under Δ''.*

Proof 3 *By induction on the derivation $\Gamma \vdash C \Rightarrow C'$. First prove that adding a single cast does not change the program's concrete semantics. Then induct on the number of casts added. The main observation is that compilation only adds casts but does not otherwise change the program, casts do not have side effects, and each non-failed evaluation step removes casts.*

Well-typed compilation combined with the progress theorem states a program after compilation can fail only at an inserted cast. We say that a program is *well-behaved* if no concrete inputs exist that cause the program to reach a command that contains a failed cast. The key point is that the set *ch* of checker expressions are sound and complete active checkers for the the property of being well-behaved. That is, if we find values $v_1, \ldots, v_n$ which, when substituted into the unknowns $\alpha_1, \ldots, \alpha_n$, cause the path condition to evaluate to `true` but a checker expression in *ch* to evaluate to `false`, then these values will cause a run-time check failure. First, we prove a lemma that states concrete values that satisfy a symbolic path condition will cause execution to follow the same path.

Lemma 1 *(Lockstep Lemma.) Suppose* $\Gamma \vdash C$*,* $\Delta_0 \in \Gamma$*, and* $\Sigma_0 \in \Gamma$*. For all* $k \geq 0$*, let* $\langle C, \Delta_0, \Sigma_0, pc_0, ch_0 \rangle \rightarrow^k \langle C_k, \Delta_k, \Sigma_k, pc_k, ch_k \rangle$*.*

Suppose the values $v_1, \ldots, v_n$ *satisfy* $\emptyset \vdash pc[\alpha_i = v_i] \rightarrow$ ***true****, that* $\Gamma \vdash \alpha_i : T_i$*, that* $\forall i.v_i \in T_i$*, and* $\Sigma_0[\alpha_i = v_i] \rightarrow \Delta'$*. For all* $0 \leq i \leq k$*, let* $\langle C, \Delta', \Sigma_0, pc_0, ch_0 \rangle \rightarrow \langle C'_i, \Delta'_i, \Sigma'_i, pc'_i, ch'_i \rangle$*. Then* $C_i = C'_i$ *and* $pc_i = pc'_i$ *or*C'_i *contains a failed cast under* Δ'_i*.*

Proof 4 *For each fixed* k*, we induct on* i*. The non-trivial inductive cases are* E-IFTrue *and* E-IFFalse*, as no other rules update pc or cause control flow change. We argue that because the values* v_i *satisfy* pc_k*, and because each* pc_i *is a prefix of* pc_k*, then the values* v_i *must also satisfy* pc_i *for* $0 \leq i \leq k$*. Because each* pc_i *is a conjunction of* ***if*** *conditions, we therefore have for an if statement with guard* b_i *that if* $\Delta_i \vdash b_i \rightarrow v$ *then* $\Delta'_i \vdash b_i \rightarrow v$*. This causes control flow and the program counter to be updated as desired.*

Theorem 7 *(Soundness and Completeness of Active Type Checking.) Suppose* $\Gamma \vdash C$*,* $\Delta_0 \in$ *Gamma, and* $\Sigma_0 \in \Gamma$*. For all* $k \geq 0$*, suppose* $\langle C, \Delta_0, \Sigma_0, pc_0, ch_0 \rangle^k \rightarrow \langle C_k, \Delta_k, \Sigma_k, pc_k, ch_k \rangle$*.*

Suppose the values $v_1, \ldots, v_n$ *satisfy* $\emptyset \vdash pc[\alpha_i = v_i] \rightarrow$ ***true****, that* $\Gamma \vdash \alpha_i : T_i$*, that* $\forall i.v_i \in T_i$*, and* $\Sigma[\alpha_i = v_i] \rightarrow \Delta'$*.*

Then there exists an expression $e \in (ch_i - ch_{i-1})$ *such that* $\emptyset \vdash \neg e[\alpha_i = v_i] \rightarrow$ ***true*** *if and only if for some* $i \leq k$*, we have* $\langle C, \Delta', \Sigma, pc_0, ch_0 \rangle \rightarrow^i \langle C', \Sigma'', \Delta'', pc', ch'' \rangle$ *and* C' *contains a failed cast under* Δ''*.*

Note that we allow for the new values to cause a cast failure at any program state before the k'th evaluation. We need this because the execution may potentially fail a cast before reaching the k'th command.

Finally, we turn our attention to the question of *combining* checks for different properties by adding additional type annotations to our type environment. For example, we might want to simultaneously check for division by zero and integer overflow. We can think of the combination as follows: we start with a type environment Γ_{div} that contains only the type of the division operator. Next, we add type annotations to check for integer overflow to Γ_{div} to obtain $\Gamma_{div,int}$. This type environment contains enough annotations and types to simultaneously check both properties.

We then want *monotonicity* of combination. That is, suppose $\Gamma_{div} C \Rightarrow C'$, and $\Gamma_{div,int} C \Rightarrow C''$. Then if C' contains a failed cast under some concrete store Δ, we would like C'' to also fail a cast under the same concrete store Δ. Put another way, adding properties should be *conservative* in the set of programs rejected. Note that C'' may not fail at the same cast as C', because the same concrete values may trigger multiple errors. Checking additional properties in may mean catching an error earlier in execution, and so C'' might not progress to the same cast where C' fails.

The following theorem states a condition for augmenting a type environment Γ to obtain a new type environment Γ' in a "consistent" way so this monotonicity property holds. Specifically, if this condition is met, then the casts inserted by the compilation judgment with Γ' will reject at least as many programs for run-time errors as the casts inserted by compiling with Γ.

Theorem 8 *(Conservation Under Additional Checks.) We say that $\Gamma <: \Gamma'$ if the following two conditions hold :*

1. *for all variables v_i such that $\Gamma \vdash v_i S_i$, and $\Gamma' \vdash S_i'$, we have $\Gamma' \vdash S_i <: S_i'$*

2. *for all operators* $\texttt{op}_i$ $\Gamma \vdash \texttt{op}_i x_1 : T_1 \rightarrow \cdots \rightarrow x_n : T_n \rightarrow T$ *and* $\Gamma \vdash \texttt{op}_i x_1 : T'_1 \rightarrow \cdots \rightarrow x_n : T'_n \rightarrow T$*, we have* $\Gamma' \vdash T_i <: T'_i$*.*

Suppose $\Gamma <: \Gamma'$*. Then the following properties hold:*

1. *If* $\Gamma \vdash C$ *and* $\Gamma \vdash \Delta$*, then* $\Gamma' \vdash C$ *and* $\Gamma' \vdash \Delta$*.*

2. *Suppose* $\Gamma \vdash C \Rightarrow C'$ *and* $\Gamma' \vdash C \Rightarrow C''$*. For all* $k \geq 0$*, for all* $\Delta_0 \in \Gamma$*, for* $\langle C', \Delta_0 \rangle \rightarrow^k \langle C'_k, \Delta'_k$*, then if* C'_k *contains a failed cast under* Δ'_k*, then there exists an* i*, with* $0 \leq i \leq k$ *such that* $\langle C'', \Delta_0 \rangle \rightarrow \langle C''_i, \Delta''_i \rangle$ *and* C''_i *contains a failed cast under* Δ''_i*.*

3.5 Optimizations

3.5.1 Minimizing Calls to the Constraint Solver

As discussed in Section 3.3, (negations of) constraints injected by various active checkers in a same path constraint can be solved independently one-by-one since they have no side effects. We call this a *naive combination* of checker constraints.

However, the number of calls to the constraint solver can be reduced by bundling together constraints injected at the same or equivalent program states into a single conjunction. If pc denotes the path constraint for a given program state, and $\phi_{C1}, \ldots, \phi_{Cn}$ are a set of constraints injected in that state by each of the active checkers, we can define the combination of these active checkers by injecting the formula $\phi_C = \phi_{C1} \wedge \cdots \wedge \phi_{Cn}$ in the path constraint, which will result in the single query $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$ to the constraint solver. We can also bundle in the same conjunction constraints ϕ_{Ci} injected by

```
Procedure CombineActiveCheckers($I, pc, \phi_{C1}, \ldots, \phi_{Cn}$):
1. Let $x =$ Solve($pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$)
2. If $x =$ UNSAT return $I$
3. For all $i$ in $[1, n]$, eliminate $\phi_{Ci}$ if $x$ satisfies $\neg\phi_{Ci}$
4. Let $\phi_{C1}, \ldots, \phi_{Cm}$ denote the remaining $\phi_{Ci}$ ($m < n$)
5. If $m = 0$, return $I \cup \{x\}$
6. Call CombineActiveCheckers($I \cup \{x\}, pc, \phi_{C1}, \ldots, \phi_{Cm}$)
```

Figure 3.13: Function to compute a strongly-sound combination of active checkers.

active checkers at different program states anywhere in between two conditional statements, i.e., anywhere between two constraints in the path constraint (since those program states are indistinguishable by that path constraint). This combination reduces the number of calls to the constraint solver but, if the query $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$ is satisfied, a satisfying assignment produced by the constraint solver may not satisfy *all* the disjuncts, i.e., may violate only *some* of the properties being checked. Hence, we call this a *weakly-sound combination.*

A *strongly-sound*, or *sound* for short, combination can be obtained by making additional calls to the constraint solver using the simple function shown in Figure 3.13. By calling

$$\texttt{CombineActiveCheckers}(\emptyset, pc, \phi_{C1}, \ldots, \phi_{Cn})$$

the function returns a set I of input values that covers *all* the disjuncts that are satisfiable in the formula $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$. The function first queries the solver with the disjunction of all the checker constraints (line 1). If the solver returns `UNSAT`, we know that all these constraints are unsatisfiable (line 2). Otherwise, we check the solution x returned by the constraint solver against each checker constraint to determine which are satisfied by solution x (line 3). (This is a model-checking check, not a satisfiability check; in practice, this can be implemented by calling the constraint solver with the formula $\bigwedge_i (b_i \Leftrightarrow \neg\phi_{Ci}) \wedge pc \wedge (\bigvee_i b_i)$ where b_i is a fresh boolean variable which evaluates to `true` iff $\neg\phi_{Ci}$ is satisfied by a satisfying assignment x returned by the constraint solver; determining which checker constraints are satisfied by x can then be done by looking up the values of the corresponding bits b_i in solution x.) Then, we remove these checker constraints from the disjunction (line 4), and query the solver again until all checker constraints that can be satisfied have been satisfied by some input value in I. If t out of the n checkers can be satisfied in conjunction with the path constraint pc, this function requires at most $\texttt{min}(t+1, n)$ calls to the constraint solver, because each call removes at least one checker from consideration. Obtaining strong soundness with fewer than t calls to the constraint solver is not possible in the worse case. Note that the naive combination defined above is strongly-sound, but always requires n calls to the constraint solver.

It is worth emphasizing that none of these combination strategies attempt to minimize the number of input values (solutions) needed to cover all the satisfiable disjuncts. This could be done by querying first the constraint solver with the *conjunction* of all checker constraints to check whether any solution satisfies all these constraints simultaneously, i.e., to check whether their intersection is non-empty. Otherwise, one could then iteratively query the solver with smaller and smaller conjunctions to force the solver to return a minimum set of satisfying assignments that cover all the checker constraints. Unfortunately, this procedure may require in the worse case $O(2^n)$ calls to the constraint solver. (The problem can be shown to be NP-complete by a reduction from the NP-hard SET-COVER problem.)

Weakly and strongly sound combinations capture possible overlaps, inconsistencies or redundancies between active checkers at equivalent program states, but is independent of how each checker is specified: it can be applied to any active checker that injects a formula at a given program state. Also, the above definition is independent of the specific reasoning capability of the constraint solver. In particular, the constraint solver may or may not be able to reason precisely about combined theories (abstract domains and decision procedures) obtained by combining individual constraints injected by different active checkers. Any level of precision is acceptable with our framework and is an orthogonal issue (e.g., see [45]). Whether the constraint solver supports incremental solving is also orthogonal.

3.5.2 Minimizing Formulas

Minimizing the number of calls to the constraint solver should not be done at the expense of using longer formulas. Fortunately, the above strategies for combining constraints injected by active checkers also reduce formula sizes.

For instance, consider a path constraint pc and a set of n constraints $\phi_{C1} \ldots \phi_{Cn}$ to be injected at the end of pc. The naive combination makes n calls to the constraint solver, each with a formula of length $|pc| + |\phi_{Ci}|$, for all $1 \leq i \leq n$. In contrast, the weak combination makes only a single call to the constraint solver with a formula of size

$|pc| + \Sigma_{1 \leq i \leq n} |\phi_{Ci}|$, i.e., a formula (typically much) smaller than the sum of the formula sizes with the naive combination. The strong combination makes, in the worse case, n calls to the constraint solver with formulas of size $|pc| + \Sigma_{1 \leq i \leq j} |\phi_{Ci}|$ for all $1 \leq j \leq n$, i.e., possibly bigger formulas than the naive combination. But often, the strong combination makes fewer calls than the naive combination, and matches the weak combination in the best case (when none of the disjuncts $\neg\phi_{Ci}$ are satisfiable).

In practice, path constraints pc tend to be long, much longer than injected constraints ϕ_{Ci}. A simple optimization (implemented in [40, 17, 81, 43]) consists of eliminating the constraints in pc which do not share symbolic variables (including by transitivity) with the negated constraint c to be satisfied. This *unrelated constraint elimination* can be done syntactically by constructing an undirected graph G with one node per constraint in $pc \cup \{c\}$ and one node per symbolic (input) variable such that there is an edge between a constraint and a variable iff the variable appears in the constraint. Then, starting from the node corresponding to constraint c, one performs a (linear-time) traversal of the graph to determine with constraints c' in pc are reachable from c in G. At the end of the traversal, only the constraints c' that have been visited are kept in the conjunction sent to the constraint solver, while the others are eliminated.

With unrelated constraint elimination and the naive checker combination, the size of the reduced path constraint pc_i may vary when computed starting from each of the n constraints ϕ_{Ci} injected by the active checkers. In this case, n calls to the constraint solver are made with the formulas $pc_i \wedge \neg\phi_{Ci}$, for all $1 \leq i \leq n$. In contrast, the weak combination makes a single call to the constraint solver with the formula $pc' \wedge (\vee_i \neg\phi_{Ci})$ where pc' denotes

```
#define k 100 // constant
void Q(int *x, int a[k]){ // inputs
  int tmp1,tmp2,i;
  if (x == NULL) return;
  for (i=0; i<=k;i++) {
    if (a[i]>0) tmp1 = tmp2+*x;
    else tmp2 = tmp1+*x;
  }
  return;
}
```

Figure 3.14: A program with $O(2^k)$ possible execution paths. A naive application of a NULL dereference active checker results in $O(k \cdot 2^k)$ additional calls to the constraint solver, while local constraint caching eliminates the need for any additional calls to the constraint solver.

the reduced path constraint computed when starting with the constraint $\vee_i \neg\phi_{Ci}$. It is easy to see that $|pc'| \leq \Sigma_i |pc_i|$, and therefore that the formula used with the weak combination is again smaller than the sum of the formula sizes used with the naive combination. Loosely speaking, the strong combination includes again both the naive and weak combinations as two possible extremes.

3.5.3 Caching Strategies

No matter what strategy is used for combining checkers at a single program point, constraint *caching* can significantly reduce the overhead of using active checkers.

To illustrate the benefits of constraint caching, consider a NULL dereference active checker (see Section 3.3) and the program Q in Figure 3.14. Program Q has 2^k+1 executions, where 2^k of those dereference the input pointer `x` k times each. A naive approach to dynamic test generation with a NULL dereference active checker would inject k constraints of the form `x` $\neq$ `NULL` at each dereference of `*x` during every such execution of Q, which would result in a total of $k \cdot 2^k$ additional calls to the constraint solver (i.e., k calls for each of those executions).

To limit this expensive number of calls to the constraint solver, a first optimization consists of *locally caching* constraints in the current path constraint in such a way that syntactically identical constraints are never injected more than once in any path constraint. (Remember path constraints are simply conjunctions.) This optimization is applicable to any path constraint, with or without active checkers. The correctness of this optimization is based on the following observation: if a constraint c is added to a path constraint pc, then for any longer pc' extending pc, we have $pc' \Rightarrow pc$ (where $\Rightarrow$ denotes logical implication) and $pc' \wedge \neg c$ will always be unsatisfiable because c is in pc'. In other words, adding the same constraint multiple times in a path constraint is pointless since only the negation of its first occurrence has a chance to be satisfiable.

Constraints generated by active checkers can be dealt with by injecting those in the path constraint like regular constraints. Indeed, for any constraint c injected by an active checker either at the end of a path constraint pc or at the end of a longer path constraint pc' (i.e., such that $pc' \Rightarrow pc$), we have the following:

- if $pc \wedge \neg c$ is unsatisfiable, then $pc' \wedge \neg c$ is unsatisfiable;
- conversely, if $pc' \wedge \neg c$ is satisfiable, then $pc \wedge \neg c$ is satisfiable (and has the same solution).

Therefore, we can check $\neg c$ as early as possible, i.e., in conjunction with the shorter pc, by inserting the first occurrence of c in the path constraint. If an active checker injects the same constraint later in the path constraint, local caching will simply remove this second redundant occurrence.

By injecting constraints generated by active checkers into regular path constraints and by using local caching, a given constraint c, like `x` $\neq$ `NULL` in our previous example, will appear at most once in each path constraint, and a single call to the constraint solver will be made to check its satisfiability for each path, instead of k calls as with the naive approach without local caching. Moreover, because the constraint `x` $\neq$ `NULL` already appears in the path constraint due to the `if` statement on line 4 before any pointer dereference `*x` on lines 6 or 7, it will never be added again to the path constraint with local caching, and no additional calls will be made to the constraint solver due to the NULL pointer dereference active checker for this example.

Another optimization consists of caching constraints *globally* [17]: whenever the constraint solver is called with a query, this query and its result are kept in a (hash) table shared between execution paths during a directed search. In Section 3.7, the effect of both local and global caching is measured empirically.

3.6 Implementation

We implemented active checkers as part of a dynamic test generation tool called SAGE (*Scalable, Automated, Guided Execution*) [43]. SAGE uses the `iDNA` tool [7] to trace executions of Windows programs, then virtually re-executes these traces with the `TruScan` trace replay framework [73]. During re-execution, SAGE checks for file read operations and marks the resulting bytes as symbolic. As re-execution progresses, SAGE generates symbolic constraints for the path constraint. After re-execution completes, SAGE uses the constraint solver `Disolver` [49] to generate new input values that will drive the program

Number	Checker	Number	Checker
0	Path Exploration	7	Integer Underflow
1	DivByZero	8	Integer Overflow
2	NULL Deref	9	MOVSX Underflow
3	SAL NotNull	10	MOVSX Overflow
4	Array Underflow	11	Stack Smash
5	Array Overflow	12	AllocArg Underflow
6	REP Range	13	AllocArg Overflow

Figure 3.15: Active checkers implemented.

down new paths. SAGE then completes this cycle by testing and tracing the program on the newly generated inputs. The new execution traces obtained from those new inputs are sorted by the number of new code blocks they discover, and the highest ranked trace is expanded next to generate new test inputs and repeat the cycle [43]. While we describe in the technical report version of this paper how to combine static analysis with active property checking, the current SAGE implementation does not perform any static analysis [42].[1]

The experiments reported in the next section were performed with 13 active checkers, shown in Figure 3.15 with an identifying number. The number 0 refers to a constraint generated by observing a branch on tainted data, as in basic DART or EXE, that becomes part of the path constraint. Number 1 refers to a division-by-zero checker, 2 denotes a NULL pointer dereference checker, and 4 and 5 denote array underflow and overflow checkers (see

[1]Static analysis of the codec applications discussed next is problematic due to function pointers and in-line assembly code.

Section 3.3). Number 3 refers to an active checker that looks for function arguments that have been annotated with the `notnull` attribute in the SAL property language [47], and attempts to force those to be NULL. Checker type 6 looks for the x86 `REP MOVS` instruction, which copies a range of bytes to a different range of bytes, and attempts to force a condition where the ranges overlap, causing unpredictable behavior. Checkers 7 and 8 are for integer underflows and overflows. Checkers type 9 and 10 target the `MOVSX` instruction, which sign-extends its argument and may lead to loading a very large value if the argument is negative. The “stack smash” checker, type 11, attempts to solve for an input that directly overwrites the stack return pointer, given a pointer dereference that depends on a symbolic input. Finally, checkers type 12 and 13 look for heap allocation functions with symbolic arguments; if found, they attempt to cause overflow or underflow of these arguments.

An active checker in SAGE first registers a `TruScan` callback for specific events that occur during re-execution. For example, an active checker can register a callback that fires each time a symbolic input is used as an address for a memory operation. The callback then inspects the concrete and symbolic state of the re-execution and decides whether or not to emit an active checker constraint. If the callback does emit such a constraint, SAGE stores it in the current path constraint.

SAGE implements a *generational search* [43]: given a path constraint, all the constraints in that path are systematically negated one-by-one, placed in a conjunction with the prefix of the path constraint leading it, and attempted to be solved with the constraint solver. Constraints injected by active checkers are inserted in the path constraint and treated as regular constraints during a generational search.

Because we work with x86 machine-code traces, some information we would like to use as part of our active checkers is not immediately available. For example, when SAGE observes a `load` instruction with a symbolic offset during re-execution, it is not clear what the bound should be for the offset. We work around these limitations by leveraging the `TruScan` re-execution infrastructure. During re-execution, `TruScan` observes calls to known allocator functions. By parsing the arguments to these calls and their return values, as well as detecting the current stack frame, `TruScan` builds a map from each concrete memory address to the bounds of the containing memory object. We use the bounds associated with the memory object pointed to by the concrete value of the address as the upper and lower bound for an active bounds check of the memory access. Therefore, source code or debug symbols are not required, although they may be used if available.

3.7 Evaluation

We report results of experiments with active checkers and two applications which play media files and are widely used on Windows. Figure 3.16 shows the result of a single symbolic execution and test generation task for each of these two test programs. The second column indicates which checkers injected constraints during that program execution. The last column gives the number of symbolic input bytes read during that single execution, which is 100 to 1,000 times larger than previously reported with dynamic test generation [40, 17, 81].

For each application, we ran microbenchmarks to quantify the marginal cost of active checking during a single symbolic execution task and measure the effectiveness of our

Test	Checkers Injected	Time (secs)	*pc* size	# checker constraints	# Tests
Media 1	0,2,4,5,7,8	37	67	46	105
Media 2	0,1,2,4,5,7,8,9,10,11	1226	2990	6080	5122

Test	# Instructions	Symbolic Input Size
Media 1	3795771	65536
Media 2	279478553	27335

Figure 3.16: Statistics from a *single* symbolic execution and test generation task with a naive combination of all 13 checkers. We report the checker types that injected constraints, the total time for symbolic execution test generation, the number of constraints in the total path constraint, the total number of injected checker constraints, the number of tests generated, the number of instructions executed after the first file read, and the number of symbolic input bytes.

optimizations. We then performed long-running searches with active checkers to investigate their effectiveness at finding bugs. These searches were performed on a 32-bit Windows Vista machine with two dual-core AMD Opteron 270 processors running at 2 GHz, with 4 GB of RAM and a 230 GB hard drive; all four cores were used in each search. We now describe observations from these experiments. We stress that these observations are from a limited sample size and should be taken with caution.

3.7.1 Microbenchmarks

Figure 3.17 presents statistics for our two test programs obtained with a single symbolic execution and test generation task with no active checkers, or the weak, strong and naive combinations of active checkers discussed in Section 3.5. For each run, we report the total run time (in seconds), the time spent in the constraint solver (in seconds), the

Media 1	none	weak	strong	naive
Total Time (s)	16	37	42	37
Solver Time (s)	5	5	10	5
# Tests Gen	59	70	87	105
# Disjunctions	N/A	11	11	N/A
Dis. Min/Mean/Max	N/A	2/4.2/16	2/4.2/16	N/A
# Path Constr.	67	67	67	67
# Checker Constr.	N/A	46	46	46
# Solver Calls	67	78	96	113
Max CtrList Size	77	141	141	141
Mean CtrList Size	2.7	2.7	2.7	3
Local Cache Hit	79%	81%	88%	88%

Media 2	none	weak	strong	naive
Total Time (s)	761	973	1140	1226
Solver Time (s)	421	463	601	504
# Tests Gen	1117	1833	2734	5122
# Disjunctions	N/A	1125	1125	N/A
Dis. Min/Mean/Max	N/A	1/5.4/216	1/5.4/216	N/A
# Path Constr.	3001	2990	2990	2990
# Checker Constr.	N/A	6080	6080	6080
# Solver Calls	3001	4115	5368	9070
Max CtrList Size	11141	91739	91739	91739
Mean CtrList Size	368	373	373	372
Local Cache Hit	39%	19.5%	19.5%	19.5%

Figure 3.17: Microbenchmark statistics.

number of test generated, the number of disjunctions bundling together checker constraints (if applicable) before calling the constraint solver, the minimum, mean and maximum number of constraints in disjunctions (if applicable[2]), the total number of constraints in the path constraint, the total number of constraints injected by checkers, the number of calls made to the constraint solver, statistics about the size needed to represent all path and checker constraints (discussed further below) and the local cache hit. Each call to the constraint solver was set with a timeout value of 5 seconds, which we picked because almost all queries we observed terminated within this time.

Checkers produce more test cases than path exploration at a reasonable cost. As expected, using checkers increases total run time but also generates more tests. For example, all checkers with naive combination for Media 2 creates 5122 test cases in 1226 seconds, compared to 1117 test cases in 761 seconds for the case of no active checkers; this gives us 4.5 times as many test cases for 61% more time spent in this case. As expected (see Section 3.5), the naive combination generates more tests than the strong combination, which itself generates more tests than the weak combination. Perhaps surprisingly, most of the extra time is spent in symbolic execution, not in solving constraints. This may explain why the differences in runtime between the naive, strong and weak cases are relatively not that significant. Out of curiosity, we also ran experiments (not shown here) with a "basic" set of checkers that consisted only of Array Bounds and DivByZero active checkers; this produced fewer test cases, but had little to no runtime penalty for test generation for both test programs.

[2]In the strong case, the mean number does not include disjunctions iteratively produced by the algorithm of Figure 3.13, which explains why the mean is the same as in the weak case.

Weak combination has the lowest overhead. We observed that the solver time for weak combination of disjunctions was the lowest for Media 2 runs with active checkers and tied for lowest with the naive combination for Media 1. The strong disjunction generates more test cases, but surprisingly takes longer than the naive combination in both cases. For Media 1, this is due to the strong combination hitting one more 5-second timeout constraints than the naive combination. For Media 2, we believe this is due to the overhead involved in constructing repeated disjunction queries. Because disjunctions in both cases have fairly few disjuncts on average (4 or 5), this overhead dominates for the strong combination, while the weak one is still able to make progress by handling the entire disjunction in one query.

For the SAGE system, the solver used requires less than a second for most queries. Therefore the time improvement for weak combination over naive combination in this microbenchmark is small, saving roughly 3 minutes, even though the number of calls to the solver is in fact reduced by almost 50%. Other implementations may obtain better time performance, especially if they more precisely model memory or include more difficult constraints. Both figures include the effect of other optimizations, such as caching, which we show are important for active checking.

Unrelated constraint elimination is important for checkers. Our implementation of the unrelated constraint optimization described in Section 5.2 introduces additional *common subexpression variables*. Each of these variable defines a subexpression that appears in more than one constraint. In the worst case, the maximum possible size of a list of constraints passed to our constraint solver is the sum of the number of these variables, plus the size of the path constraint, plus the number of checker constraints injected. We collected the maximum possible constraint list size (Max CtrList Size) and the mean size of constraint lists produced after our unrelated constraint optimization (Mean CtrList Size). The maximum possible size does not depend on our choice of weak, strong, or naive combination, but the mean list size is slightly affected. We observe in the Media 2 microbenchmarks that the maximum possible size jumps dramatically with the addition of checkers, but that the mean size stays almost the same. Furthermore, even in the case without checkers, the mean list size is 100 times smaller than the maximum. The Media 1 case was less dramatic, but still showed post-optimization constraint lists an order of magnitude smaller than the maximum. This shows that unrelated constraint optimization is key to efficiently implement active checkers.

3.7.2 Macrobenchmarks

For macrobenchmarks, we ran a generational search for 10 hours starting from the same initial media file, and generated test cases with no checkers, and with the weak and strong combination of all 13 checkers. We then tested each test case by running the program with AppVerifier [22], configured to check for heap errors. For each crashing test case, we recorded the checker kinds responsible for the constraints that generated the test. Since a search can generate many different test cases that exhibit the same bug, we "bucket" crashing files by the *stack hash* of the crash, which includes the address of the faulting instruction. We also report a bucket kind, which is either a NULL pointer dereference, a read access violation (ReadAV), or a write access violation (WriteAV). It is possible for the same bug to be reachable by program paths with different stack hashes for the same root cause. Our experiments always report the distinct stack hashes. We also computed the hit rate for global caching during each search.

Crash Bucket	Kind	0	2	4	5	7	8
1867196225	NULL	No/W/S				W/S	W/S
1867196225	ReadAV	No/W				W	W
1277839407	ReadAV	S					
1061959981	ReadAV		S			S	S
1392730167	ReadAV	S					
1212954973	ReadAV				S	S	S
1246509355	ReadAV				S	W/S	W/S
1527393075	ReadAV	S					
1011628381	ReadAV					S	W/S
2031962117	ReadAV	No/W/S					
286861377	ReadAV	No/S					
842674295	WriteAV		S	S		S	S

Figure 3.18: Crash buckets found for Media 1 by 10-hour searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. A total of 41658 tests were generated and tested in 30 hours, with 783 crashing files in 12 buckets.

Crash Bucket	Kind	0
790577684	ReadAV	No/W/S
825233195	ReadAV	No/W/S
795945252	ReadAV	No/W/S
1060863579	ReadAV	No/W

Figure 3.19: Crash buckets found for Media 2 by 10-hours searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. A total of 11849 tests were generated and tested in 30 hours, with 25 crashing files in 4 buckets.

Media 1	0	2	4	5	7	8
Injected	27612	26	13	13	11153	11153
Solved	18056	22	2	3	3179	5552
Crashes	339	22	2	3	139	136
Yield	1.9%	100%	100%	100%	4.4%	2.4%

Figure 3.20: Constraints injected by checker types, solved, crashes, and yield for Media 1, over both weak and strong combination for 10-hour searches.

Checkers can find bugs missed by path exploration. Figure 3.18 shows the crash buckets found for Media 2 by 10-hours searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. For instance, an "S" in a column means that at least one crash in the bucket was found by the search with strong combination. The type of checkers whose constraint found the crash bucket is also indicated in the figure. For Media 1, the Null Deref (type 2) active checker found 2 crash buckets, the Array Underflow and Overflow (types 4 and 5) active checkers found 3 crash buckets, while the Integer Underflow and Overflow (types 7 and 8) active checkers found 7 crash buckets. Without any active checkers, the tool is able to find only 4 crash buckets in 10 hours of search, and misses the serious WriteAV bug detected by the strong combination only. For Media 2, in contrast, the test cases generated by active checkers did not find any new crash buckets, as shown in Figure 3.19. The Media 1 case shows on some programs, checkers are effective at finding several important bugs missed by path exploration alone, but the Media 2 example shows this does not hold for all programs.

Checker yield can vary widely. Figures 3.20and 3.21 report the overall number of

Media 2	0	1	2	4	5
Injected	13425	12	2146	1544	1551
Solved	4735	0	61	10	20
Crashes	7	0	0	0	0
Yield	1.4%	N/A	0%	0%	0%
	7	8	9	10	11
Injected	11158	11177	5	5	10
Solved	576	2355	0	5	0
Crashes	0	0	0	0	0
Yield	0%	0%	N/A	0%	N/A

Figure 3.21: Constraints injected by checker types, solved, crashes, and yield for Media 2, over both weak and strong combination for 10-hour searches.

injected constraints of each type during all 10-hours searches, and how many of those were successfully solved to create new test cases. It also reports the checker *yield*, or percentage of test cases that led to crashes. For Media 1, active checkers have a higher yield than test cases generated by path exploration (type 0). For Media 2, several checkers did inject constraints that were solvable, but their yield is 0% as they did not find any new bugs. The yield indicates how precise symbolic execution is. For Media 1, symbolic execution is very precise as every checker constraint violation for checker types 2, 4 and 5 actually leads to a crash (as is the case with a fully sound and complete constraint generation and solving as shown in Section 3.3); even if symbolic execution is perfect, the yield for the integer under/overflow active checkers may be less than 100% because not every integer under/overflow leads to a crash. In contrast, symbolic execution in our current implementation does not seem precise enough for Media 2, as yields are poor for this benchmark.

Local and global caching are effective. Local caching can remove a significant number of constraints during symbolic execution. For Media 1, we observed a 80% or more local cache hit rate (see Figure 3.17). For Media 2, the hit rates were less impressive but still removed roughly 20% of the constraints.

Our current implementation does not have a global cache for query results (see Section 3.5). To measure the impact of global caching on our macrobenchmark runs, we added code that dumps to disk the SHA-1 hash of each query to the constraint solver, and then computes the global cache hit rate. For Media 1, all searches showed roughly a 93% hit rate, while for Media 2 we observed 27%. This shows that there are significant redundancies in queries made by different test generation tasks during the same search.

3.8 Conclusion

The more one checks for property violations, the more one should find software errors. We have defined and studied active property checking, a new form of dynamic property checking based on dynamic symbolic execution, constraint solving and test generation. We showed how active type checking extends traditional static and dynamic type checking. We presented several optimizations to implement active property checkers efficiently, and discussed results of experiments with several large shipped Windows applications. Active property checking was able to detect several new bugs in those applications. While our results are for x86 binaries, our techniques apply to any architecture and could be used for testing of other systems.

Overall, we showed that without careful optimizations, active property checking can significantly slow down dynamic test generation. For the SAGE implementation, we found caching and unrelated constraint optimization to be the most important. While weak and strong combination did in fact reduce the number of calls to the constraint solver, this was not the bottleneck. Our results for Media 1 demonstrate that active property checking can find a significant number of bugs not discovered by path exploration alone. At the same time, the results for Media 2 show that the current SAGE constraints do not fully capture the execution of some programs. Increasing constraint precision should lead to more expensive solver calls, therefore making our weak and strong combinations more important.

We have also performed exploratory searches on several other applications, including two shipped as part of Office 2007 and two media parsing layers. In one of the Office applications and media layer, the division by zero checker and the integer overflow checker each created test cases leading to previously-unknown division by zero errors. In the other cases, we also discovered new bugs in test cases created by checkers, but needed to use an internal tool for runtime passive checking of memory safety violations that is more precise than AppVerifier. Overall, active property checking, with our optimizations, is an effective and scalable technique for discovering many serious security-critical bugs.

Chapter 4

Integer Bugs, Linux Programs, and Reporting at Scale

4.1 Introduction

Integer overflow bugs recently became the second most common bug type in security advisories from OS vendors [23]. Unfortunately, traditional static and dynamic analysis techniques are poorly suited to detecting integer-related bugs. In this paper, we argue that *dynamic test generation* is better suited to finding such bugs, and we develop new methods for finding a broad class of integer bugs with this approach. We have implemented these methods in a new tool, *SmartFuzz*, that analyzes traces from commodity Linux x86 programs.

Integer bugs result from a mismatch between machine arithmetic and mathematical arithmetic. For example, machine arithmetic has bounded precision; if an expression has a value greater than the maximum integer that can be represented, the value wraps around to fit in machine precision. This can cause the value stored to be smaller than expected by the programmer. If, for example, a wrapped value is used as an argument to `malloc`, the result is an object that is smaller than expected, which can lead to a buffer overflow later if the programmer is not careful. This kind of bug is often known as an integer overflow bug. In Section 4.2 we describe two other classes of integer bugs: *width conversions*, in which converting from one type of machine integer to another causes unexpected changes in value, and *signed/unsigned conversions*, in which a value is treated as both a signed and an unsigned integer. These kinds of bugs are pervasive and can, in many cases, cause serious security vulnerabilities. Therefore, eliminating such bugs is important for improving software security.

While new code can partially or totally avoid integer bugs if it is constructed appropriately [61], it is also important to find and fix bugs in legacy code. Previous approaches to finding integer bugs in legacy code have focused on static analysis or runtime checks. Unfortunately, existing static analysis algorithms for finding integer bugs tend to generate many false positives, because it is difficult to statically reason about integer values with sufficient precision. Alternatively, one can insert runtime checks into the application to check for overflow or non-value-preserving width conversions, and raise an exception if they occur. One problem with this approach is that many overflows are benign and harmless. Throwing an exception in such cases prevents the application from functioning and thus causes false positives. Furthermore, occasionally the code intentionally relies upon overflow semantics; e.g., cryptographic code or fast hash functions. Such code is often falsely flagged by static analysis or runtime checks. In summary, both static analysis and runtime checking tend to suffer from either many false positives or many missed bugs.

In contrast, *dynamic test generation*, as described in this thesis, is a promising approach for avoiding these shortcomings. Our main approach is to use symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. Then, we run the program on these test cases and use standard tools that check for buggy behavior to recognize bugs. We only report test cases that are verified to trigger incorrect behavior by the program. As a result, we have confidence that all test cases we report are real bugs and not false positives.

Others have previously reported on using dynamic test generation to find some kinds of security bugs [17]. The contribution of this chapter is to show how to extend those techniques to find integer-related bugs. We show that this approach is effective at finding many bugs, without the false positives endemic to prior work on static analysis and runtime checking.

The ability to eliminate false positives is important, because false positives are time-consuming to deal with. In slogan form: false positives in static analysis waste the programmer's time; false positives in runtime checking waste the end user's time; while false positives in dynamic test generation waste the tool's time. Because an hour of CPU time is much cheaper than an hour of a human's time, dynamic test generation is an attractive way to find and fix integer bugs.

We have implemented our approach to finding integer bugs in *SmartFuzz*, a tool for performing symbolic execution and dynamic test generation on Linux x86 applications.

SmartFuzz works with binary executables directly, and does not require or use access to source code. Working with binaries has several advantages, most notably that we can generate tests directly from shipping binaries. In particular, we do not need to modify the build process for a program under test, which has been a pain point for static analysis tools [20]. Also, this allows us to perform whole-program analysis: we can find bugs that arise due to interactions between the application and libraries it uses, even if we don't have source code for those libraries. Of course, working with binary traces introduces special challenges, most notably the sheer size of the traces and the lack of type information that would be present in the source code. We discuss the challenges and design choices, in particular how SmartFuzz compares with SAGE, in Section 4.4.

In Section 4.5 we describe the techniques we use to generate test cases for integer bugs in dynamic test generation. We discovered that these techniques find many bugs, too many to track manually. To help us prioritize and manage these bug reports and streamline the process of reporting them to developers, we built *Metafuzz*, a web service for tracking test cases and bugs (Section 4.6). Metafuzz helps minimize the amount of human time required to find high-quality bugs and report them to developers, which is important because human time is the most expensive resource in a testing framework. Finally, Section 4.7 presents an empirical evaluation of our techniques and discusses our experience with these tools.

The contributions of this chapter are the following:

- We design novel algorithms for finding signed/unsigned conversion vulnerabilities using symbolic execution. In particular, we develop a novel type inference approach that allows us to detect which values in an x86 binary trace are used as signed integers, unsigned integers, or both. We discuss challenges in scaling such an analysis to commodity Linux media playing software and our approach to these challenges.

- We extend the range of integer bugs that can be found with symbolic execution, including integer overflows, integer underflows, width conversions, and signed/unsigned conversions. No prior symbolic execution tool has included the ability to detect all of these kinds of integer vulnerabilities. We follow up the work in the previous chapter with experiments on a wider range of integer conversion errors than previously reported.

- We implement these methods in *SmartFuzz*, a tool for symbolic execution and dynamic test generation of x86 binaries on Linux. We describe key challenges in symbolic execution of commodity Linux software, and we explain design choices in SmartFuzz motivated by these challenges.

- We report on the bug finding performance of SmartFuzz and compare SmartFuzz to the `zzuf` black box fuzz testing tool. The `zzuf` tool is a simple, yet effective, *fuzz testing* program which randomly mutates a given seed file to find new test inputs, without any knowledge or feedback from the target program. We have tested a broad range of commodity Linux software, including the media players `mplayer` and `ffmpeg`, the ImageMagick `convert` tool, and the `exiv2` TIFF metadata parsing library. This software comprises over one million lines of source code, and our test cases result in symbolic execution of traces that are millions of x86 instructions in length.

- We identify challenges with reporting bugs at scale, and introduce several techniques for addressing these challenges. For example, we present evidence that a simple stack hash is not sufficient for grouping test cases to avoid duplicate bug reports, and then we develop a *fuzzy stack hash* to solve these problems. Our experiments find approximately 77 total distinct bugs in 864 compute hours, giving us an average cost of $2.24 per bug at current Amazon EC2 rates. We quantify the overlap in bugs found by the two tools, and we show that SmartFuzz finds bugs missed by `zzuf`, including one program where SmartFuzz finds bugs but `zzuf` does not.

Between June 2008 and November 2008, Metafuzz has processed over 2,614 test runs from both SmartFuzz and the `zzuf` black box fuzz testing tool [52], comprising 2,361,595 test cases. To our knowledge, this is the largest number of test runs and test cases yet reported for dynamic test generation techniques. We have released our code under the GPL version 2 and BSD licenses[1]. Our vision is a service that makes it easy and inexpensive for software projects to find integer bugs and other serious security relevant code defects using dynamic test generation techniques. Our work shows that such a service is possible for a large class of commodity Linux programs.

4.2 Integer Bugs

We now describe the three main classes of integer bugs we want to find: integer overflow/underflow, width conversions, and signed/unsigned conversion errors [9]. All three classes of bugs occur due to the mismatch between machine arithmetic and arithmetic over unbounded integers.

[1]http://www.sf.net/projects/catchconv

Overflow/Underflow. Integer overflow (and underflow) bugs occur when an arithmetic expression results in a value that is larger (or smaller) than can be represented by the machine type. The usual behavior in this case is to silently "wrap around," e.g. for a 32-bit type, reduce the value modulo 2^{32}. Consider the function `badalloc` in Figure 4.1. If the multiplication `sz * n` overflows, the allocated buffer may be smaller than expected, which can lead to a buffer overflow later.

Width Conversions. Converting a value of one integral type to a wider (or narrower) integral type which has a different range of values can introduce *width conversion* bugs. For instance, consider `badcpy` in Figure 4.1. If the first parameter is negative, the conversion from `Int16` to `UInt32` will trigger sign-extension, causing `m` to be very large and likely leading to a buffer overflow. Because `memcpy`'s third argument is declared to have type `size_t` (which is an unsigned integer type), even if we passed `n` directly to `memcpy` the implicit conversion would still make this buggy. Width conversion bugs can also arise when converting a wider type to a narrower type.

Signed/Unsigned Conversion. Lastly, converting a signed integer type to an unsigned integer type of the same width (or vice versa) can introduce bugs, because this conversion can change a negative number to a large positive number (or vice versa). For example, consider `badcpy2` in Figure 4.1. If the first parameter `n` is a negative integer, it will pass the bounds check, then be promoted to a large unsigned integer when passed to `memcpy`. `memcpy` will copy a large number of bytes, likely leading to a buffer overflow.

```
char *badalloc(int sz, int n) {
  return (char *) malloc(sz * n);
}
void badcpy(Int16 n, char *p, char *q) {
  UInt32 m = n;
  memcpy(p, q, m);
}
void badcpy2(int n, char *p, char *q) {
  if (n > 800)
     return;
  memcpy(p, q, n);
}
```

Figure 4.1: Examples of three types of integer bugs.

4.3 Related Work

An earlier version of SmartFuzz and the Metafuzz web site infrastructure described in this paper were used for previous work that compares dynamic test generation with black-box fuzz testing by different authors [2]. That previous work does not describe the SmartFuzz tool, its design choices, or the Metafuzz infrastructure in detail. Furthermore, this paper works from new data on the effectiveness of SmartFuzz, except for an anecdote in our "preliminary experiences" section. We are not aware of other work that directly compares dynamic test generation with black-box fuzz testing on a scale similar to ours.

The most closely related work on integer bugs is Godefroid et al. [42], who describe dynamic test generation with bug-seeking queries for integer overflow, underflow, and some narrowing conversion errors in the context of the SAGE tool. Our work looks at a wider range of narrowing conversion errors, and we consider signed/unsigned conversion while their work does not. The EXE and KLEE tools also use integer overflow to prioritize different test cases in dynamic test generation, but they do not break out results on the number of bugs found due to this heuristic [17, 18]. The KLEE system also focuses on scaling dynamic test generation, but in a different way. While we focus on a few "large" programs in our results, KLEE focuses on high code coverage for over 450 smaller programs, as measured by trace size and source lines of code. These previous works also do not address the problem of type inference for integer types in binary traces.

IntScope is a static binary analysis tool for finding integer overflow bugs [87]. IntScope translates binaries to an intermediate representation, then it checks lazily for potentially harmful integer overflows by using symbolic execution for data that flows into "taint sinks" defined by the tool, such as memory allocation functions. SmartFuzz, in contrast, *eagerly* attempts to generate new test cases that cause an integer bug at the point in the program where such behavior could occur. This difference is due in part to the fact that IntScope reports errors to a programmer directly, while SmartFuzz filters test cases using a tool such as `memcheck`. As we argued in the Introduction, such a filter allows us to employ aggressive heuristics that may generate many test cases. Furthermore, while IntScope renders signed and unsigned comparisons in their intermediate representation by using hints from the x86 instruction set, they do not explicitly discuss how to use this information to perform type inference for signed and unsigned types, nor do they address the issue of scaling such inference to traces with millions of instructions. Finally, IntScope focuses only on integer overflow errors, while SmartFuzz covers underflow, narrowing conversion, and signed/unsigned conversion bugs in addition.

The dynamic test generation approach we use was introduced by Godefroid et al. [40] and independently by Cadar and Engler [15]. The SAGE system by Godefroid et al. works, as we do, on x86 binary programs and uses a generational search, but SAGE makes several different design choices we explain in Section 4.4. Lanzi et al. propose a design for dynamic test generation of x86 binaries that uses static analysis of loops to assist the solver, but their implementation is preliminary [59]. KLEE, in contrast, works with the intermediate representation generated by the Low-Level Virtual Machine target for `gcc` [18]. Larson and Austin applied symbolic range analysis to traces of programs to look for potential buffer overflow attacks, although they did not attempt to synthesize crashing inputs [60]. The BitBlaze [12] infrastructure of Song et al. also performs symbolic execution of x86 binaries, but their focus is on malware and signature generation, not on test generation.

Other approaches to integer bugs include static analysis and runtime detection. The Microsoft Prefast tool uses static analysis to warn about intraprocedural integer overflows [67]. Both Microsoft Visual C++ and `gcc` can add runtime checks to catch integer overflows in arguments to `malloc` and terminate a program. Brumley et al. provide rules for such runtime checks and show they can be implemented with low overhead on the x86 architecture by using jumps conditioned on the overflow bit in EFLAGS [11]. Both of these approaches fail to catch signed/unsigned conversion errors. Furthermore, both static analysis and runtime checking for overflow will flag code that is correct but relies on overflow semantics, while our approach only reports test cases in case of a crash or a Valgrind error report.

Blexim gives an introduction to integer bugs [10]. Fuzz testing has received a great deal of attention since its original introduction by Miller et al [68]. Notable public demonstrations of fuzzing's ability to find bugs include the Month of Browser Bugs and Month of Kernel Bugs [71, 64]. DeMott surveys recent work on fuzz testing, including the autodafe fuzzer, which uses `libgdb` to instrument functions of interest and adjust fuzz testing based on those functions' arguments [30, 86].

Our Metafuzz infrastructure also addresses issues not treated in previous work on test generation. First, we make *bug bucketing* a first-class problem and we introduce a *fuzzy stack hash* in response to developer feedback on bugs reported by Metafuzz. The SAGE paper reports bugs by stack hash, and KLEE reports on using the line of code as a bug bucketing heuristic, but we are not aware of other work that uses a fuzzy stack hash. Second, we report techniques for reducing the amount of human time required to process test cases generated by fuzzing and improve the quality of our error reports to developers; we are not aware of previous work on this topic. Such techniques are vitally important because human time is the most expensive part of a test infrastructure. Finally, Metafuzz uses on-demand computing with the Amazon Elastic Compute Cloud, and we explicitly quantify the cost of each bug found, which was not done in previous work.

4.4 Dynamic Test Generation

We describe the architecture of *SmartFuzz*, a tool for dynamic test generation of x86 binary programs on Linux. Dynamic test generation on x86 binaries—without access to source code—raises special challenges. We discuss these challenges and motivate our fundamental design choices.

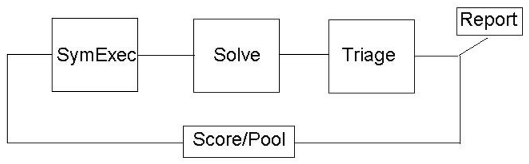


Figure 4.2: Dynamic test generation includes four stages: symbolic execution, solving to obtain new test cases, then triage to determine whether to report a bug or score the test case for addition to the pool of unexplored test cases.

4.4.1 Architecture

The SmartFuzz architecture is as follows: First, we add one or more test cases to a pool. Each test case in the pool receives a score given by the number of new basic blocks seen when running the target program on the test case. By "new" we mean that the basic block has not been observed while scoring any previous test case; we identify basic blocks by the instruction pointer of their entry point.

In each iteration of test generation, we choose a high-scoring test case, execute the program on that input, and use symbolic execution to generate a set of constraints that record how each intermediate value computed by the program relates to the inputs in the test case. SmartFuzz implements the symbolic execution and scoring components using the Valgrind binary analysis framework, and we use STP [36] to solve constraints.

For each symbolic branch, SmartFuzz adds a constraint that tries to force the program down a different path. We then query the constraint solver to see whether there exists any solution to the resulting set of constraints; if there is, the solution describes a new test case. We refer to these as *coverage queries* to the constraint solver.

SmartFuzz also injects constraints that are satisfied if a condition causing an error or potential error is satisfied (e.g., to force an arithmetic calculation to overflow). We then query the constraint solver; a solution describes a test case likely to cause an error. We refer to these as *bug-seeking queries* to the constraint solver. Bug-seeking queries come in different *types*, depending on the specific error they seek to exhibit in the program.

Both coverage and bug-seeking queries are explored in a *generational search* similar to the SAGE tool [43]. Each query from a symbolic trace is solved in turn, and new test cases created from successfully solved queries. A single symbolic execution therefore leads to many coverage and bug-seeking queries to the constraint solver, which may result in many new test cases.

We *triage* each new test case as it is generated, i.e. we determine if it exhibits a bug. If so, we report the bug; otherwise, we add the test case to the pool for scoring and possible symbolic execution. For triage, we use Valgrind `memcheck` on the target program with each test case, which is a tool that observes concrete execution looking for common programming errors [82]. We record any test case that causes the program to crash or triggers a `memcheck` warning.

We chose `memcheck` because it checks a variety of properties, including reads and writes to invalid memory locations, memory leaks, and use of uninitialized values. Re-implementing these analyses as part of the SmartFuzz symbolic execution tool would be wasteful and error-prone, as the `memcheck` tool has had the benefit of multiple years of use in large-scale projects such as Firefox and OpenOffice. The `memcheck` tool is known as a tool with a low false positive rate, as well, making it more likely that developers will pay attention to bugs reported by `memcheck`. Given a `memcheck` error report, developers do not even need to know that associated test case was created by SmartFuzz.

We do not attempt to classify the bugs we find as exploitable or not exploitable, because doing so by hand for the volume of test cases we generate is impractical. Many of the bugs found by `memcheck` are memory safety errors, which often lead to security vulnerabilities. Writes to invalid memory locations, in particular, are a red flag. Finally, to report bugs we use the Metafuzz framework described in Section 4.6.

4.4.2 Design Choices

Intermediate Representation. The sheer size and complexity of the x86 instruction set poses a challenge for analyzing x86 binaries. We decided to translate the underlying x86 code on-the-fly to an intermediate representation, then map the intermediate representation to symbolic formulas. Specifically, we used the Valgrind binary instrumentation tool to translate x86 instructions into VEX, the Valgrind intermediate representation [74]. The BitBlaze system works similarly, but with a different intermediate representation [12]. Details are available in an extended version of this paper[2].

Using an intermediate representation offers several advantages. First, it allows for a degree of platform independence: though we support only x86 in our current tool, the VEX library also supports the AMD64 and PowerPC instruction sets, with ARM support under active development. Adding support for these additional architectures requires only adding support for a small number of additional VEX instructions, not an entirely new instruction set from scratch. Second, the VEX library generates IR that satisfies the single static assignment property and performs other optimizations, which makes the translation from IR to formulas more straightforward. Third, and most importantly, this choice allowed

[2]`http://www.dmolnar.com/metafuzz-full.pdf`

us to outsource the pain of dealing with the minutae of the x86 instruction set to the VEX library, which has had years of production use as part of the Valgrind memory checking tool. For instance, we don't need to explicitly model the EFLAGS register, as the VEX library translates it to boolean operations. The main shortcoming with the VEX IR is that a single x86 instruction may expand to five or more IR instructions, which results in long traces and correspondingly longer symbolic formulas.

Online Constraint Generation. SmartFuzz uses *online* constraint generation, in which constraints are generated while the program is running. In contrast, SAGE (another tool for dynamic test generation) uses *offline* constraint generation, where the program is first traced and then the trace is replayed to generate constraints [43]. Offline constraint generation has several advantages: it is not sensitive to concurrency or nondeterminism in system calls; tracing has lower runtime overhead than constraint generation, so can be applied to running systems in a realistic environment; and, this separation of concerns makes the system easier to develop and debug, not least because trace replay and constraint generation is reproducible and deterministic. The main downside of offline constraint generation is that the tracing must record a large amount of information to avoid sensitivity to concurrency during re-execution; the iDNA framework used by SAGE logs all system call results and many reads and writes from memory, then attempts to compress this data by using different checkpoints of system state to allow for compact representation of what would otherwise be a prohibitively large instruction trace [7]. In short, offline constraint generation has important software engineering advantages.

SmartFuzz uses online constraint generation primarily because, when the Smart-

Fuzz project began, we were not aware of an available offline trace-and-replay framework with an intermediate representation comparable to VEX. Today, O'Callahan's `chronicle-recorder` could provide a starting point for a VEX-based offline constraint generation tool [76].

On the other hand, offline constraint generation can be limited if the tracing infrastructure does not record information that would be useful later for generating constraints. For example, to save space, some tracing infrastructures, including iDNA in its default setting, do not record the contents of memory unless the memory is read during that execution. This loses information that could be used for modeling symbolic pointer dereferences, because a symbolic pointer may point to memory that was not read during the execution but is fixed (e.g. indexing into a table of values).

Memory Model. Other symbolic execution tools such as EXE and KLEE model memory as a set of symbolic arrays, with one array for each allocated memory object. We do not. Instead, for each load or store instruction, we first concretize the memory address before accessing the symbolic heap. In particular, we keep a map M from concrete memory addresses to symbolic values. If the program reads from concrete address `a`, we retrieve a symbolic value from M(`a`). Even if we have recorded a symbolic expression a associated with this address, the symbolic address is ignored. Note that the value of `a` is known at constraint generation time and hence becomes (as far as the solver is concerned) a constant. Store instructions are handled similarly. Similarly, each store instruction updates this map by examining the concrete address that the program is writing to and updating the corresponding portion of the map.

While this approach sacrifices precision, it scales better to large traces. We note that the SAGE tool adopts a similar memory model. In particular, concretizing addresses generates symbolic formulas that the constraint solver can solve much more efficiently, because the solver does not need to reason about aliasing of pointers.

Only Tainted Data is Symbolic. We track the taint status of every byte in memory. As an optimization, we do not store symbolic information for untainted memory locations, because by definition untainted data is not dependent upon the untrusted inputs that we are trying to vary. We have found that only a tiny fraction of the data processed along a single execution path is tainted. Consequently, this optimization greatly reduces the size of our constraint systems and reduces the memory overhead of symbolic execution.

Focus on Fuzzing Files. We decided to focus on single-threaded programs, such as media players, that read a file containing untrusted data. Thus, a test case is simply the contents of this file, and SmartFuzz can focus on generating candidate files. This simplifies the symbolic execution and test case generation infrastructure, because there are a limited number of system calls that read from this file, and we do not need to account for concurrent interactions between threads in the same program. We know of no fundamental barriers, however, to extending our approach to multi-threaded and network-facing programs.

Our implementation associates a symbolic input variable with each byte of the input file. As a result, SmartFuzz cannot generate test cases with more bytes than are present in the initial seed file.

Multiple Cooperating Analyses. Our tool is implemented as a series of independent cooperating analyses in the Valgrind instrumentation framework. Each analysis adds its own instrumentation to a basic block during translation and exports an interface to the other analyses. For example, the instrumentation for tracking taint flow, which determines the IR instructions to treat as symbolic, exports an interface that allows querying whether a specific memory location or temporary variable is symbolic. A second analysis then uses this interface to determine whether or not to output STP constraints for a given IR instruction.

The main advantage of this approach is that it makes it easy to add new features by adding a new analysis, then modifying our core constraint generation instrumentation. Also, this decomposition enabled us to extract our taint-tracking code and use it in a different project with minimal modifications, and we were able to implement the binary type inference analysis described in Section 4.5, replacing a different earlier version, without changing our other analyses.

Optimize in Postprocessing. Another design choice was to output constraints that are as "close" as possible to the intermediate representation, performing only limited optimizations on the fly. For example, we implement the "related constraint elimination," as introduced by tools such as EXE and SAGE [17, 43], as a post-processing step on constraints created by our tool. We then leave it up to the solver to perform common subexpression elimination, constant propagation, and other optimizations. The main benefit of this choice is that it simplifies our constraint generation. One drawback of this choice is that current solvers, including STP, are not yet capable of "remembering" optimizations from one query to the next, leading to redundant work on the part of the solver. The main drawback of this choice, however, is that while after optimization each individual query is small, the total symbolic trace containing all queries for a program can be several gigabytes. When running our tool on a 32-bit host machine, this can cause problems with maximum file size for a single file or maximum memory size in a single process.

4.5 Techniques for Finding Integer Bugs

We now describe the techniques we use for finding integer bugs.

Overflow/Underflow. For each arithmetic expression that could potentially overflow or underflow, we emit a constraint that is satisfied if the overflow or underflow occurs. If our solver can satisfy these constraints, the resulting input values will likely cause an underflow or overflow, potentially leading to unexpected behavior.

Width Conversions. For each conversion between integer types, we check whether it is possible for the source value to be outside the range of the target value by adding a constraint that's satisfied when this is the case and then applying the constraint solver. For conversions that may sign-extend, we use the constraint solver to search for a test case where the high bit of the source value is non-zero.

Signed/Unsigned Conversions. Our basic approach is to try to reconstruct, from the x86 instructions executed, signed/unsigned type information about all integral values. This information is present in the source code but not in the binary, so we describe an algorithm to infer this information automatically.

Consider four types for integer values: "Top," "Signed," "Unsigned," or "Bottom." Here, "Top" means the value has not been observed in the context of a signed or unsigned integer; "Signed" means that the value has been used as a signed integer; "Unsigned" means the value has been used as an unsigned integer; and "Bottom" means that the value has been used inconsistently as both a signed and unsigned integer. These types form a four-

```
int main(int argc, char** argv) {
  char * p = malloc(800);
  char * q = malloc(800);
  int n;
  n = atol(argv[1]);
  if (n > 800)
      return;
  memcpy(p, q, n);
  return 0;
}
```

Figure 4.3: A simple test case for dynamic type inference and query generation. The signed comparison `n > 800` and unsigned `size_t` argument to `memcpy` assign the type "Bottom" to the value associated with `n`. When we solve for an input that makes `n` negative, we obtain a test case that reveals the error.

point lattice. Our goal is to find symbolic program values that have type "Bottom." These values are candidates for signed/unsigned conversion errors. We then attempt to synthesize an input that forces these values to be negative.

We associate every instance of every temporary variable in the Valgrind intermediate representation with a type. Every variable in the program starts with type Top. During execution we add *type constraints* to the type of each value. For x86 binaries, the sources of type constraints are signed and unsigned comparison operators: e.g., a signed comparison between two values causes both values to receive the "Signed" type constraint. We also add unsigned type constraints to values used as the length argument of `memcpy` function, which we can detect because we know the calling convention for x86 and we have debugging symbols for glibc. While the x86 instruction set has additional operations, such as `IMUL` that reveal type information about their operands, we do not consider these; this means only that we may incorrectly under-constrain the types of some values.

Any value that has received both a signed and unsigned type constraint receives the type Bottom. After adding a type constraint, we check to see if the type of a value has moved to Bottom. If so, we attempt to solve for an input which makes the value negative. We do this because negative values behave differently in signed and unsigned comparisons, and so they are likely to exhibit an error if one exists. All of this information is present in the trace without requiring access to the original program source code.

We discovered, however, that `gcc 4.1.2` inlines some calls to `memcpy` by transforming them to `rep movsb` instructions, even when the `-O` flag is not present. Furthermore, the Valgrind IR generated for the `rep movsb` instruction compares a decrementing counter variable to zero, instead of counting up and executing an unsigned comparison to the loop bound. As a result, on gcc 4.1.2 a call to `memcpy` does not cause its length argument to be marked as unsigned. To deal with this problem, we implemented a simple heuristic to detect the IR generated for `rep movsb` and emit the appropriate constraint. We verified that this heuristic works on a small test case similar to Figure 4.3, generating a test input that caused a segmentation fault.

A key problem is storing all of the information required to carry out type inference without exhausting available memory. Because a trace may have several million instructions, memory usage is key to scaling type inference to long traces. Furthermore, our algorithm requires us to keep track of the types of all values in the program, unlike constraint generation, which need concern itself only with tainted values. An earlier version of our analysis created a special "type variable" for each value, then maintained a map from IR locations to type variables. Each type variable then mapped to a type. We found that in addition to being hard to maintain, this analysis often led to a number of live type variables that scaled linearly with the number of executed IR instructions. The result was that our analysis ran out of memory when attempting to play media files in the `mplayer` media player.

To solve this problem, we developed a garbage-collected data structure for tracking type information. To reduce memory consumption, we use a union-find data structure to partition integer values into equivalence classes where all values in an equivalence class are required to have the same type. We maintain one type for each union-find equivalence class; in our implementation type information is associated with the representative node for that equivalence class. Assignments force the source and target values to have the same types, which is implemented by merging their equivalence classes. Updating the type for a value can be done by updating its representative node's type, with no need to explicitly update the types of all other variables in the equivalence class.

It turns out that this data structure is acyclic, due to the fact that VEX IR is in SSA form. Therefore, we use reference counting to garbage collect these nodes. In addition, we benefit from an additional property of the VEX IR: all values are either stored in memory, in registers, or in a temporary variable, and the lifetime of each temporary variable is implicitly limited to that of a single basic block. Therefore, we maintain a list of temporaries that are live in the current basic block; when we leave the basic block, the type information associated with all of those live temporaries can be deallocated. Consequently, the amount of memory needed for type inference at any point is proportional to the number of tainted (symbolic) variables that are live at that point—which is a significant improvement over the naive approach to type inference. The Appendix contains a more detailed specification of these algorithms.

4.6 Triage and Reporting at Scale

Both SmartFuzz and `zzuf` can produce hundreds to thousands of test cases for a single test run. We designed and built a web service, *Metafuzz*, to manage the volume of tests. We describe some problems we found while building Metafuzz and techniques to overcoming these problems. Finally, we describe the user experience with Metafuzz and bug reporting.

4.6.1 Problems and Techniques

The Metafuzz architecture is as follows: first, a Test Machine generates new test cases for a program and runs them locally. The Test Machine then determines which test cases exhibit bugs and sends these test cases to Metafuzz. The Metafuzz web site displays these test cases to the User, along with information about what kind of bug was found in which target program. The User can pick test cases of interest and download them for further investigation. We now describe some of the problems we faced when designing Metafuzz, and our techniques for handling them. Section 4.7 reports our experiences with using Metafuzz to manage test cases and report bugs.

Problem: Each test run generated many test cases, too many to examine by hand.

Technique: We used Valgrind's `memcheck` to automate the process of checking whether a particular test case causes the program to misbehave. `Memcheck` looks for memory leaks, use of uninitialized values, and memory safety errors such as writes to memory that was not allocated [82]. If `memcheck` reports an error, we save the test case. In addition, we looked for core dumps and non-zero program exit codes.

Problem: Even after filtering out the test cases that caused no errors, there were still many test cases that do cause errors.

Technique: The `metafuzz.com` front page is a HTML page listing all of the potential bug reports. showing all potential bug reports. Each test machine uploads information about test cases that trigger bugs to Metafuzz.

Problem: The machines used for testing had no long-term storage. Some of the test cases were too big to attach in e-mail or Bugzilla, making it difficult to share them with developers.

Technique: Test cases are uploaded directly to Metafuzz, providing each one with a stable URL. Each test case also includes the Valgrind output showing the Valgrind error, as well as the output of the program to `stdout` and `stderr`.

Problem: Some target projects change quickly. For example, we saw as many as four updates per day to the `mplayer` source code repository. Developers reject bug reports against "out of date" versions of the software.

Technique: We use the Amazon Elastic Compute Cloud (EC2) to automatically attempt to reproduce the bug against the latest version of the target software. A button on the Metafuzz site spawns an Amazon EC2 instance that checks out the most recent version of the target software, builds it, and then attempts to reproduce the bug.

Problem: Software projects have specific reporting requirements that are tedious to implement by hand. For example, `mplayer` developers ask for a stack backtrace, disassembly, and register dump at the point of a crash.

Technique: Metafuzz automatically generates bug reports in the proper format from the failing test case. We added a button to the Metafuzz web site so that we can review the resulting bug report and then send it to the target software's bug tracker with a single click.

Problem: The same bug often manifests itself as many failing test cases. Reporting the same bug to developers many times wastes developer time.

Technique: We use the call stack to identify multiple instances of the same bug. Valgrind memcheck reports the call stack at each error site, as a sequence of instruction pointers. If debugging information is present, it also reports the associated filename and line number information in the source code.

Initially, we computed a *stack hash* as a hash of the sequence of instruction pointers in the backtrace. This has the benefit of not requiring debug information or symbols. Unfortunately, we found that a naive stack hash has several problems. First, it is sensitive to address space layout randomization (ASLR), because different runs of the same program may load the stack or dynamically linked libraries at different addresses, leading to different hash values for call stacks that are semantically the same. Second, even without ASLR, we found several cases where a single bug might be triggered at multiple call stacks that were similar but not identical. For example, a buggy function can be called in several different places in the code. Each call site then yields a different stack hash. Third, any slight change to the target software can change instruction pointers and thus cause the same bug to receive a different stack hash. While we do use the stack hash on the client to avoid uploading test cases for bugs that have been previously found, we found that we could not use stack hashes alone to determine if a bug report is novel or not.

To address these shortcomings, we developed a *fuzzy stack hash* that is forgiving of slight changes to the call stack. We use debug symbol information to identify the name of the function called, the line number in source code (excluding the last digit of the line number, to allow for slight changes in the code), and the name of the object file for each frame in the call stack. We then hash all of this information for the three functions at the top of the call stack.

The choice of the number of functions to hash determines the "fuzzyness" of the hash. At one extreme, we could hash all extant functions on the call stack. This would be similar to the classic stack hash and report many semantically same bugs in different buckets. On the other extreme, we could hash only the most recently called function. This fails in cases where two semantically different bugs both exhibit as a result of calling `memcpy` or some other utility function with bogus arguments. In this case, both call stacks would end with `memcpy` even though the bug is in the way the arguments are computed. We chose three functions as a trade-off between these extremes; we found this sufficient to stop further reports from the `mplayer` developers of duplicates in our initial experiences. Finding the best fuzzy stack hash is interesting future work; we note that the choice of bug bucketing technique may depend on the program under test.

While any fuzzy stack hash, including ours, may accidentally lump together two distinct bugs, we believe this is less serious than reporting duplicate bugs to developers. We added a post-processing step on the server that computes the fuzzy stack hash for test cases that have been uploaded to Metafuzz and uses it to coalesce duplicates into a single bug bucket.

Problem: Because Valgrind `memcheck` does not terminate the program after seeing an error, a single test case may give rise to dozens of Valgrind error reports. Two different test cases may share some Valgrind errors but not others.

Technique: First, we put a link on the Metafuzz site to a single test case for each bug bucket. Therefore, if two test cases share some Valgrind errors, we only use one test case for each of the errors in common. Second, when reporting bugs to developers, we highlight in the title the specific bugs on which to focus.

4.7 Results

4.7.1 Preliminary Experience

We used an earlier version of SmartFuzz and Metafuzz in a project carried out by a group of undergraduate students over the course of eight weeks in Summer 2008. When beginning the project, none of the students had any training in security or bug reporting. We provided a one-week course in software security. We introduced SmartFuzz, `zzuf`, and Metafuzz, then asked the students to generate test cases and report bugs to software developers. By the end of the eight weeks, the students generated over 1.2 million test cases, from which they reported over 90 bugs to software developers, principally to the `mplayer` project, of which 14 were fixed. For further details, we refer to their presentation [2].

	SLOC	seedfile type and size	Branches	x86 instrs	IRStmts	asserts	queries
mplayer	723468	MP3 (159000 bytes)	20647045	159500373	810829992	1960	36
ffmpeg	304990	AVI (980002 bytes)	4147710	19539096	115036155	4778690	462346
exiv2	57080	JPG (22844 bytes)	809806	6185985	32460806	81450	1006
gzip	140036	TAR.GZ (14763 bytes)	24782	161118	880386	95960	13309
bzip	26095	TAR.BZ2 (618620 bytes)	107396936	746219573	4185066021	1787053	314914
ImageMagick	300896	PNG (25385 bytes)	98993374	478474232	2802603384	583	81

Figure 4.4: The size of our test programs. We report the source lines of code for each test program and the size of one of our seed files, as measured by David A. Wheeler's `sloccount`. Then we run the test program on that seed file and report the total number of branches, x86 instructions, Valgrind IR statements, STP assert statements, and STP query statements for that run. We ran symbolic execution for a maximum of 12 hours, which was sufficient for all programs except `mplayer`, which terminated during symbolic execution.

4.7.2 Experiment Setup

Test Programs. Our target programs were `mplayer` version `SVN-r28403-4.1.2`, `ffmpeg` version `SVN-r16903`, `exiv2` version `SVN-r1735`, `gzip` version 1.3.12, `bzip2` version 1.0.5, and ImageMagick `convert` version $6.4.8 - 10$, which are all widely used media and compression programs. Table 4.4 shows information on the size of each test program. Our test programs are large, both in terms of source lines of code and trace lengths. The percentage of the trace that is symbolic, however, is small.

Test Platform. Our experiments were run on the Amazon Elastic Compute Cloud (EC2), employing a "small" and a "large" instance image with SmartFuzz, `zzuf`, and all our test programs pre-installed. At this writing, an EC2 small instance has 1.7 GB of RAM and a single-core virtual CPU with performance roughly equivalent to a 1GHz 2007 Intel Xeon. An EC2 large instance has 7 GB of RAM and a dual-core virtual CPU, with each core

	mplayer	ffmpeg	exiv2	gzip	bzip2	convert
Coverage	2599	14535	1629	5906	12606	388
ConversionNot32	0	3787	0	0	0	0
Conversion32to8	1	26	740	2	10	116
Conversion32to16	0	16004	0	0	0	0
Conversion16Sto32	0	121	0	0	0	0
SignedOverflow	1544	37803	5941	24825	9109	49
SignedUnderflow	3	4003	48	1647	2840	0
UnsignedOverflow	1544	36945	4957	24825	9104	35
UnsignedUnderflow	0	0	0	0	0	0
MallocArg	0	24	0	0	0	0
SignedUnsigned	2568	21064	799	7883	17065	49

Figure 4.5: The number of each query type for each test program after a single 24-hour run.

having performance roughly equivalent to a 1 GHz Xeon.

We ran all `mplayer` runs and `ffmpeg` runs on EC2 large instances, and we ran all other test runs with EC2 small instances. We spot-checked each run to ensure that instances successfully held all working data in memory during symbolic execution and triage without swapping to disk, which would incur a significant performance penalty. For each target program we ran SmartFuzz and `zzuf` with three seed files, for 24 hours per program per seed file. Our experiments took 288 large machine-hours and 576 small machine-hours, which at current EC2 prices of $0.10 per hour for small instances and $0.40 per hour for large instances cost $172.80.

Query Types. SmartFuzz queries our solver with the following types of queries: `Coverage`, `ConversionNot32`, `Conversion32to8`, `Conversion32to16`, `SignedOverflow`, `UnsignedOverflow`, `SignedUnderflow`, `UnsignedUnderflow`, `MallocArg`, and `SignedUnsigned`. `Coverage` queries refer to queries created as part of the generational search by flipping path conditions. The others are *bug-seeking queries* that attempt to synthesize inputs leading to specific kinds of bugs. Here `MallocArg` refers to a set of bug-seeking queries that attempt to force inputs to known memory allocation functions to be negative, yielding an implicit conversion to a large unsigned integer, or force the input to be small.

Experience Reporting to Developers. Our original strategy was to report all distinct bugs to developers and let them judge which ones to fix. The `mplayer` developers gave us feedback on this strategy. They wanted to focus on fixing the most serious bugs, so they preferred seeing reports only for out-of-bounds writes and double free errors. In contrast, they were not as interested in out-of-bound reads, even if the resulting read caused a segmentation fault. This helped us prioritize bugs for reporting.

4.7.3 Bug Statistics

Integer Bug-Seeking Queries Yield Bugs. Figure 4.6 reports the number of each type of query to the constraint solver over all test runs. For each type of query, we report the number of test files generated and the number of distinct bugs, as measured by our fuzzy stack hash. Some bugs may be revealed by multiple different kinds of queries, so there may be overlap between the bug counts in two different rows of Figure 4.6.

The table shows that our dynamic test generation methods for integer bugs succeed in finding bugs in our test programs. Furthermore, the queries for signed/unsigned bugs found the most distinct bugs out of all bug-seeking queries. This shows that our novel method for detecting signed/unsigned bugs (Section 4.5) is effective at finding bugs.

	Queries	Test Cases	Bugs
Coverage	588068	31121	19
ConversionNot32	4586	0	0
Conversion32to8	1915	1377	3
Conversion32to16	16073	67	4
Conversion16Sto32	206	0	0
SignedOverflow	167110	0	0
SignedUnderflow	20198	21	3
UnsignedOverflow	164155	9280	3
MallocArg	30	0	0
SignedUnsigned	125509	6949	5

Figure 4.6: The number of bugs found, by query type, over all test runs. The fourth column shows the number of distinct bugs found from test cases produced by the given type of query, as classified using our fuzzy stack hash.

SmartFuzz Finds More Bugs Than zzuf, on mplayer. For `mplayer`, SmartFuzz generated 10,661 test cases over all test runs, while `zzuf` generated 11,297 test cases; SmartFuzz found 22 bugs while `zzuf` found 13. Therefore, in terms of number of bugs, SmartFuzz outperformed `zzuf` for testing `mplayer`. Another surprising result here is that SmartFuzz generated nearly as many test cases as `zzuf`, despite the additional overhead for symbolic execution and constraint solving. This shows the effect of the generational search and the choice of memory model; we leverage a single expensive symbolic execution and fast solver queries to generate many test cases. At the same time, we note that `zzuf` found a serious InvalidWrite bug, while SmartFuzz did not.

A previous version of our infrastructure had problems with test cases that caused the target program to run forever, causing the search to stall. Therefore, we introduced a timeout, so that after 300 CPU seconds, the target program is killed. We manually examined the output of `memcheck` from all killed programs to determine whether such test cases represent errors. For `gzip` we discovered that SmartFuzz created six such test cases, which account for the two out-of-bounds read (InvalidRead) errors we report; `zzuf` did not find any hanging test cases for `gzip`. We found no other hanging test cases in our other test runs.

Different Bugs Found by SmartFuzz and `zzuf`. We ran the same target programs with the same seed files using `zzuf`. Figure 4.7 shows bugs found by each type of fuzzer. With respect to each tool, SmartFuzz found 37 total distinct bugs and `zzuf` found 59 distinct bugs. We found some overlap between bugs as well: 19 bugs were found by both fuzz testing tools, for a total of 77 distinct bugs.

This shows that while there is overlap between the two tools, SmartFuzz finds bugs that `zzuf` does not and vice versa. Therefore, it makes sense to try both tools when testing software.

Note that we did not find any bugs for `bzip2` with either fuzzer, so neither tool was effective on this program. This shows that fuzzing is not always effective at finding bugs, especially with a program that has already seen attention for security vulnerabilities. We also note that SmartFuzz found InvalidRead errors in `gzip` while `zzuf` found no bugs in this program. Therefore `gzip` is a case where SmartFuzz's directed testing is able to trigger a bug, but purely random testing is not.

Different zzuf Bugs In Repeated Runs. We then re-ran our 24 hour experiments with **zzuf** to determine whether the number of bugs found changed. Table 4.8 shows that the

	mplayer		ffmpeg		exiv2		gzip		convert	
SyscallParam	4	3	2	3	0	0	0	0	0	0
UninitCondition	13	1	1	8	0	0	0	0	3	8
UninitValue	0	3	0	3	0	0	0	0	0	2
Overlap	0	0	0	1	0	0	0	0	0	0
Leak_DefinitelyLost	2	2	2	4	0	0	0	0	0	0
Leak_PossiblyLost	2	1	0	2	0	1	0	0	0	0
InvalidRead	1	2	0	4	4	6	2	0	1	1
InvalidWrite	0	1	0	3	0	0	0	0	0	0
Total	22	13	5	28	4	7	2	0	4	11
Cost per bug	\$1.30	\$2.16	\$5.76	\$1.03	\$1.80	\$1.03	\$3.60	NA	\$1.20	\$0.65

Figure 4.7: The number of bugs, after fuzzy stack hashing, found by SmartFuzz (the number on the left in each column) and `zzuf` (the number on the right). We also report the cost per bug, assuming \$0.10 per small compute-hour, \$0.40 per large compute-hour, and 3 runs of 24 hours each per target for each tool.

	mplayer		ffmpeg		exiv2		gzip		convert	
SyscallParam	1	3	0	3	0	0	0	0	1	0
UninitCondition	0	1	1	8	0	0	0	0	0	8
UninitValue	6	3	1	3	0	0	0	0	0	2
Overlap	0	0	0	1	0	0	0	0	0	0
Leak_DefinitelyLost	2	2	1	4	0	0	1	0	1	0
Leak_PossiblyLost	0	1	0	2	0	1	0	0	0	0
InvalidRead	4	2	0	4	0	6	0	0	0	1
InvalidWrite	0	1	0	3	0	0	0	0	0	0
Total	13	13	3	28	0	7	1	0	2	11

Figure 4.8: The number of bugs found by **zzuf** on two different sets of 24 hour experiments with three seed files. The number on the right in each column is the number of bug buckets found in the first set of experiments of each type. The number on the left is the number of bug buckets of each type found in the second set.

number of bugs found changed substantially from one run to the next. This shows that the random choices made by **zzuf** have a substantial impact on the specific bugs found, even when testing the same program. We note that in our second round of experiments, **zzuf** was able to successfully find a bug in `gzip`, a memory leak, while the first round of experiments with **zzuf** found no bugs in `gzip`.

Block Coverage. We measured the number of basic blocks in the program visited by the execution of the seed file, then measured how many new basic blocks were visited during the test run. We discovered `zzuf` added a higher percentage of new blocks than SmartFuzz in 13 of the test runs, while SmartFuzz added a higher percentage of new blocks in 4 of the test runs (the SmartFuzz `convert-2` test run terminated prematurely.) Table 4.9 shows the initial basic blocks, the number of blocks added, and the percentage added for each fuzzer. We see that the effectiveness of SmartFuzz varies by program; for `convert` it is particularly effective, finding many more new basic blocks than `zzuf`.

Contrasting SmartFuzz and zzuf Performance. Despite the limitations of random testing, the blackbox fuzz testing tool `zzuf` found bugs in four out of our six test programs. In three of our test programs, `zzuf` found more bugs than SmartFuzz. Furthermore, `zzuf` found the most serious InvalidWrite errors, while SmartFuzz did not. These results seem surprising, because SmartFuzz exercises directed testing based on program behavior while `zzuf` is a purely blackbox tool. We would expect that SmartFuzz should find all the bugs found by `zzuf`, given an unbounded amount of time to run both test generation methods.

In practice, however, the time which we run the methods is limited, and so the question is which bugs are discovered first by each method. We have identified possible reasons for this behavior, based on examining the test cases and Valgrind errors generated by both tools[3]. We now list and briefly explain these reasons.

Header parsing errors. The errors we observed are often in code that parses the header of a file format. For example, we noticed bugs in functions of `mplayer` that parse MP3 headers. These errors can be triggered by simply placing "wrong" values in data fields of header files. As a result, these errors do not require complicated and unlikely predicates to be true before reaching buggy code, and so the errors can be reached without needing the full power of dynamic test generation. Similarly, code coverage may be affected by the style of program used for testing.

Difference in number of bytes changed. The two different methods change a vastly different number of bytes from the original seed file to create each new test case. SmartFuzz changes exactly those bytes that must change to force program execution down a single new path or to violate a single property. Below we show that in our programs there are only a small number of constraints on the input to solve, so a SmartFuzz test case differs from its seed file by only a small number of bytes. In contrast, `zzuf` changes a fixed fraction of the input bytes to random other bytes; in our experiments, we left this at the default value of 0.004.

The large number of changes in a single fuzzed test case means that for header parsing or other code that looks at small chunks of the file independently, a single `zzuf` test case will exercise many different code paths with fuzzed chunks of the input file. Each path which originates from parsing a fuzzed chunk of the input file is a potential bug. Furthermore, because Valgrind `memcheck` does not necessarily terminate the program's execution

[3]We have placed representative test cases from each method at `http://www.metafuzz.com/example-testcases.tgz`

when a memory safety error occurs, one such file may yield multiple bugs and corresponding bug buckets.

In contrast, the SmartFuzz test cases usually change only one chunk and so explore only one "fuzzed" path through such code. Our code coverage metric, unfortunately, is not precise enough to capture this difference because we measured block coverage instead of path coverage. This means once a set of code blocks has been covered, a testing method receives no further credit for additional paths through those same code blocks.

Small number of generations reached by SmartFuzz. Our 24 hour experiments with SmartFuzz tended to reach a small number of generations. While previous work on whitebox fuzz testing shows that most bugs are found in the early generations [43], this feeds into the previous issue because the number of differences between the fuzzed file and the original file is proportional to the generation number of the file. We are exploring longer-running SmartFuzz experiments of a week or more to address this issue.

Loop behavior in SmartFuzz. Finally, SmartFuzz deals with loops by looking at the unrolled loop in the dynamic program trace, then attempting to generate test cases for each symbolic `if` statement in the unrolled loop. This strategy is not likely to create new test cases that cause the loop to execute for vastly more iterations than seen in the trace. By contrast, the `zzuf` case may get lucky by assigning a random and large value to a byte sequence in the file that controls the loop behavior. On the other hand, when we looked at the `gzip` bug found by SmartFuzz and not by `zzuf`, we discovered that it appears to be due to an infinite loop in the `inflate_dynamic` routine of `gzip`.

	Initial basic blocks		Blocks added by tests		Ratio of prior two columns	
Test run	SmartFuzz	`zzuf`	SmartFuzz	`zzuf`	SmartFuzz	`zzuf`
mplayer-1	7819	7823	5509	326	70%	4%
mplayer-2	11375	11376	908	1395	7%	12%
mplayer-3	11093	11096	102	2472	0.9%	22%
ffmpeg-1	6470	6470	592	20036	9.14%	310%
ffmpeg-2	6427	6427	677	2210	10.53%	34.3%
ffmpeg-3	6112	611	97	538	1.58%	8.8%
convert-1	8028	8246	2187	20	27%	0.24%
convert-2	8040	8258	2392	6	29%	0.073%
convert-3	NA	10715	NA	1846	NA	17.2%
exiv2-1	9819	9816	2934	3560	29.9%	36.3%
exiv2-2	9811	9807	2783	3345	28.3%	34.1%
exiv2-3	9814	9810	2816	3561	28.7%	36.3%
gzip-1	2088	2088	252	334	12%	16%
gzip-2	2169	2169	259	275	11.9%	12.7%
gzip-3	2124	2124	266	316	12%	15%
bzip2-1	2779	2778	123	209	4.4%	7.5%
bzip2-2	2777	2778	125	237	4.5%	8.5%
bzip2-3	2823	2822	115	114	4.1%	4.0%

Figure 4.9: Coverage metrics: the initial number of basic blocks, before testing; the number of blocks added during testing; and the percentage of blocks added.

4.7.4 SmartFuzz Statistics

Integer Bug Queries Vary By Program. Table 4.5 shows the number of solver queries of each type for one of our 24-hour test runs. We see that the type of queries varies from one program to another. We also see that for `bzip2` and `mplayer`, queries generated by type inference for signed/unsigned errors account for a large fraction of all queries to the constraint solver. This results from our choice to eagerly generate new test cases early in the program; because there are many potential integer bugs in these two programs, our symbolic traces have many integer bug-seeking queries. Our design choice of using an independent tool such as `memcheck` to filter the resulting test cases means we can tolerate such a large number of queries because they require little human oversight.

Time Spent In Each Task Varies By Program. Figure 4.10 shows the percentage of time spent in symbolic execution, coverage, triage, and recording for each run of our experiment. We also report an "Other" category, which includes the time spent in the constraint solver. This shows us where we can obtain gains through further optimization. The amount of time spent in each task depends greatly on the seed file, as well as on the target program. For example, the first run of `mplayer`, which used a mp3 seedfile, spent 98.57% of total time in symbolic execution, and only 0.33% in coverage, and 0.72% in triage. In contrast, the second run of `mplayer`, which used a mp4 seedfile, spent only 14.77% of time in symbolic execution, but 10.23% of time in coverage, and 40.82% in triage. We see that the speed of symbolic execution and of triage is the major bottleneck for several of our test runs, however. This shows that future work should focus on improving these two areas.

	Total	SymExec	Coverage	Triage	Record	Other
gzip-1	206522s	0.1%	0.06%	0.70%	17.6%	81.6%
gzip-2	208999s	0.81%	0.005%	0.70%	17.59%	80.89%
gzip-3	209128s	1.09%	0.0024%	0.68%	17.8%	80.4%
bzip2-1	208977s	0.28%	0.335%	1.47%	14.0%	83.915%
bzip2-2	208849s	0.185%	0.283%	1.25%	14.0%	84.32%
bzip2-3	162825s	25.55%	0.78%	31.09%	3.09%	39.5%
mplayer-1	131465s	14.2%	5.6%	22.95%	4.7%	52.57%
mplayer-2	131524s	15.65%	5.53%	22.95%	25.20%	30.66%
mplayer-3	49974s	77.31%	0.558%	1.467%	10.96%	9.7%
ffmpeg-1	73981s	2.565%	0.579%	4.67%	70.29%	21.89%
ffmpeg-2	131600s	36.138%	1.729%	9.75%	11.56%	40.8%
ffmpeg-3	24255s	96.31%	0.1278%	0.833%	0.878%	1.8429%
convert-1	14917s	70.17%	2.36%	24.13%	2.43%	0.91%
convert-2	97519s	66.91%	1.89%	28.14%	2.18%	0.89%
exiv2-1	49541s	3.62%	10.62%	71.29%	9.18%	5.28%
exiv2-2	69415s	3.85%	12.25%	65.64%	12.48%	5.78%
exiv2-3	154334s	1.15%	1.41%	3.50%	8.12%	85.81%

Figure 4.10: The percentage of time spent in each of the phases of SmartFuzz. The second column reports the total wall-clock time, in seconds, for the run; the remaining columns are a percentage of this total. The “other” column includes the time spent solving STP queries.

	average before	average after	ratio (before/after)
mplayer	292524	33092	8.84
ffmpeg	350846	83086	4.22
exiv2	81807	10696	7.65
gzip	348027	199336	1.75
bzip2	278980	162159	1.72
convert	2119	1057	2.00

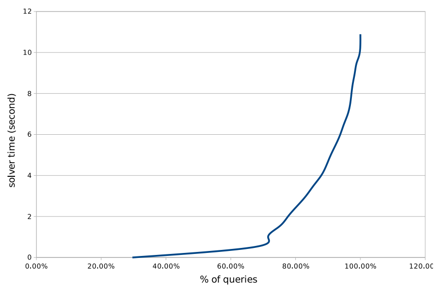


Figure 4.11: On the left, average query size before and after related constraint optimization for each test program. On the right, an empirical CDF of solver times.

4.7.5 Solver Statistics

Related Constraint Optimization Varies By Program. We measured the size of all queries to the constraint solver, both before and after applying the related constraint optimization described in Section 4.4. Figure 4.11 shows the average size for queries from each test program, taken over all queries in all test runs with that program. We see that while the optimization is effective in all cases, its average effectiveness varies greatly from one test program to another. This shows that different programs vary greatly in how many input bytes influence each query.

The Majority of Queries Are Fast. Figure 4.11 shows the empirical cumulative distribution function of STP solver times over all our test runs. For about 70% of the test cases, the solver takes at most one second. The maximum solver time was about 10.89 seconds. These results reflect our choice of memory model and the effectiveness of the related constraint optimization. Because of these, the queries to the solver consist only of operations over bitvectors (with no array constraints), and most of the sets of constraints sent to the solver are small, yielding fast solver performance.

4.8 Conclusion

We described new methods for finding integer bugs in dynamic test generation, and we implemented these methods in SmartFuzz, a new dynamic test generation tool for x86 Linux binary programs. We then reported on our experiences building the web site `metafuzz.com` and using it to manage test case generation at scale. In particular, we found that SmartFuzz finds bugs not found by `zzuf` and vice versa, showing that a comprehensive testing strategy should use both white-box and black-box test generation tools.

Furthermore, we showed that our methods can find integer bugs without the false positives inherent to static analysis or runtime checking approaches, and we showed that our methods scale to commodity Linux media playing software. The Metafuzz web site is live, and we have released our code to allow others to use our work.

Chapter 5

Conclusion

We have shown that dynamic test generation and whitebox fuzz testing is practical for large scale testing of widely-used software on the Windows and Linux platforms. Our work shows that modern constraint satisfaction and binary analysis tools are mature enough to create a robust testing service, capable of checking multiple properties and finding high value bugs at modest cost. The work described in this thesis, however, is only a beginning. We now describe future directions in whitebox fuzz testing itself. We then discuss directions for improving the remaining stages of the "bug cycle."

5.1 Whitebox Fuzz Testing: Future Directions

We now sketch several future directions for work in whitebox fuzz testing. For each direction we give pointers to relevant research work and list some ideas for progress.

5.1.1 More Application Classes

Our work focuses on file-reading applications for Windows and Linux. These applications are important because they include media players and file parsers, which account for a large number of vulnerabilities exploited in practice. Even so, work remains to extend the approach to other classes of applications, including network facing applications, web applications, and distributed applications.

In the case of network-facing applications, the Tupni project uses symbolic execution techniques to extract a grammar which approximates the language of inputs accepted by a network application. One of the main motivations for such a grammar is as an input to a grammar-based black box fuzz testing tool, such as SPIKE or FuzzGuru [84]. Directly applying whitebox fuzz testing techniques here may yield quicker discovery of high value bugs. In addition, there is the interesting question of how to properly trace and replay client/server applications.

Web applications, where the client consists of code running inside a web browser and the server runs as part of a web server, are a major and fast growing class of applications. Furthermore, the bugs in these applications are typically of a different type than those found in client programs, e.g. SQL injection attacks instead of buffer overflows. For web applications, the Arduilla system of Kizeun et al. performs symbolic execution of PHP code and can synthesize examples of SQL injection and cross site scripting attacks. One of the problems here is efficiently executing both client and server together, given that the two pieces have different languages and different runtimes.

Finally, for distributed applications we are not aware of any tool that performs symbolic execution on a whole distributed application. A starting point here would be a distributed trace and replay tool, such as Friday [37]. One of the key difficulties here is accounting for the effects of concurrency and timing, either through performing deterministic replay of the entire distributed application or by developing a whitebox fuzz testing technique tolerant of these effects.

5.1.2 The Unreasonable Effectiveness of SMT Solvers

Our work depends on the existence of a solver capable of quickly answering queries generated by our symbolic execution tools. Fortunately, our results show that today's solvers are up to the challenge. Furthermore, every other recent work on dynamic test generation has found that today's solvers are capable of quickly solving individual queries in the majority of cases, even though overall solving time may remain a large proportion of the tool's time.

This excellent performance is a paradox, because constraint satisfaction from symbolic execution is undecidable in general, and at least NP-complete if restricted to programs guaranteed to terminate in polynomial time. Today's solvers are unreasonably effective at the special class of instances arising from dynamic test generation. Why should this be so?

With Mark Winterrowd, we have begun a preliminary investigation of this phenomenon. Our method is to generate a large corpus of queries from multiple programs using the SmartFuzz tool. For each query in the corpus, we test the query solving time for several different solvers. This gives us an understanding of which queries cause solvers to take a long time, which in turn points the way toward features of the query that correlate with a long or short solving time.

5.1.3 Learning and Hot Spot Prioritization

A key problem in our generational search is how to choose the next test case for symbolic execution. We saw in Chapter 2 that a simple heuristic based on code coverage did not lead to a major improvement in the number of bugs discovered in our test programs. We can formulate this as a *reinforcement learning problem*, in which a learning algorithm adaptively makes choices of test cases to symbolically execute and receives feedback in the form of which test cases lead to the discovery of new bugs and which do not. Each trace obtained by our tools has multiple features that could inform the choices of a learning algorithm.

For example, recent work in empirical software engineering shows that bugs cluster in hot spots, code containing an unusually high proportion of bugs. Such hot spots reveal themselves by looking at code metrics or at previous discovery spots of security critical bugs [75]. Common sense states that if particular code is a bug hot spot, then we should spend more resources on testing that code. We can also use specific machine learning techniques, such as interpretable regression, to test this common sense.

Supposing for the moment that we do want to prioritize the testing of such hot spots, however, the best way to do this with whitebox fuzz testing is not clear. As one idea, we could give preference to generating tests that cover code paths in the hot spot. We could also use heavier weight static analysis techniques on the hot spot, or invest in generating summaries of the hot spot code [38].

Once we have built a specific search strategy, we can simply run it on a code base in parallel with the “vanilla” random choice heuristic described in Chapter 2. Then we can measure the difference in bugs found. This approach for evaluation allows for a "meta-search" over different heuristics until the best heuristic for a given suite of test cases is found.

A key theoretical question here is how to interpret the results observed from such experiments. First, without a convincing model of how bugs are distributed in the software under consideration, we cannot meaningfully define a notion of "statistical significance." Second, even if we have a model we believe characterizes some large class of software, how do we know this model will capture bugs in future software under consideration? The way to answer these questions may be in using a basket of heuristics that are adaptively weighted based on initial experience with the software under test, but this needs future research.

5.1.4 State explosion

The problem of picking the next test case to symbolically execute is just a symptom of a deeper problem: the state explosion in the analysis of any reasonably sized software. Summaries, as introduced by Godefroid, are one key technique for addressing this problem [38]. In addition, recent work in this direction has looked at synthesizing loop invariants and using them to simplify the set of constraints to explore [80].

5.1.5 When Should We Stop Testing?

A long standing question for all forms of software testing is when the cost of continuing to test outweighs the benefit from finding more bugs. Put another way, when should we stop testing? Dalal and Mallows framed this question as a *size-dependent sequential search* problem, in which each bug has a different probability of being found, or "size" (i.e.

more easily found bugs are “larger”) [27]. They then derive the optimal stopping rule under certain assumptions about the distribution of bug sizes.

The data generated by our experiments could be used to evaluate this framework in the context of whitebox fuzz testing. If we take the fuzzy hash from Chapter 4 as our measure of a bug, then the number of test cases falling into each hash bucket gives us a natural notion of the observed size for each bug. We could then derive for each test run the cutoff according to the Dalal and Mallows rule, then run for, say, twice that number of tests.

The resulting empirical data would help us measure the *regret* for each test run, or the amount of value lost if we follow the stopping rule. For example, we might define regret to be the number of bugs we would have missed by following the stopping rule on the same test run. We might also try to take into account the fact that there are multiple programs and multiple test inputs at hand, so the question is how to best allocate resources jointly across all programs and seed files. In any case, if the regret is consistently low, this would provide empirical evidence in favor of the given stopping rule.

5.1.6 Cooperative Whitebox Fuzz Testing

We saw in Chapter 2 that Whitebox fuzz testings e?ectiveness depends critically on the choice of seed input. Where should we look to find “good” seed inputs? Inspired by Liblit et al.’s Cooperative Bug Isolation [63], it would be interesting to build a system that uses lightweight instrumentation to identify promising candidates for such seed inputs from machines in the wild. Ideally, such instrumentation would be lightweight enough to be on by default. Instead of focusing only on inputs that cause program crashes, this approach looks for inputs that exercise code previously untouched, or exercise new paths through hot spot code. Because a users inputs can contain sensitive information, it may be enough to simply send back the path condition exercised by the input, as previously applied for crash-inducing inputs [19].

5.1.7 Streaming Algorithms and Whitebox Fuzzing

We argued in Chapter 4 that to be efficient with large programs, a tools memory usage must scale sublinearly in the size of the program trace. The algorithms community has developed a rich set of techniques for such "streaming algorithms," which may yield algorithms for dynamic test generation. The key insight is that the stream of x86 instructions in a trace matches the model of such streaming algorithms, where the algorithm looks at each element once and must keep a sublinear (ideally constant) amount of storage.

In addition, we may apply lower bounds for streaming algorithms to characterize which extensions of dynamic test generation can and cannot be e?ciently implemented. Such an analysis has been carried out for network intrusion detection properties, which suggests these techniques could also be applicable to the whitebox fuzz testing case [62].

5.2 Beyond Bug Finding

For many pieces of software, both whitebox and blackbox fuzz testing can quickly find more bugs in that software than anyone could hope to fix in a lifetime. Therefore, the bottleneck in managing such bugs shifts to reducing the amount of human time required to generate and test a patch for a security bug. While there is a rich line of work in automatically generating "vulnerability signatures" given an example input triggering a bug, this work is primarily aimed at worm defense [25] Therefore there is little attention paid to making the generated patch easy to maintain or understand. As a first step, we should be able to check that a programmers candidate patch catches inputs that previously exhibited the underlying bug.

5.3 Final Thoughts

The smart testing service outlined in this thesis was not practical in 2003, when the author began his PhD. Two major shifts occurred in the intervening six years. First, SMT solvers matured and targeted program analysis as a key problem domain. Second, the rise of on-demand computing makes available large amounts of computing power for short-term engagements, such as testing a single build of software. It is no longer crazy for security and programming language researchers to consider projects that require storing information about every run of a program, large amounts of storage, and enormous amounts of computation for even a single analysis run. Furthermore, these projects can now be used by others without requiring massive fixed capital costs. These trends will enable a new style of security research, based on automatic reasoning and analysis, which we can transfer more quickly to the rest of the world. We need only rise to the challenge.

Bibliography

[1] D. Aitel. The advantages of block-based protocol analysis for security testing, 2002. `http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html`.

[2] Marjan Aslani, NGA CHUNG, Jason Doherty, Nichole Stockman, and William Quach. Comparison of blackbox and whitebox fuzzers in finding software bugs, November 2008. TRUST Retreat Presentation.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*, 2004. `http://www.cs.wisc.edu/wpis/papers/cc04.ps`.

[4] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[5] Clark W. Barrett, Leonardo Mendonça de Moura, and Aaron Stump. Smt-comp: Satisfiability modulo theories competition. In Kousha Etessami and Sriram K. Raja-

mani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23. Springer, 2005.

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Tests from Counterexamples. In *Proceedings of ICSE'2004 (26th International Conference on Software Engineering)*. ACM, May 2004.

[7] S. Bhansali, W. Chen, S. De Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments VEE*, 2006.

[8] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[9] blexim. Basic integer overflows. *Phrack*, 0x0b, 2002.

[10] blexim. Basic integer overflows. *Phrack*, 0x0b(0x3c), 2002. `http://www.phrack.org/archives/60/p60-0x0a.txt`.

[11] D. Brumley, T. Chieh, R. Johnson, H. Lin, and D. Song. RICH : Automatically protecting against integer-based vulnerabilities. In *NDSS (Symp. on Network and Distributed System Security)*, 2007.

[12] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[13] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman,

and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[14] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[15] C. Cadar and D. Engler. EGT: Execution generated testing. In *SPIN*, 2005.

[16] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of SPIN'2005 (12th International SPIN Workshop on Model Checking of Software)*, volume 3639 of *Lecture Notes in Computer Science*, San Francisco, August 2005. Springer-Verlag.

[17] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI 2008*, 2008.

[19] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 319–328, New York, NY, USA, 2008. ACM.

[20] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities

in debian linux. In *PLAS - Programming Languages and Analysis for Security*, 2007. `http://www.cs.berkeley.edu/~daw/papers/fmtstr-plas07.pdf`.

[21] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Microsoft Corporation. AppVerifier, 2007. `http://www.microsoft.com/technet/prodtechnol/windows/appcompatibility/appverifier.mspx`.

[23] MITRE Corporation. Vulnerability type distributions in cve, 22 may 2007., 2007. `http://cve.mitre.org/docs/vuln-trends/index.html`.

[24] Symantec Corporation. Symantec vulnerability threat report 2008, 2009.

[25] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, , and P. Barham. Vigilante: End-to-end containment of internet worms. In *Symposium on Operating Systems Principles (SOSP)*, 2005.

[26] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of ICSE'2005 (27th International Conference on Software Engineering)*. ACM, May 2005.

[27] S. R. Dalal and C. L. Mallows. When should one stop testing software. *j-J-AM-STAT-ASSOC*, 83(403):872–879, ???? 1988.

[28] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume

4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[29] D. Deharbe and S. Ranise. SAT solving for software verification. In *Proc. of IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation.*, September 2005.

[30] J. DeMott. The evolving art of fuzzing. In *DEF CON 14*, 2006. `http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.odp`.

[31] Will Drewry and Tavis Ormandy. Flayer. In *WOOT : The Workshop on Offensive Technologies*, 2007.

[32] Bruno Duertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T)? In *Computer Aided Verification - CAV*, 2006.

[33] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of WODA'2003 (ICSE Workshop on Dynamic Analysis)*, Portland, May 2003.

[34] C. Flanagan. Hybrid type checking. In *Proceedings of POPL'2006 (33rd ACM Symposium on Principles of Programming Languages)*, January 2006.

[35] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, August 2000.

[36] V. Ganesh and D. Dill. STP: A decision procedure for bitvectors and arrays. CAV 2007, 2007. `http://theory.stanford.edu/~vganesh/stp.html`.

[37] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, pages 285–298, Cambridge, MA, 04/2007 2007. USENIX, USENIX.

[38] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.

[39] P. Godefroid and N. Klarlund. Software Model Checking: Searching for Computations in the Abstract or the Concrete (Invited Paper). In *Proceedings of IFM'2005 (Fifth International Conference on Integrated Formal Methods)*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32, Eindhoven, November 2005. Springer-Verlag.

[40] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.

[41] P. Godefroid, M. Y. Levin, and D. Molnar. Active Property Checking. Technical report, Microsoft, 2007. MSR-TR-2007-91.

[42] P. Godefroid, M. Y. Levin, and D. Molnar. Active Property Checking. Technical report, Microsoft, 2007. MSR-TR-2007-91.

[43] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, San Diego, February 2008. `http://research.microsoft.com/users/pg/public_psfiles/ndss2008.pdf`.

[44] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering (FSE)*, 2006.

[45] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proceedings of PLDI'2006 (ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation)*, Ottawa, June 2006.

[46] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.

[47] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

[48] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *Proceedings of PLDI'2002 (2002 ACM SIGPLAN Conference on Programming Language Design and Implementation)*, pages 69–82, 2002.

[49] Y. Hamadi. Disolver: the distributed constraint solver version 2.44, 2006. `http://research.microsoft.com/~youssefh/DisolverWeb/disolver.pdf`.

[50] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 125–138, Berkeley, January 1992.

[51] Sean Heelan. Automatic exploit generation: Lessons learned

so far, 2009. http://seanhn.wordpress.com/2009/07/05/automatic-exploit-generation-lessons-learned-so-far/.

[52] Sam Hocevar. zzuf, 2007. http://caca.zoy.org/wiki/zzuf.

[53] M. Howard. Lessons learned from the animated cursor security bug, 2007. http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx.

[54] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 668–674. Springer, 2009.

[55] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: Amplifying the effectiveness of software testing. Technical report, UC-Berkeley, April 2007. UCB/EECS-2007-35.

[56] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[57] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

[58] Nevis Labs. Nevis labs threat report 2008, 2008.

[59] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In *Software Engineering for Secure Systems, 2007. SESS '07:*

ICSE Workshops 2007, 2007. `http://idea.sec.dico.unimi.it/~roberto/pubs/sess07.pdf`.

[60] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium*, Washington D.C., August 2003.

[61] D. LeBlanc. Safeint 3.0.11, 2008. `http://www.codeplex.com/SafeInt`.

[62] Kirill Levchenko, Ramamohan Paturi, and George Varghese. On the difficulty of scalably detecting network attacks. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 12–20, New York, NY, USA, 2004. ACM.

[63] Benjamin Robert Liblit. *Cooperative bug isolation*. PhD thesis, UC Berkeley, Berkeley, CA, USA, 2004. Chair-Aiken, Alexander.

[64] LMH. Month of kernel bugs, November 2006. `http://projects.info-pull.com/mokb/`.

[65] R. Majumdar and K. Sen. Hybrid Concolic testing. In *Proceedings of ICSE'2007 (29th International Conference on Software Engineering)*, Minneapolis, May 2007. ACM.

[66] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[67] Microsoft Corporation. Prefast, 2008.

[68] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliabil-

ity of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[69] D. Molnar, X.C. Li, and D. Wagner. Dynamic test generation to find integer bugs in X86 binary linux programs. In *USENIX Security Symposium*, 2009.

[70] D. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors, 2007. UC Berkeley EECS, 2007-23.

[71] H.D. Moore. Month of browser bugs, July 2006. `http://browserfun.blogspot.com/`.

[72] G. J. Myers. *The Art of Software Testing.* Wiley, 1979.

[73] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Languages Design and Implementation (PLDI)*, 2007.

[74] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[75] Stephan Neuhaus, Thomas Zimmermann, and Andreas Zeller. Predicting vulnerable software components. Technical report, Universitt des Saarlandes, Saarbrcken, Germany, February 2007.

[76] Rob O'Callahan. Chronicle-recorder, 2008. `http://code.google.com/p/chronicle-recorder/`.

[77] J. Offutt and J. Hayes. A Semantic Model of Program Faults. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 195–200, San Diego, January 1996.

[78] Pex. Web page: `http://research.microsoft.com/Pex`.

[79] Protos. Web page: `http://www.ee.oulu.fi/research/ouspg/protos/`.

[80] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. Technical report, UC-Berkeley, March 2009. UCB/EECS-2009-34.

[81] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.

[82] J. Seward and N. Nethercote. Using valgrind to detect undefined memory errors with bit precision. In *Proceedings of the USENIX Annual Technical Conference*, 2005. `http://www.valgrind.org/docs/memcheck2005.pdf`.

[83] A. Sotirov. Windows animated cursor stack overflow vulnerability, 2007. `http://www.determina.com/security.research/vulnerabilities/ani-header.html`.

[84] Spike. Web page: `http://www.immunitysec.com/resources-freesoftware.shtml`.

[85] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04 (International Symposium on Software Testing and Analysis)*, Boston, July 2004.

[86] M. Vuagnoux. Autodafe: An act of software torture. In *22nd Chaos Communications Congress, Berlin, Germany*, 2005. `autodafe.sourceforge.net`.

[87] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Network Distributed Security Symposium (NDSS)*, 2009.

[88] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.