View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Central Florida (UCF): STARS (Showcase of Text, Archives, Research &...

STARS

University of Central Florida
STARS

HIM 1990-2015

2013

Implementation and testing of a blackbox and a whitebox fuzzer for file compression routines

Toby Tobkin
University of Central Florida

Part of the Electrical and Computer Engineering Commons

Find similar works at: https://stars.library.ucf.edu/honorstheses1990-2015
University of Central Florida Libraries http://library.ucf.edu

This Open Access is brought to you for free and open access by STARS. It has been accepted for inclusion in HIM 1990-2015 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

Recommended Citation

Tobkin, Toby, "Implementation and testing of a blackbox and a whitebox fuzzer for file compression routines" (2013). *HIM 1990-2015*. 1475.
https://stars.library.ucf.edu/honorstheses1990-2015/1475

IMPLEMENTATION AND TESTING OF A BLACKBOX AND A WHITEBOX FUZZER FOR FILE COMPRESSION ROUTINES

by

TOBY J. TOBKIN

A thesis submitted in partial fulfillment of the requirements
for the Honors in the Major Program in Computer Science
in the College of Engineering and Computer Science
and in The Burnett Honors College
at the University of Central Florida
Orlando, FL

Spring Term 2013

Thesis Chair: Dr. Ratan Guha

© 2013 Toby Tobkin

ABSTRACT

Fuzz testing is a software testing technique that has risen to prominence over the past two decades. The unifying feature of all fuzz testers (fuzzers) is their ability to somehow automatically produce random test cases for software. Fuzzers can generally be placed in one of two classes: black-box or white-box. Blackbox fuzzers do not derive information from a program's source or binary in order to restrict the domain of their generated input while white-box fuzzers do. A tradeoff involved in the choice between blackbox and whitebox fuzzing is the rate at which inputs can be produced; since blackbox fuzzers need not do any "thinking" about the software under test to generate inputs, blackbox fuzzers can generate more inputs per unit time if all other factors are equal.

The question of how blackbox and whitebox fuzzing should be used together for ideal economy of software testing has been posed and even speculated about, however, to my knowledge, no publically available study with the intent of characterizing an answer exists. The purpose of this thesis is to provide an initial exploration of the bug-finding characteristics of blackbox and whitebox fuzzers. A blackbox fuzzer is implemented and extended with a concolic execution program to make it whitebox. Both versions of the fuzzer are then used to run tests on some small programs and some parts of a file compression library.

DEDICATION

For Dr. Ratan Guha, who gave me advice and direction for my education that I could not do without.

ACKNOWLEDGMENTS

I wish to acknowledge those who helped me to finish my thesis. I would like to thank Dr. Ratan Guha for undertaking me as his undergraduate student, for directing me in my research and work, and for helping me to overcome the barriers and failures that I encountered along the way. I also would like to express my gratitude towards Sharif Hassan for his guidance and support. Finally, I would like to thank Dr. Paul Dombrowski and Dr. Mostafa Bassiouni for their feedback and contributions as committee members.

TABLE OF CONTENTS

Chapter 1: Introduction 1

Motivation 2

Thesis Organization 3

Chapter 2: Background and Related Work 4

Background on Fuzzing and Automated Randomized Testing 4

Concolic Execution 6

Literature Review 8

Chapter 3: Testing Software Using A Blackbox And Whitebox Fuzzer 11

Experimental Design 11

Architecture of This Experiment 11

Choosing Fuzzers 12

Controlling For Multiple Crashes from One Bug 13

Blackbox Fuzzer Design 14

Software Under Test Monitor Design 17

Whitebox Fuzzer Design 19

Experiments 22

Small Benchmarks 23

Large Benchmarks 26

Chapter 4: An Exploratory Economic Analysis 32

The Full Process Used For Blackbox And Whitebox Fuzzing 32
Modelling the Process of Blackbox and Whitebox Fuzzing 34
Configuring The Software Under Test To Accept Arbitrary Input 34
Writing The Fuzz Test Driver 36
Symbolically Instrumenting Software (Whitebox Fuzzing Only) 38
Fuzzing Software 41
Tabulating Unique Software Bugs 41
A Complete Model 43
Chapter 5: Conclusions And Discussion 49
Appendix A: Source Code 51
fuzzer.hpp 52
fuzzer.cpp 54
Works Cited 64

LIST OF FIGURES

Figure 1: Symbolic execution tree corresponding to program above 7

Figure 2: High-level design of the blackbox fuzzing session 16

Figure 3: High-level design of the software under test monitor component........ 18

Figure 4: High-level architecture of the whitebox fuzzer 21

Figure 5: Graph of crashes triggered over time for uniform_test.c 24

Figure 6: Gantt charts of the blackbox and whitebox fuzzing processes........ 33

Figure 7: Labor costs of configuring software under test to accept arbitrary input 36

Figure 8: Labor costs of writing fuzz test drivers 37

Figure 9: Labor costs of symbolically instrumenting software under test........ 40

Figure 10: Labor costs of symbolically instrumenting software under test........ 40

Figure 11: Labor costs of tabulating unique software bugs........ 43

List of Tables

Table 1: Labor costs of configuring software under test to accept arbitrary input 35

Table 2: Labor costs of writing fuzz test drivers 37

Table 3: Labor costs of tabulating unique software bugs 43

CHAPTER 1: INTRODUCTION

Real-world software applications are often very complex in nature, and in practice these applications nearly always contain bugs in their implementation. This is more-or-less unavoidable because software is often implemented by many different humans, and typically represents a much larger system than can have all of its details fully conceptualized at once.

Given this, some method of eliminating errors in software implementation is usually considered necessary to deliver a reliable software product. The primary method used in practice today is software testing [1]. However, software testing labor intensive, and mostly done by human beings doing things such as manually writing unit tests. Because of these two factors, software testing is very expensive, and it is common for software testing costs to be 50% or more of a software project's total expenses [1].

The high costs mentioned have provided substantial motivation for the utilization of more automated testing techniques to reduce costs. One of the techniques that has risen to prominence in recent years is often called "fuzzing." The exact definition of fuzzing is often stretched, but in general fuzzing entails giving a piece of software a large number of randomly generated test cases—usually several orders of magnitude more test cases than a human can write manually. Blackbox fuzzing refers to randomly generating inputs for software under test with little or no consideration of the software under test's inner workings. Even though it is a relatively unintelligent technique in nature to utilize, blackbox fuzzing's brute force methodology has found success finding bugs in some major software projects [2].

Although fuzzing often produces useful testing results, its effectiveness can be improved substantially by letting it "know" more about the software under test. Much research effort as of late

has been done with this very goal in mind; it is felt that by making the "dumb" blackbox fuzzer "smarter," or more whitebox, that even more software bugs can be found without human interaction.

In this research, this topic of making a blackbox fuzzer smarter and more whitebox is investigated. A blackbox fuzzer is implemented in C++ that can provided a very large amount of randomly mutated inputs to a program being tested. Following this, the blackbox fuzzer is loosely integrated with a concolic analysis engine that knows how to produce concrete inputs to exercise large numbers of reachable C program branches. Using the standalone blackbox fuzzer and the fuzzer integrated with the concolic analysis engine, a compression library is tested.

Motivation

An issue often brought up by those familiar with the topic of Fuzzing is what kind of tradeoff should be made between rate of test case generation and the amount of computational analysis given to test cases on average [1]. If much computational time is given to analyzing the target program to generate test cases, each test case may on average be good at finding a bug in the program, but it is possible that not enough test cases will be generated to fully exercise the code covered, or it may be possible that only parts of the program that can be effectively analyzed will be tested due to limitations of current whitebox fuzzing techniques available to the public [2].

What seems to be absent from current research on fuzz testing is a thorough and scientific evaluation of available techniques on a common problem set. One group of Berkeley students attempted this in 2008 [3], but acknowledged that they were "not able to make a solid conclusion" because no data on timing was kept.

Thesis Organization

In Chapter 2, background information on software testing and fuzzing will be provided to aid the reader in understanding the research presented. In Chapter 3, a comparison of fuzz testing techniques is presented, including justification for experimental design choices. In Chapter 4, conclusions are drawn from the results obtained in Chapter 3 and discussed.

Chapter 2: Background and Related Work

Background on Fuzzing and Automated Randomized Testing

Fuzz testing is a conceptually simple yet undeniably potent automated black-box testing technique for software first described by Miller et al. in 1989 [4]. Like many technology metaphors, the word "fuzz" reflects an intuitive analogy to its definition; fuzz testing traditionally refers to giving a program a large amount of completely random input (fuzz) while monitoring for program failures of various types. Despite its appearance as a "dumb" approach to software testing, simple black-box fuzz testing has surprised and is continuing to surprise software testers and researchers with its effectiveness in a wide variety of domains including testing Unix utilities, finding security flaws in Adobe Reader X, and discovering limitations of malware detection engines [4] [5] [6] [7]. Eventually, fuzz testing rose to prominence as both a popular testing tool, as seen in the Month of Browser Bugs [8], and as a part of several software companies secure development lifecycles, including Microsoft's [9], Adobe's [10], and Cisco's [11].

However, it is rather obvious that black-box fuzz testing has serious limitations. Consider the following C-style code fragment:

```
if (x == 357)
    function1();
else
    function2();
```

Assuming x is a 32-bit integer, fuzzing x with purely random values would almost always cause `function2()` to be executed. Conversely, `function1()` has an underwhelming 1 in 2^{32} chance of being tested in a given instance because x must equal exactly 357 for that branch to be taken. Cases such as these, which are common, are especially problematic when doing random testing of highly structured inputs such as file formats. In such a scenario, an overwhelming majority of cases

may fail early in the execution path because of failed checksums or improper values in headers that may be dependent on each other.

Automated black-box testing solutions to this problem of overwhelming rejection of inputs have been proposed and implemented, but at the expense of the ideal of testing being truly automated. Since the solutions require testers to build descriptions of the file format being tested, this type of testing often proves cost prohibitive, especially in scenarios in which no description of the file format is published or requires reverse engineering. An analogous situation exists for network protocols.

This issue that exists without the previously mentioned labor expenditure of building input descriptions, commonly referred to by saying a technique exhibits poor or shallow code coverage, has not gone unnoticed by researchers. Several well-known papers in the past ten years have made contributions to what is now called "white-box fuzzing," leading the definition of fuzzing somewhat astray from its purely random roots [9] [10]. Leveraging both static and dynamic code analysis techniques, these papers have demonstrated some cases where more sophisticated white-box fuzzing techniques have a substantial advantage over black-box fuzzing without the upfront labor cost of designing a grammar or structure for a black-box fuzzer. Of particular note, Microsoft developed and internally deployed the whitebox fuzzer SAGE [12], which they claim has made their software testing substantially more productive.

Despite the endorsement of Microsoft and others, however, an informal survey of published guides and workshops on the Internet reveals little evidence that white-box fuzzing techniques have gained traction with software testers; searches of Black Hat, Infosec, Microsoft, and Defcon turned up little mention of using publically available tools for white-box fuzz testing. Black-box tools such as The Peach Fuzzing Platform [13] and the now rather dated SPIKE continue to be

the focus, with the exception of formal research. This is in spite of the fact of the demonstrated advances in white-box testing in terms of both effectiveness and automation.

Concolic Execution

Concolic execution is the execution of a program both concretely and symbolically [14]. "Concolic" is a combination of the words "concrete" and "symbolic." *Concrete execution* is simply the execution of a program using a specific input. *Symbolic execution* [15] involves solving conditions for branching in general.

Symbolic execution requires that the symbolic values of program variables and path conditions be stored for each branch in a program [16]. Exploring all of the possible branches of a program is accomplished by creating and systematically negating path conditions, and then using a constraint solver such as Z3 [17], inputs are generated that guarantee exercising of each path.

Concrete and symbolic execution can be well explained by using a simple example adapted from the paper describing Symbolic Java Pathfinder by Pasareanu et al. at NASA [16]. Consider the following code fragment and corresponding symbolic execution tree:

```
[1] int x, y, result;
[2]    if (x > y)
[3]      result = x - y;
[4]   else
[5]      result = y - x;
[6] assert (result > 0)
```

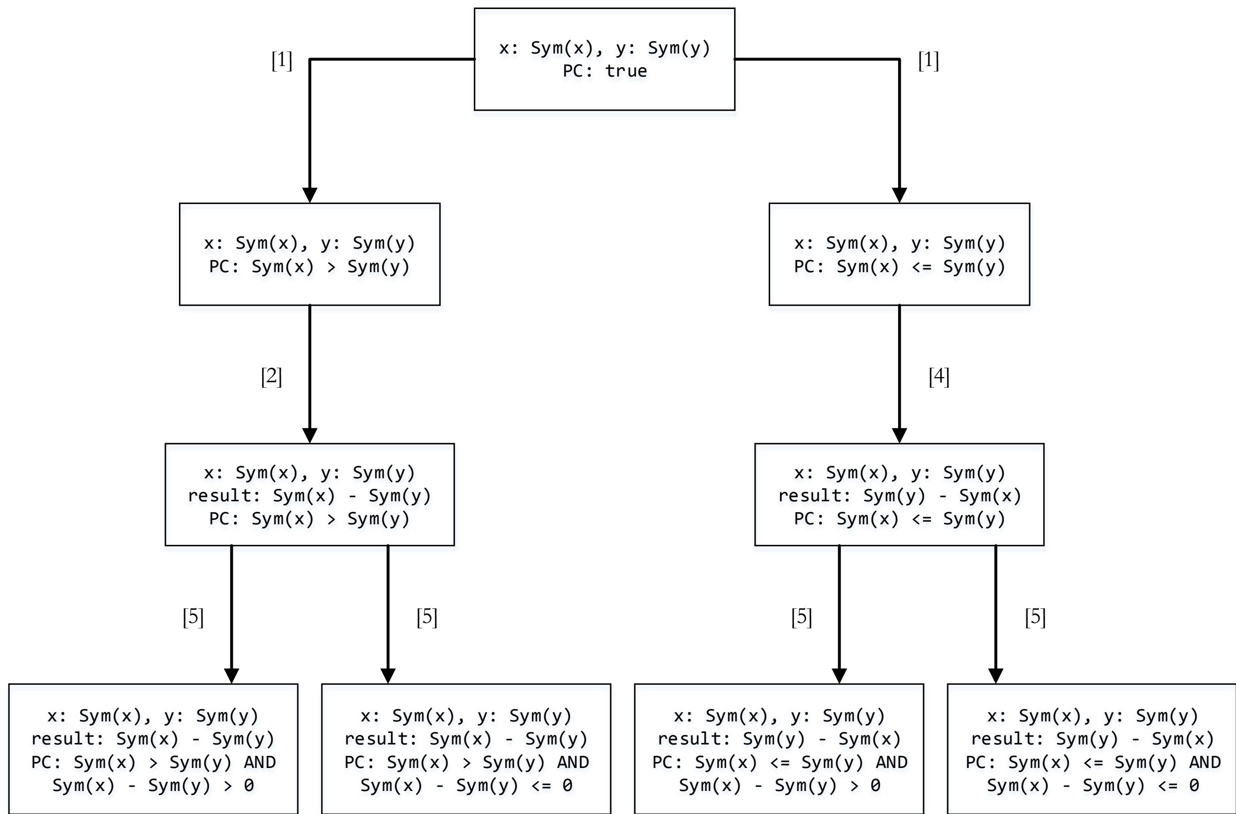


Figure 1: Symbolic execution tree corresponding to program above

On a given concrete execution, only one particular leaf of the possible executions in the tree will be reached. If, for instance, x was set to be 5 and y was set to be 3, path 1 would be exercised. The major advantage of concrete execution over symbolic execution is that no potentially computationally expensive constraint analysis is needed to be performed to exercise the given path. The disadvantage of concrete execution is that in practice, it may be unevenly cover code, for instance, in the situation of hand-writing unit tests or doing blackbox fuzzing.

The symbolic execution generalizes the conditions upon which each branch is taken, and the generalized path constraints are shown in the boxes. At each branch, the generalized values of variables are used to generate the path constraints necessary to take one side of the branch. The path condition is negated and solved for to take the other branch. The advantage of symbolic execution is great—it can often generate test cases that cover all possible branches of a program. However, it has two major drawbacks. First, it can be difficult to symbolically reason about items such as floating point values and memory pointers [18]. Second, symbolic execution quickly becomes a computationally intractable problem as the number of branches in a program increases [19]. The number of unique execution paths in a program can grow exponentially with the number or branches, or can even grow without bounds in the case of some loops. This is often called path explosion or state explosion [18].

Literature Review

Since the inception of the idea of fuzz testing in the late 80's at the University of Wisconsin-Madison, the key motif in research has been to improve the idea of randomized (fuzz) testing by effectively making it less random. The first large step towards this goal has been the creation of

grammar-assisted blackbox fuzzers such as the aforementioned SPIKE and Peach Fuzzing Platform. However, grammar-assisted fuzzers also often have large labor costs associated with their operation.

Research has been done to mitigate the costs associated with doing grammar-assisted blackbox fuzzing, a notoriously difficult task for some undocumented protocols; for instance, Samba, which provides file and print services for various Windows clients using Unix-like operating systems, took over a decade to develop [20] because of the lack of information on the protocols it utilized. Cabello et al. developed Polygot in 2007 as one of the first steps towards reducing these costs [21]; instead of focusing on packet data, Polygot derived information from network protocols via dynamic analysis of the binaries using the protocol. Though effective, one significant criticism of this method is that it only revealed a "flat" structure of the protocol being analyzed rather than its truer hierarchical one. Lin et al. proposed an improvement to this in 2008 by noting that different protocol fields in a given message are normally handled in different execution contexts [22].

However, if Microsoft's success with white-box fuzzing is a good indicator, the future lies with this approach. Whitebox fuzzing has often been based on combining fuzz testing, concrete execution, and symbolic execution into one tool [18]. Research in the mid- and late-2000's strengthened the prospects of using concrete and symbolic (concolic) execution in practice for software testing by addressing limitations of the technique such as exponential path explosion [23] and imprecise pointer reasoning [24], enabling some fuzzers to exercise a large number of execution paths of programs while still being automatic with their input generation.

Overall, the present situation seems to be that fully randomized blackbox fuzz testing and concolic whitebox fuzz testing represent extreme points on a spectrum of computation required per test case generated. The former does no analysis and applies all computational resources to writing new test cases. The latter devotes a substantial proportion of required CPU time to analyzing the

tested program in an attempt to maximize the number of execution paths exercised with its fuzzed inputs. However, Ganesh et al. have developed a whitebox fuzzer called BuzzFuzz that represents an intermediate between the two aforementioned techniques [25]. Although both concolic fuzzers and BuzzFuzz derive information about variables from a program's source and concrete runs, BuzzFuzz does so in a simpler, less computationally intensive way. Concolic fuzzers maintain logical expressions for each variable that may become very complex, while BuzzFuzz simply relates sets of input bytes to program variables using a technique called "taint analysis." Although not as well suited for exhausting all possible execution paths, BuzzFuzz's techniques may be more effective in some scenarios because it can generate test cases at a higher rate than concolic testers.

CHAPTER 3: TESTING SOFTWARE USING A BLACKBOX AND WHITEBOX FUZZER

Experimental Design

Architecture of This Experiment

The experiment consists of two separate fuzzing sessions with as many factors controlled as possible besides the method of generating fuzzer seed inputs. Both the blackbox and whitebox fuzzing sessions are run for a set number of iterations. Each iteration consists of the fuzzer running the software under test on a particular fuzzed input, the fuzzer monitoring the software under test for a crash, and finally recording data on what happened. Every iteration, the following data is recorded:

- The cumulative number of iterations performed by that point in time
- The cumulative number of crashes encountered by that point in time
- The wall clock time elapsed

Additionally, upon a fuzzed input causing the software under test to crash, the following information is recorded in addition to the previously mentioned data:

- The signal terminating the program
- The input that caused the program to be terminated
- The file from which the input terminating the program was derived

Choosing Fuzzers

Certain requirements must be met to ensure the best possible accuracy of results. First, the fuzzers chosen should represent the state of the art as closely as is possible. Second, some method of controlling for duplicate bug discoveries must be used.

Choosing software that best represents the state of the art is a process that mostly concerns the whitebox fuzzer. The ideal whitebox fuzzer for this experiment will have implemented more of the research advances in whitebox fuzzing than other available software. Determining which particular fuzzer this is entails tabulating all of the available whitebox fuzzers, performing a literature review of advances in whitebox fuzzing, and evaluating which of the fuzzers implements the most of these advances. Other restrictions such as being able to accurately keep time data on when bugs are found and licensing also must be considered.

Choosing a blackbox fuzzer is less involved since the basic methodology of blackbox fuzzing has, by its definition, remained static relative to whitebox fuzzing. The only requirement of the blackbox fuzzer is that it does not experience any unnecessary performance bottlenecks.

With regards to choice of both the blackbox and the whitebox fuzzer, a preliminary restriction is that the software is available for academic use. There exists commercial software fitting the definition of a whitebox fuzzer, however the software is not advertised as being available for academic use. There have also been whitebox fuzzers written as proofs of concept for particular research publications, but that have not been released publically [25]. Finally, what is perhaps the most advanced whitebox fuzzer in existence, and for which much of the research contributing to advances in whitebox fuzzing technology was funded for, Microsoft's SAGE is only used internally at Microsoft.

The ability to log data on when bugs were found is also necessary. Without this ability, it is not possible to characterize what bug-finding over time looks like using different fuzzing techniques. This turns out to be an important restriction because one of the most technically advanced concolic testing tools available for academic use, Microsoft Pex, was not designed with this capability, and cannot be modified to be able to keep track of time data because of its closed source nature.

For this experiment, custom built "one-off" blackbox fuzzer is used for two reasons. First, adapting a pre-built fuzzer to keep timing data was judged to take more effort than simply building the logic into a new fuzzer. Bottlenecks in I/O performance were also able to be avoided by only keeping track of data necessary for the experiment. Written in native C++, it spends minimal time calculating input mutations, effectively randomly generating input at maximum speed. Second, custom-writing the blackbox fuzzer allowed more parallels in design to be drawn between the blackbox and whitebox fuzzing techniques, allowing for a better comparison between the two.

A custom whitebox fuzzer design was also used for this experiment. The ability to do concrete and symbolic analysis was a must for this experiment, which, it appears, narrows the candidate software to Microsoft Pex and JPF-Concolic. Neither of these two testing engines supported gathering timing information, so a simple extension to CREST was written instead.

Controlling For Multiple Crashes from One Bug

It is possible, and often likely, that multiple crashes of the program under test will be associated with the same bug in the program's source code. Consider the following C# code fragment:

In the code fragment above, any time the argument "position" exceeds the size of the array "array," the program will crash because array boundaries are checked at runtime in C#. If the size of

```
static int getElement(int position)
{
   return array[position];
}
```

the array is 5, then the sequence of inputs {2, 6, 8, 1, 3, 10, 12} will trigger 4 program crashes. However, all 4 of these crashes were caused by a single bug: not checking to see if `position` was within the array's boundaries.

Since the number of bugs discovered by a fuzzer is a better metric of performance than the number of crashes it causes, it is useful to have a method of counting these 4 crashes as only 1 bug discovery. This makes sense because bugs cause program crashes, not the other way around.

The method used for controlling for multiple crashes caused by a single bug is hashing the stack traces of terminated programs. A script is used to automatically process all of the core dumps produced from the software under test in a particular fuzzing session and determine which stack traces are the same. This information is used to change the data on crashes over time into data on unique bugs found over time.

In practice, however, this technique was never employed in the experiments conducted because both fuzzing techniques failed to cause program crashes.

Blackbox Fuzzer Design

The custom blackbox fuzzer used in this experiment is traditional in design in the sense that it can provide purely random input to a compiled target software under test, although if needed it could do random mutations of an input file. For performance reasons, a lightweight software under test monitor is integrated into the design of the fuzzer so that an external program such as OllyDbg does not need to be used. The program monitor is discussed further in the next section.

In short, the blackbox fuzzer takes input data from all of the files in a specified directory. Then, for each input file, a set of fuzzed inputs is derived and provided to the software under test. During each execution of a fuzzed input, the software under test is monitored for operating system kill signals. Depending on whether or not the software under test is terminated, appropriate information is recorded by the test monitor component of the fuzzer. A high-level design diagram illustrating the blackbox fuzzing process follows.

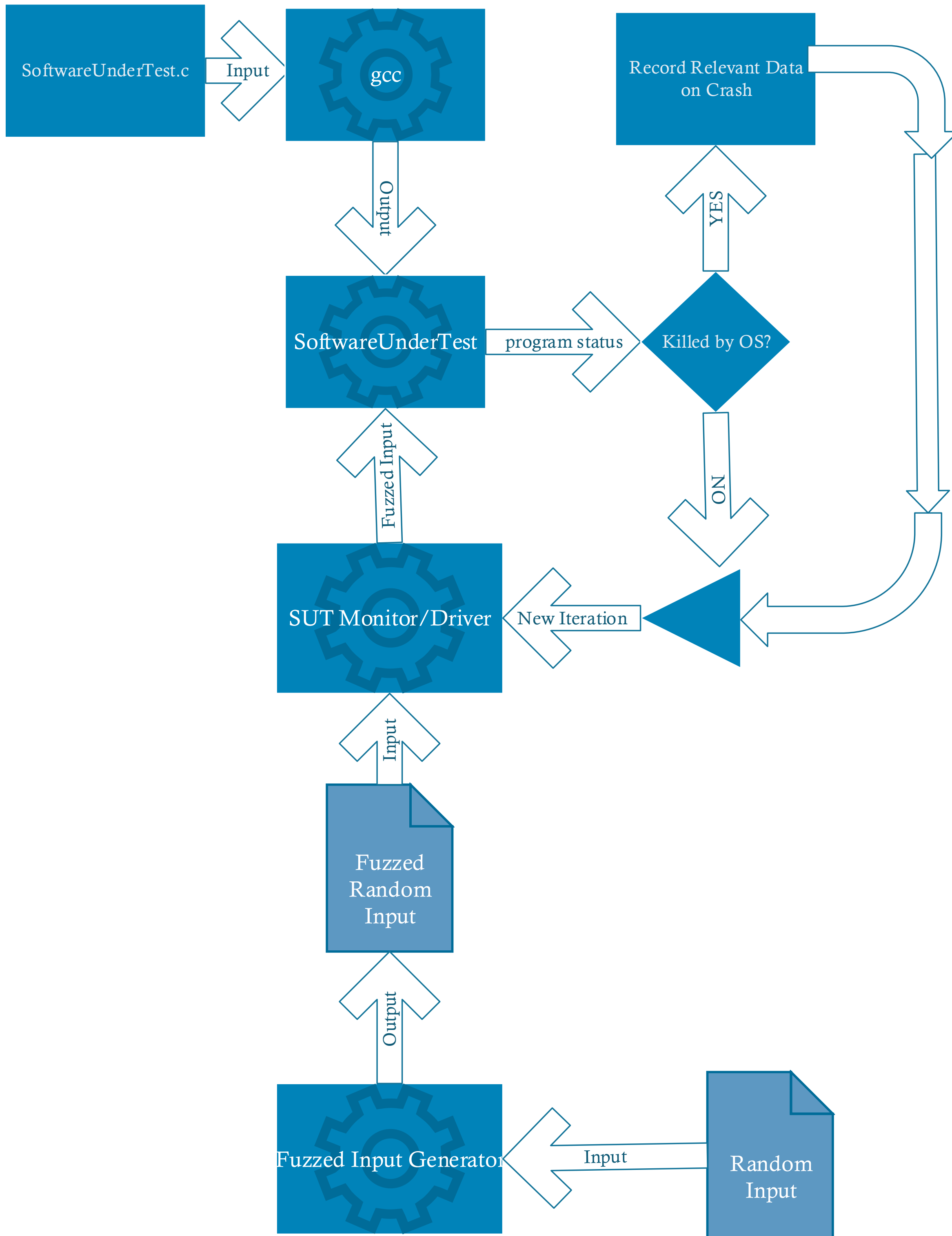


Figure 2: High-level design of the blackbox fuzzing session

Software Under Test Monitor Design

The software under test monitor (SUT monitor) was a component integrated into both the blackbox and whitebox fuzzer. A primary design goal of the SUT monitor was to solve the following two problems faced by the Berkeley undergraduates in their fuzzing research [5]:

1. Lack of accurate data on the time taken to trigger each individual crash
2. Poor storage performance

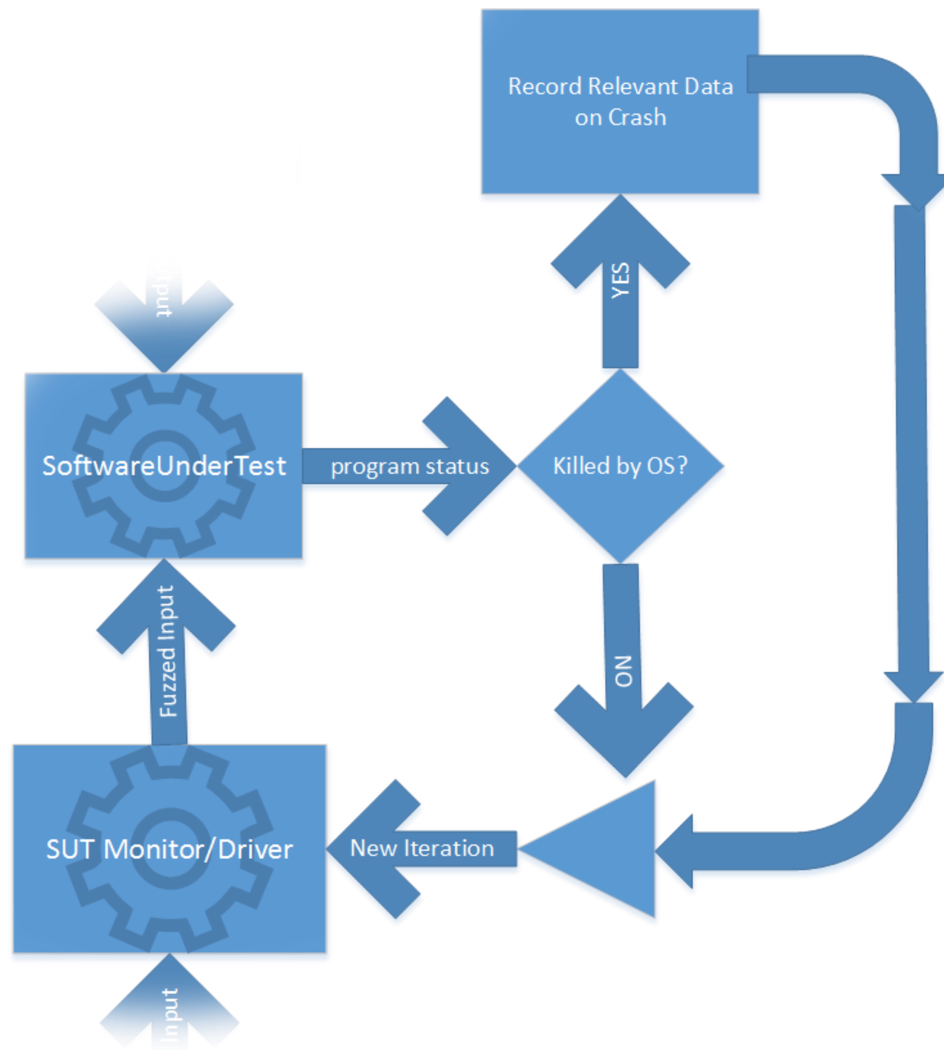


Figure 3: High-level design of the software under test monitor component

To solve the problem described in item 1 above, the SUT monitor places the current wall clock time associated with a crash into the log file output buffer in addition to any information needed to reconstruct the crash later. This information can later be aggregated to determine information such as, but not limited to, the rate at which crashes occur, the rate at which bugs are

found, or the susceptibility of the software under test to being crashed by a fuzzed input in terms of approximate clock cycles per crash.

The problem described in item 2 above is approached by using the paradigm of accessing the hard disk as infrequently as possible. This is done at the architectural level of the entire fuzzing process, but some elements of this paradigm are reflected in the implementation of the test monitor. To this end, only enough information is recorded to forensically reconstruct a crash later on if one occurs. The idea here is to only record information that is absolutely necessary to minimize the amount of writing done to the disk. Referring to figure 3, this can be seen by noting that the "Killed by OS?" branch skips a process in the common case of the software under test not crashing.

Several other optimizations could have been done substantially further improve the problem described in item 2, but they were left for a later project.

Whitebox Fuzzer Design

The whitebox fuzzer used in this research was designed as a simple extension to a concolic testing tool, CREST-Z3 [26]. CREST-Z3 was selected as the tool to be extended for several reasons. First, it fulfills the requirement that it can reason about the software under test in a (mostly) automated fashion. Second, to my knowledge, CREST has not been studied for its usefulness substantially outside of the initial research paper it was introduced in; another paper that used CREST could not be found by a rudimentary search using academic search tools, and the number of downloads that CREST has on its hosting site indicate that it has a very small user base. Third, in the paper that it was introduced in, CREST was not applied to fuzzing, making the experiments here somewhat novel.

The overall idea in the design of the extension is that although CREST's concolic testing produces excellent code coverage, it does not exercise the covered code very well because of the nature of the techniques employed. Adding a fuzzing stage to CREST provides advantages over both purely concrete fuzzing and also the concolic testing techniques that CREST uses. The advantage over blackbox fuzzing is that, because seed input represents valid input exercising a wide variety of branches, this extension to CREST may be able to trigger bugs in code that would otherwise be unreachable through blackbox fuzzing. The advantage over using CREST by itself is that sometimes more test input values will be tried on a given branch. This provides a substantial advantage over using CREST alone because CREST does not attempt to reason about parts of C programs that often cause errors such as array indices and arithmetic not important to branching. A simple example of these advantages are shown in the "uniform_test" experiment in the next section.

A high-level overview of the entire whitebox fuzzing process used can be seen in the diagram that follows. Basically, after the software under test is hand instrumented, and then compiled by CREST, CREST is used to generate high-coverage input files to the software under test. These input files are given to the fuzzer, which fuzzes them in memory and provides the results to the software under test as input. Identically to the blackbox fuzzer, the whitebox fuzzer then monitors the software under test for a crash and records appropriate data.

Components of the whitebox fuzzing process that are in addition to the blackbox fuzzing process are highlighted in green in the diagram.

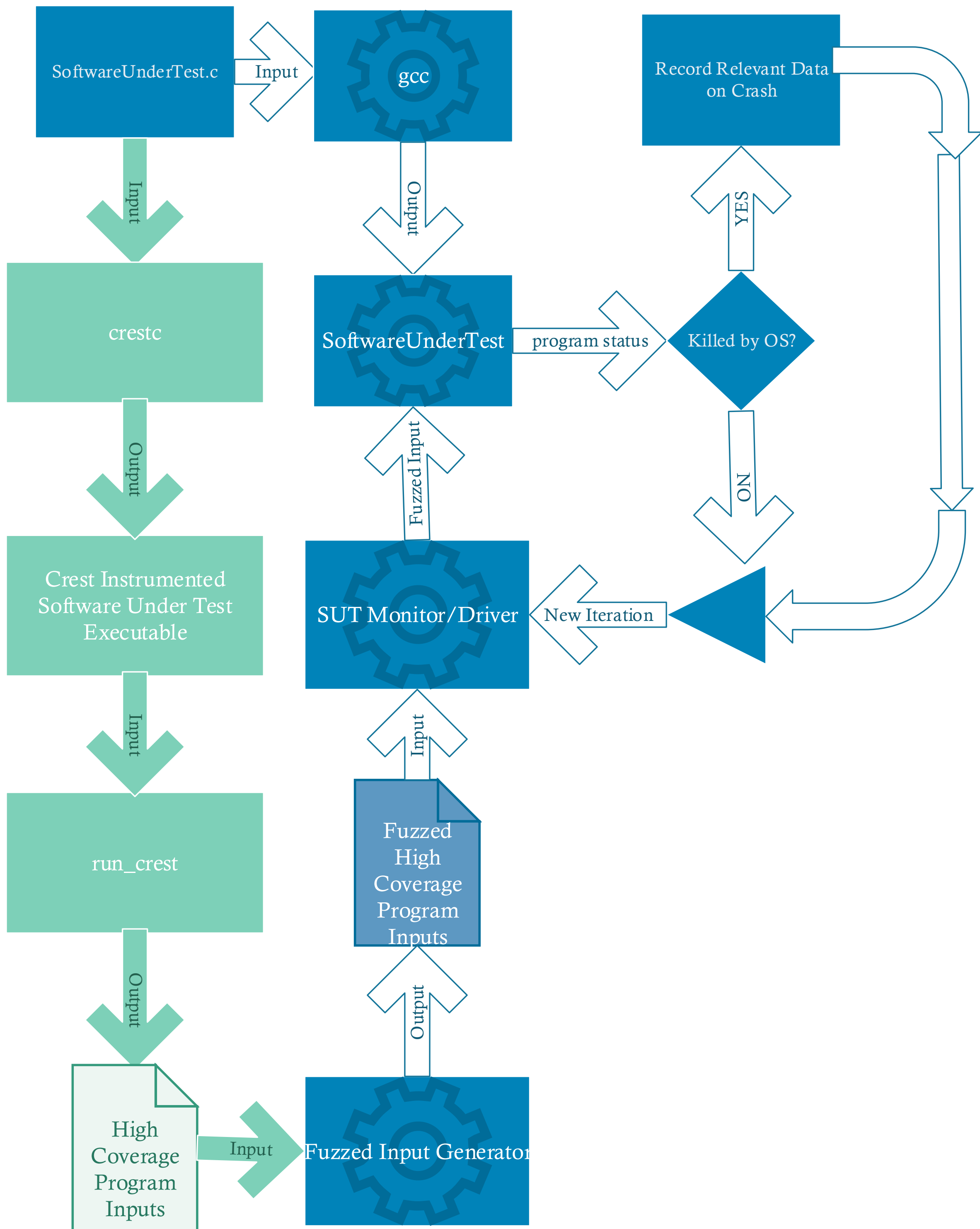


Figure 4: High-level architecture of the whitebox fuzzer

However, this technique has a limitation that stems from its simplicity. This technique makes the assumption that slightly changing a set of input values will not simply cause frequently exercised branches to be exercised more instead of further exercising lesser-used branches. However, this assumption could very well be false in the general case if, for instance, the software under test is given a file with a checksum as input. A better implementation would have been to integrate the fuzzed input generation procedures with CREST's branch coverage routines, but this would have been a far larger undertaking.

Experiments

Experiments are divided into two sections: small benchmarks and large benchmarks. Small benchmarks consists of easily understandable, small programs that demonstrate some characteristic of the testing techniques utilized in this research. Small benchmarks are not intended to be realistic measures of testing technique performance. Large benchmarks are benchmarks are real-world programs that better represent the complexity of typical, useful software. Large benchmarks should be too complex to realistically automatically cover using any available techniques. Notably, this means that all branches should not be able to be covered using only CREST, the fuzzer, or a combination of the two.

Small Benchmarks

uniform_test

uniform_test.c is a small program designed for the purpose of demonstrating a branching structure that is impossible to explore with a blackbox fuzzer, but can be easily explored with a concolic whitebox fuzzer such as the one used here. If variables a, b, c, and d are chosen randomly, as they are with the blackbox fuzzer, the most nested print statement will execute during approximately 1 in every $2^{(32*4-2)} = 2^{126}$ executions, or approximately once every$7.9 * 10^{27}$ years at this benchmark's execution rate of 340 times per second.

uniform_test.c contains one purposefully included bug at its deepest branch. The entire program, including instrumentation to work with CREST, is shown in the code segment below:

```
#include <crest.h>
#include <stdio.h>
int main(void) {
	int a, b, c, d, e;
	CREST_int(a);
	CREST_int(b);
	CREST_int(c);
	CREST_int(d);
	CREST_int(e);

	if (a > 5 && a < 10) {
		if (b == 19) {
			if (c == 7) {
				if (d == 3) {
					printf("Some values will cause a crash.\n");
					d =  d / (e % 10);
				}
			}
		}
	}
	if (a < 9) {
		printf("No problem here.\n");
	}
	return 0;
}
```

Although good for demonstrative purposes, uniform_test.c does not necessarily represent a realistic software under test scenario. The program is small, makes use of no external libraries, and has a very obvious bug in it. Nonetheless, the fuzzing results are presented for comparison. Both fuzzers were run for 70,000 iterations.

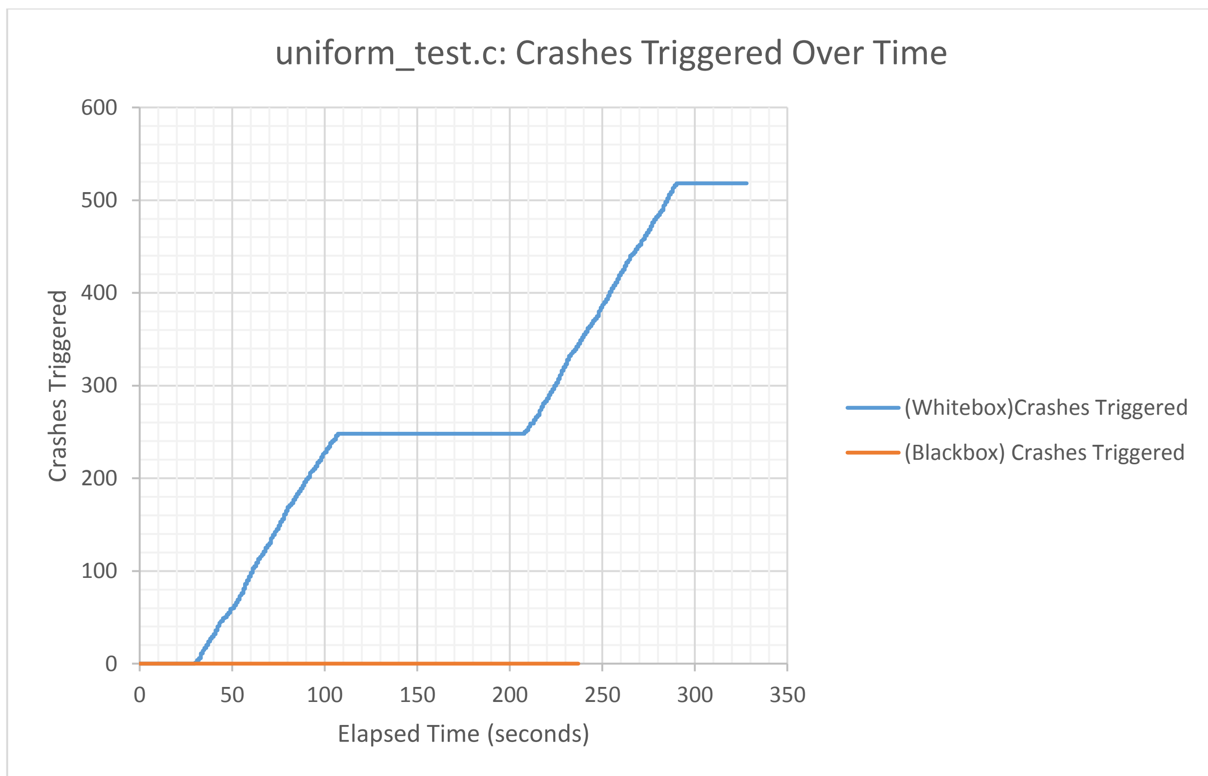


Figure 5: Graph of crashes triggered over time for uniform_test.c

The results indicate that the whitebox fuzzer was able to trigger the planted bug very often while the blackbox fuzzer was unable to trigger the bug, as expected by the preceding analysis. Although unrealistic as mentioned, this example is what established the plausibility of concolic whitebox fuzzing being more efficient than blackbox fuzzing early on in this research.

The behavior of the "(Whitebox) Crashes Triggered" curve is notable. Flat sections indicate times when the fuzzer was using input files that covered branches far from the bug. Increasing sections indicate that an input file that covered the most nested branch with the bug contained was being fuzzed.

nonlinear_solution.c

The reason why CREST was extended to use the Z3 theorem prover instead of the one it was originally designed with is that Z3 can handle solving nonlinear equations that CREST's original theorem prover could not reason about. nonlinear_solution.c was written as a small test simply to verify that Z3 functioned. nonlinear_solution.c is shown in the code segment that follows.

```
#ifdef CREST
#include <crest.h>
#endif
#include <stdio.h>
#include <stdlib.h>

//takes x as first argument, y as second argument
int main(int argc, char *argv[]) {

#ifdef CREST
	int x, y;
	CREST_int(x);
	CREST_int(y);
#else
	int x = atoi(argv[0]);
	int y = atoi(argv[1]);
#endif

	if ((y * y) % 50 == x) {
		if (x > y + 10) {
			y = y / 0; //bug
		} else {
			printf("No bug on this line\n");
		}

		printf("Success.\n");
	}
	exit(EXIT_SUCCESS);
}
```

CREST was able to identify all 4 branches in this small program and was able to concolically produce output that exercised all 4 branches.

Large Benchmarks

Several pieces of software were considered for testing in these experiments. Linux standard utilities have been commonly exercised in automated software testing research [1] [27], making them an obvious first choice, however the labor involved in instrumentation, verifying the correctness of the instrumentation, and configuration of these utilities became too labor intensive for the scope of this research. In this respect, grep, ul, tsort were considered. A JPEG compressor was also considered for use, but its heavy use of floating point arithmetic made it a poor candidate for using static analysis on, which in our case does not possess very good reasoning abilities with regards to floating point calculations. Miniz, a compression library was eventually selected to be used as a benchmark.

Miniz

Miniz is an open source compression library. It was selected as a benchmark because it possessed a large number of branches, could be easily modified to take input straight from memory instead of through files slowly written to the hard disk, and because it could be configured for instrumentation and compiling within a reasonable amount of time—approximately 10 hours.

Two of Miniz's features are tested: its ability to compress and decompress a string, and its ability to parse a zip file. Because Miniz is not a standalone program, for each feature to be tested, a small driver program is written.

Traditionally, fuzzers of file processing programs such as Miniz produce a set of fuzzed files to be consumed by the file processing software under test. However, it was found that substantial fuzzing performance improvements could be made by having Miniz "read files" from memory instead of from disk, thus allowing millions of test cases to be produced in a short period of time. As will be seen, the included code for the test drivers reflect this design; no file reading functions are included and instead data already in memory is compressed and decompressed.

Miniz String Compression and Decompression: Experiment Setup

For both blackbox and whitebox testing, a test driver was used to process input and to pass fuzzed input to the appropriate functions in Miniz. For blackbox testing, the test driver took 70 characters as a single argument and passed the 70 characters to a string decompression function. During blackbox fuzzing, the fuzzer simply calls the compiled test driver with a random 70 character argument repeatedly. The resulting compressed string was then passed to a decompression function. For whitebox testing, the 70 characters in the argument were marked as symbolic using the `CREST_char(char x)` function. CREST then reasons about these 70 bytes in order to produce sets of inputs that will exercise as many program branches as possible. The input sets produced by CREST are then used as seeds by the fuzzer to produce a large amount of concrete test input for the test driver.

The test driver used for this experiment is shown in the code segment that follows. The only difference in code between the blackbox and whitebox test driver is that a `#define CREST` line is included above `int main(int argc, char *argv[])`.

Miniz String Compression and Decompression: Experiment Results

The blackbox fuzzing session produced 1,500,000 fuzzed inputs for the test driver. However, no crashes were triggered, and, as such, not much can be drawn from these results.

```c
typedef unsigned char uint8;
typedef unsigned short uint16;
typedef unsigned int uint;

#define INPUT_SIZE 70
#define ARGUMENT_TO_USE 0

#ifdef CREST
#include <crest.h>
#endif

int main(int argc, char *argv[]) {
    // copy the characters in
    char symIn[128];
    memcpy(symIn, argv[ARGUMENT_TO_USE], INPUT_SIZE);

#ifdef CREST
    //mark all input characters as symbolic
    int charIndex;
    for (charIndex = 0; charIndex < INPUT_SIZE; charIndex++)
        CREST_char(symIn[charIndex]);
#endif

    int src_len = strlen(symIn);
    uLong cmp_len = compressBound(src_len);
    unsigned char *pCmp, *pUncomp;

    // Allocate buffers to hold compressed and uncompressed data.
    pCmp = (mz_uint8 *) malloc((size_t) cmp_len);
    pUncomp = (mz_uint8 *) malloc((size_t) src_len);

    // Compress the string then decompress it
    uLong uncomp_len = src_len;
    compress(pCmp, &cmp_len, (const unsigned char *) symIn, src_len);
    uncompress(pUncomp, &uncomp_len, pCmp, cmp_len);

    printf("Success.\n");
    return EXIT_SUCCESS;
}
```

The whitebox fuzzing session produced 1,500,000 fuzzed inputs derived from 71 seed inputs produced by CREST, including the original seeds themselves. CREST recognized 2440 possible branches that could be taken along the execution path provided by the test driver. By themselves, the seeds exercised 452 of these branches. However, no crashes were triggered by any of the fuzzed inputs.

Although good quantitative data cannot be derived from this experiment, this experiment qualitatively indicates that this type of concolic fuzzing was not effective on this benchmark.

Miniz Zip File Decompression: Experimental Setup

Miniz's Zip file decompression logic was tested similarly to the string compression and decompression instance. A test driver was written that handled the fuzzed input and passed it to the appropriate function. For blackbox testing, the test driver took 20 bytes as a single argument and passed the 20 bytes to the Zip decompression function. During blackbox fuzzing, the fuzzer simply calls the compiled test driver with a random 20 byte argument repeatedly. For whitebox testing, the 20 byte argument to the Zip decompression function are instead marked as symbolic using the `CREST_unsigned_char(unsigned char x)` function. CREST then reasons about these 20 bytes in order to produce sets of inputs that will exercise as many program branches as possible. The input sets produced by CREST are then used as seeds by the fuzzer to produce a large amount of concrete test input for the test driver.

The test driver used for this experiment is shown in the code segment that follows. The only difference in code between the blackbox and whitebox test driver is that a `#define CREST` line is included above `int main(int argc, char *argv[]).`

```
#include <crest.h>
#define INPUT_SIZE 20
#define ARGUMENT_TO_USE 0

int main(int argc, char *argv[]) {
	unsigned char* compressed_data_ptr;
	unsigned char* output_ptr; //our "compressed data", i.e. fuzzed input
	compressed_data_ptr = (unsigned char *) malloc(INPUT_SIZE); //pointer to where
the compressed data is in memory
	const size_t out_buf_size = INPUT_SIZE * 10; //output buffer size
	output_ptr = (unsigned char *) malloc(out_buf_size);
	const size_t src_buf_size = INPUT_SIZE; //source buffer size
	const unsigned int flags = 0;

#ifdef CREST
	int i;
	for (i = 0; i < INPUT_SIZE; i++)
		CREST_unsigned_char(compressed_data_ptr[i]);
#endif

	//read the specified amount of data into the buffer
	memcpy(compressed_data_ptr, argv[ARGUMENT_TO_USE], INPUT_SIZE * sizeof
(unsigned char));

	tinfl_decompress_mem_to_mem(output_ptr, out_buf_size, compressed_data_ptr,
INPUT_SIZE, flags);

	printf("Success.\n\n");
	exit(EXIT_SUCCESS);
}
```

Miniz Zip File Decompression: Experimental Results

The blackbox fuzzing session produced 1,500,000 fuzzed inputs for the test driver. However, no crashes were triggered. As such, not much can be drawn from these results.

The whitebox fuzzing session also produced 1,500,000 fuzzed inputs, however, in operation, it was in effect the same as the blackbox fuzzing session. CREST recognized 2410 possible branches that could be taken along the execution path provided by the test driver. However, CREST was only able to symbolically solve for one 20 byte input that exercised 33 branches. Similar to the blackbox fuzzing session, the fuzzed inputs derived from the single input that CREST produced did not trigger any crashes.

Although good quantitative data cannot be derived from this experiment, the experiment does qualitatively demonstrate limitations of the techniques employed by CREST.

CHAPTER 4: AN EXPLORATORY ECONOMIC ANALYSIS

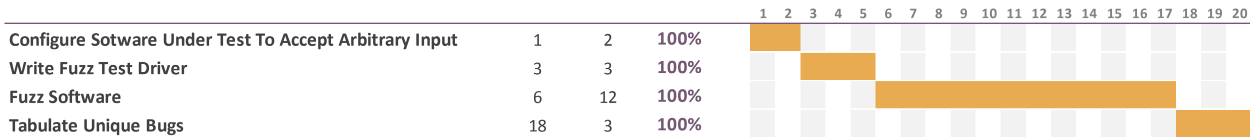
Since economics is the focus of this mathematical analysis, factors beyond those of computational efficiency are considered, different from the focus of the rest of this research. These economic factors turn out to be critical; in the United States, the cost of software engineering labor is very high. Depending on their exact classification, the Bureau of Labor Statistics reports that software developers typically make $44.85 to $49.30 per hour as of 2012 [28]. In order to have useful data on factors related to labor, my own experiences will be measured. This method, although not highly scientific, provides an interesting initial exploration of the problem examined.

The Full Process Used For Blackbox And Whitebox Fuzzing

This section will introduce the different steps involved in the blackbox and whitebox fuzzing process, and thus, the precise factors I will measure and do analysis on. As is mentioned elsewhere in this research, there are generally other tradeoffs associated with the use of a whitebox fuzzer over a blackbox fuzzer besides efficiency of test case generation. Whitebox fuzzers often also take a not-insignificant amount of additional labor to setup and use on a particular piece of software under test. Gantt charts comparing the blackbox and whitebox fuzzing process used in this research follows, not to scale to any quantitative data.

Blackbox Fuzzing

Task	Start	Duration	Complete
Configure Sotware Under Test To Accept Arbitrary Input	1	2	100%
Write Fuzz Test Driver	3	3	100%
Fuzz Software	6	12	100%
Tabulate Unique Bugs	18	3	100%



Whitebox Fuzzing

Task	Start	Duration	Complete
Configure Sotware Under Test To Accept Arbitrary Input	1	2	100%
Write Fuzz Test Driver	3	3	100%
Symbolically Instrument Software	6	15	100%
Fuzz Software	21	12	100%
Tabulate Unique Bugs	33	3	100%

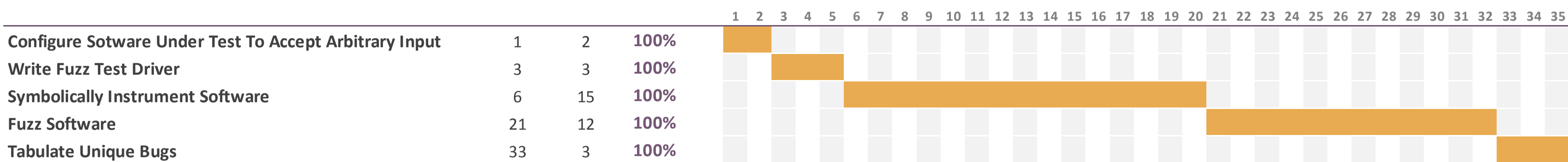


Figure 6: Gantt charts of the blackbox and whitebox fuzzing processes

As can be seen in the Gantt charts above, the primary difference in cost between blackbox and whitebox fuzzing in my experiments is the cost of symbolically instrumenting the software under test. This cost is large for a number of reasons. First, it requires manual analysis, and sometimes modification, of the software under test to determine where inputs are actually reduced to primitive data types that can be reasoned about with the concolic execution engine used. Second, verifying that the symbolic instrumentation worked properly takes a substantial amount of manual testing itself, and in my experience the failure rate was high; often even if the proper stack variables were marked as symbolic, CREST-Z3 would be unable to effectively reason about the program structures encountered.

Modelling the Process of Blackbox and Whitebox Fuzzing

In this section, the individual details of the blackbox and whitebox fuzzing process that was used in my research will be modeled. For each step shown in the Gantt charts, some tendencies of that process will be explained, and a reasonable model of this process will be chosen. At the end of the section, each constituent model will be combined to reason about the generalized simplified economic problem of finding software bugs.

Configuring The Software Under Test To Accept Arbitrary Input

A necessary goal of the fuzzer implementation completed in this research was that it needed to be higher performance than most common fuzzers because tests were done on limited computational resources. One of the primary performance optimizations was to, at a high level, design fuzzing routines to use the host computer's hard disk as little as possible. A primary ways this

was carried out was by making substitute "stub" methods for the software under test's file input/output operations. In the course of this research, this was done at least a dozen times.

Creating these stub methods amounted to the following process:

1. Identify all file input/output procedures relevant to the part of the software being tested
2. Replace any file input streams with an input stream that the fuzzer could use to provide random data to the program

This task, by nature, was both technical and tedious. A table of the approximate labor expenditures for programs of varying sizes in lines of code (LOC) is presented as follows:

Lines of Code	Labor Expenditures (Hours)
20	0.25
50	0.5
30	0.15
15	0.1
10000	3
26000	4
4900	3

Table 1: Labor costs of configuring software under test to accept arbitrary input

A scatter plot was produced using spreadsheet software of the data in the table. A qualitative graphical examination of the scatter plot indicated that the data was logarithmic in nature, and so a logarithmic curve was fitted to the data. The scatter plot and the fitted curve, $y = 0.5103\ln(x) - 1.4096$, follows:

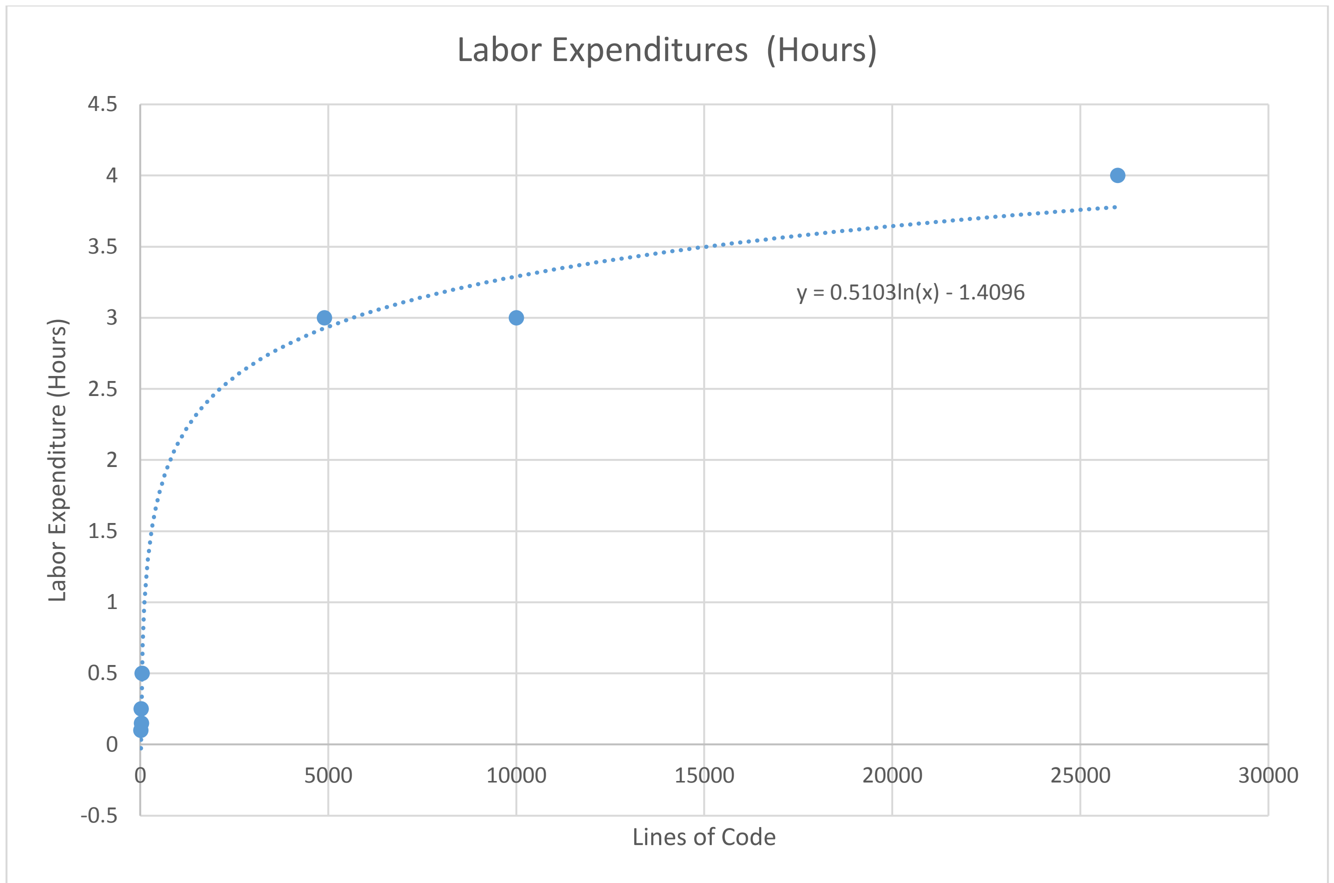


Figure 7: Labor costs of configuring software under test to accept arbitrary input

Writing The Fuzz Test Driver

For the purposes of this research, it can be assumed that the fuzz test driver does not need to be engineered from scratch. Although it is true that one was custom-engineered for this research, it only had to be slightly modified for each software under test. The process of writing the fuzz test driver is outlined as follows:

1. Compose the proper Linux execve() arguments to launch the software under test with fuzzed inputs

2. Make sure that the method's internal code reflects the correct data types being used in the arguments

Because the test driver needed the same process to be done each time, the time was constant for each software under test. A table and graph is provided for consistency:

Lines of Code	Labor Expenditures (Hours)
20	0.5
50	0.5
30	0.5
15	0.5
10000	0.5
26000	0.5
4900	0.5

Table 2: Labor costs of writing fuzz test drivers

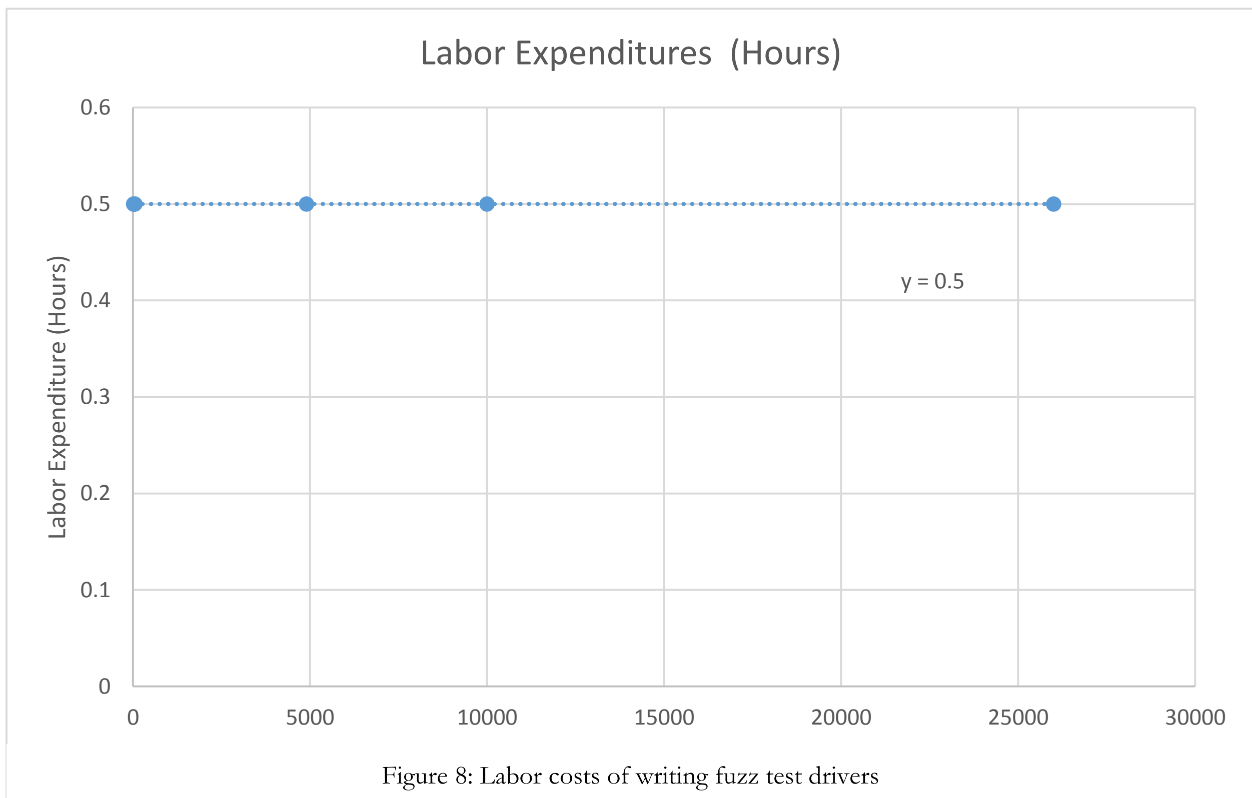


Figure 8: Labor costs of writing fuzz test drivers

As seen, a linear curve is fitted for consistency. It is simply y = 0.5 hours.

Symbolically Instrumenting Software (Whitebox Fuzzing Only)

Symbolic instrumentation of software represents the most difficult process covered here form both a math modeling standpoint, and a technical standpoint. As is mentioned in the section of this thesis on concolic execution, input data for a concolic-based whitebox fuzzer must be reasoned about symbolically. However, because of the innate computational intractability of this problem in general, and the relatively recent nature of the scientific developments of methods to cope with this intractability, only a small number of important inputs must be marked as symbolic to be reasoned about by the concolic execution engine. For a large variety of reasons that are beyond the scope of this research, evaluating every input symbolically and algorithmically would be impossible.

In the experiments presented in this research, a large degree of parallelism was kept between blackbox fuzzing sessions and whitebox fuzzing sessions so that they could be compared as scientifically as possible. Software under test inputs that would be randomized (fuzzed) in the blackbox session would be marked symbolic in the whitebox session. Refer to the following section for a simple example of how this was done using preprocessor directives: Chapter 3 > Experiments > Small Benchmarks > nonlinear_solution.c

There are three main challenges in designing a symbolically-marked version of a piece of software. First, for many programs, a large amount of analysis must go into designing a method of introducing a small amount of input that will not be rejected early in the stages of the program. This is not a “cookie-cutter” problem, and as such, a large degree of variation is seen in

how long programs take to instrument. The reasons for this are discussed in Chapter 2 > Background on Fuzzing and Automated Randomized Testing. Second, the size of the symbolically marked input must be optimized for. We want the symbolically marked input to be as large as possible, however, increasing the size too much either makes the symbolic execution intractable or crashes CREST-Z3. The latter factor could have been eliminated with substantially more work on a reengineered concolic execution engine. Third, even after symbolic execution is implemented for the software under test, often it is found to be in vein because the concolic execution engine cannot reason about the program structures tested, thus leaving the work so far as wasted. A full discussion of the limitations of current concolic execution and theorem proving techniques is beyond the scope of this research, and to my knowledge, would require a new literature review to be conducted on the topic because of the fast pace of change in this research area.

For the purposes of this paper, the latter challenge mentioned—that sometimes software cannot be analyzed using a concolic execution engine, wasting all the labor used thus far—will be termed **risk of concolic analysis failure (RCAF)**. This term is introduce to simplify this math modeling problem slightly. In reality, we simply cannot do the concolic analysis if it fails. However, we will approximate this by assuming we have lots of software to test, and that our expected labor expenditure is as follows:

$$Expected\ Labor\ Expenditure = \frac{Average\ Labor\ Expenditure}{1 - Risk\ of\ Concolic\ Analysis\ Failure}$$

RCAF will be left as a variable in the complete model because it will vary greatly depending on the type of software being tested. For instance, numerical analysis routines would have an RCAF approaching 0, but small, integer calculation programs would have an RCAF close to 1. In the

course of my experiments, my RCAF was approximately 0.5. My observed RCAF of 0.5 is reflected in the following table and graph:

Lines of Code	Expected Labor Expenditures (Hours)
20	0.3
50	0.4
30	0.2
15	0.2
10000	30
26000	50
4900	20

Figure 9: Labor costs of symbolically instrumenting software under test

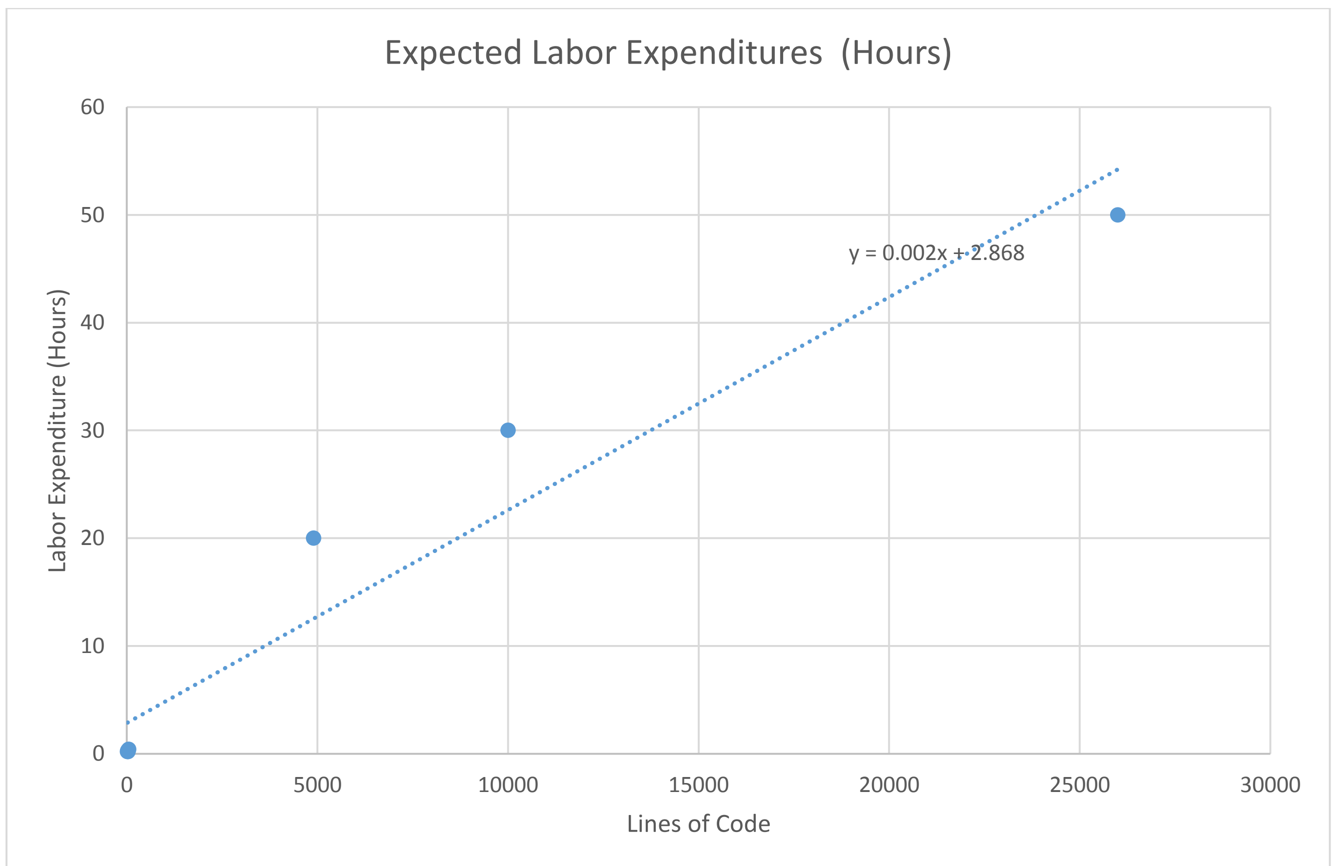


Figure 10: Labor costs of symbolically instrumenting software under test

A linear trendline was chosen to reflect the fact that increased program complexity (in terms of lines of code) will increase the amount of labor needed for symbolic instrumentation. Factoring out the included RCAF, the equation found for the trendline is:

$$Expected\ Labor\ Expenditure = 0.001 * \left(\frac{LOC}{1 - RCAF}\right) + 2.868$$

Fuzzing Software

Fuzzing the software is an automatic part of the entire process described here. As such, it has no labor cost associated with it. For this reason, I will simply say that it has associated with it some small **capital cost** of either owning or renting the computers that will run the fuzzing sessions. In an organization doing regular software testing, these machines will likely be constantly utilized, and thus not useful for other activities. Capital cost will simply be specified as a variable in the complete model.

Tabulating Unique Software Bugs

In my experience, tabulation of unique software bugs could be done mostly automatically. I will not go into the technical details, but effectively, a very professional fuzzing setup will be able to use certain fuzzer optimizations, crash dumps, development utilities, and scripting to determine the uniqueness of each item in a very large set of crashes. In the case of this research, the aforementioned system was custom-engineered due to the lack of a suitable available alternative. For the purposes of modelling, we will assume that such a system is available prebuilt.

Using such a system to obtain a "list" of unique bugs took me constant time in practice. That is, it took me approximately two hours to use the automatic tools built earlier to produce this "list" in each fuzzing session. A chart and graph is provided for consistency:

Lines of Code	Labor Expenditures (Hours)
20	2
50	2
30	2
15	2
10000	2
26000	2
4900	2

Table 3: Labor costs of tabulating unique software bugs

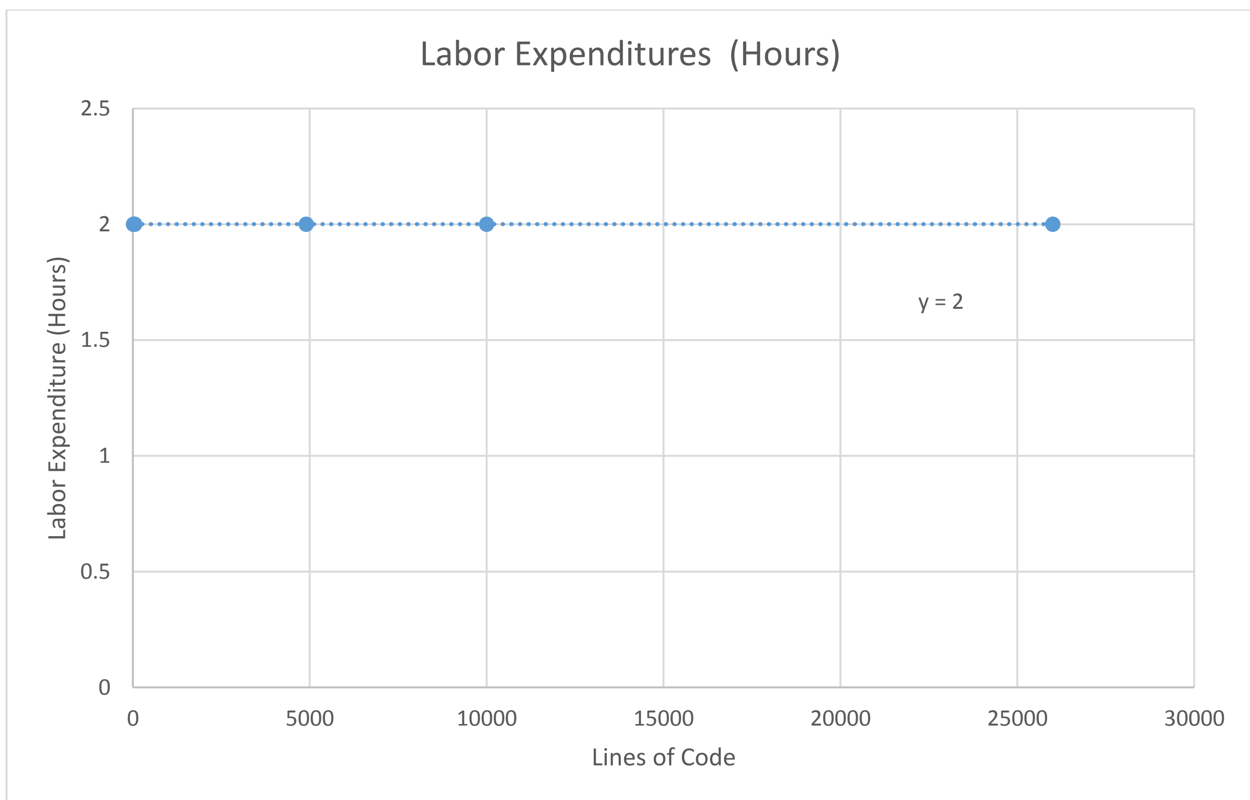


Figure 11: Labor costs of tabulating unique software bugs

A Complete Model

Our final model will be in terms of cost, typically the most important metric in a commercial software engineering environment. At a high level, this will be done by simply adding the labor costs

of the previously described processes and multiplying by the cost-per-hour of a software engineer to do the work.

There is a large amount of transferable labor overlap between the blackbox and whitebox fuzzing routines; indeed, the only difference between the two is that the whitebox fuzzing routine described here requires the additional step of symbolically instrumenting the software under test. As such, I believe that the most interesting problem to approach is the following: Given cost estimates of both blackbox and whitebox fuzzing routines, is the work differential associated with whitebox fuzzing an acceptable cost?

To answer the latter question, the complete cost model of both blackbox and whitebox fuzzing will be created. Following this, they will be related using an inequality. **Cost of labor** (COL) will be defined as the hourly wage of a software engineer performing the work.

Blackbox Model

The complete model of the cost of blackbox fuzzing is the sum of the following items, all of which were discussed in the preceding sections:

1. COL multiplied by the number of hours of labor needed to configure the software under test to accept arbitrary input
2. COL multiplied by the number of hours of labor needed to write the fuzz test driver
3. The capital cost associated with being able to run the fuzzing session
4. COL multiplied by the number of hours of labor needed to tabulate the unique software bugs

According to the analysis done in the preceding sections, in equation form, the mathematical model the monetary cost of blackbox fuzzing is the following, where ExpectedCost is the function defining the expected cost of software testing in US Dollars:

$$ExpectedCost(LOC, COL, Capital\ Cost)$$
$$= item1 + item2 + item3 + item4$$
$$= COL * (0.5013 * \ln(LOC) - 1.4096) + COL * (0.5) + (Capital\ Cost) + COL * (2)$$
$$= COL * (0.5013 * \ln(LOC) + 1.0905) + (Capital\ Cost)$$

Using this model, consider the example of testing the file decompression library used as a benchmark in Chapter 3 > Experiments > Large Benchmarks > Miniz exactly as it was in the research. In this case, LOC is 4900, COL is (reasonably) assumed to be $47.00, and Capital Cost is assumed to be $25. These values produce the result $276.45 using the model above. Thus, it is estimated to cost $276.45 to discover however many bugs are found using this technique.

Whitebox Model

The complete model of the cost of whitebox fuzzing is the sum of the following items, all of which are the same as in the blackbox model except item 3:

1. COL multiplied by the number of hours of labor needed to configure the software under test to accept arbitrary input
2. COL multiplied by the number of hours of labor needed to write the fuzz test driver
3. COL multiplied by the number of hours of labor needed to symbolically instrument the software under test
4. The capital cost associated with being able to run the fuzzing session

5. COL multiplied by the number of hours of labor needed to tabulate the unique software bugs

According to the analysis done in the preceding sections, in equation form, the mathematical model of the monetary cost of whitebox fuzzing is the following, where ExpectedCost is the function defining the expected cost of software testing in US Dollars:

$$
\begin{aligned}
&ExpectedCost(LOC, COL, Capital\ Cost, RCAF) \\
&= item1 + item2 + item3 + item4 + item5 \\
&= COL * (0.5013 * \ln(LOC) - 1.4096) + COL * (0.5) + \mathrm{COL} * \left(0.001 * \left(\frac{\mathrm{LOC}}{1 - \mathrm{RCAF}}\right)\right. \\
&\qquad\qquad \left. + 2.868\right) + (Capital\ Cost) + COL * (2) \\
&= COL * \left(0.5013 * \ln(LOC) + 0.001 * \left(\frac{\mathrm{LOC}}{1 - \mathrm{RCAF}}\right) + 1.7775\right) + (Capital\ Cost)
\end{aligned}
$$

Using this model, consider the example of testing the file decompression library used as a benchmark in Chapter 3 > Experiments > Large Benchmarks > Miniz exactly as it was in the research. In this case, LOC is 4900, COL is again assumed to be \$47.00, Capital Cost is assumed to be \$25, and RCAF is assumed to be 0.5. These values produce the result \$769.34 using the model above. Thus, it is estimated to cost \$769.34 to discover however many bugs are found using this technique, almost three times as much as blackbox fuzzing.

Cost Differential Model

Now that we have models for how much blackbox and whitebox fuzzing, it would be useful to have a model for knowing the increase in cost for doing whitebox fuzzing in addition to blackbox fuzzing. Such would be an important cost to consider if, for instance, blackbox fuzzing did not produce useful enough results and a project manager was considering implementing the more

rigorous, but expensive, whitebox fuzzing routine. A predicted number of bugs to be found would also need to be found, as would the expected cost of these bugs to the organization producing the software, but these factors represent a far more complicated problem that will not be explored here.

Assuming that both a blackbox and whitebox fuzzing session is ran, whitebox fuzzing items 1 and 2 have already been completed, and do not need to be factored into the increased cost. However, the costs associated with items 4 and 5 will be reincurred. Item 3 is not completed in the blackbox session. Given this, the cost of running a whitebox fuzzing session after having already completed a blackbox fuzzing session on a given piece of software under test is the sum of the following:

1. COL multiplied by the number of hours of labor needed to symbolically instrument the software under test
2. The capital cost associated with being able to run the fuzzing session
3. COL multiplied by the number of hours of labor needed to tabulate the unique software bugs

In equation form, based on the analysis in the preceding two sections, this increase in cost is modeled by the following equation:

$$ExpectedCost(LOC, COL, Capital\ Cost, RCAF)$$

$$= item1 + item2 + item3$$

$$= \mathrm{COL} * \left(0.001 * \left(\frac{\mathrm{LOC}}{1 - \mathrm{RCAF}}\right) + 2.868\right) + (Capital\ Cost) + COL * (2)$$

$$= \mathrm{COL} * \left(0.001 * \left(\frac{\mathrm{LOC}}{1 - \mathrm{RCAF}}\right) + 4.868\right) + (Capital\ Cost)$$

Using this model, consider the example of testing the file decompression library used as a benchmark in Chapter 3 > Experiments > Large Benchmarks > Miniz exactly as it was in the

research. In this case, LOC is 4900, COL is again assumed to be $47.00, Capital Cost is assumed to be $25, and RCAF is assumed to be 0.5. These values produce the result $714.39 using the model above. Thus, it is estimated to cost $714.39 to discover however many additional bugs would be found with whitebox fuzzing.

Chapter 5: Conclusions And Discussion

In this research, a simple blackbox and whitebox fuzzing system was built and used to test Miniz, a file compression and decompression library. Although the fuzzers built here were higher performance than ones used in similar endeavors [3], the large number of test cases generated did not trigger any bugs in Miniz's compression and decompression routines. To this end, it can be concluded that neither of the techniques implemented here are effective at finding bugs in Miniz's compression routines. CREST has been demonstrated as a good tool for several scenarios [22], but its application in the situation presented in this particular experiment was not very successful. That is, although bugs were not found within the scope of these experiments, these results pertain only to the experiments conducted here; it is entirely possible that the techniques used here would be effective at finding bugs in software other than the ones considered here.

Two factors likely contributed to the result found in these experiments. First, it seems that Miniz is an exceptionally robustly written piece of software considering its very small user base. Second, CREST was unable to reason about Miniz's compression and decompression routines very well. As was mentioned in the experiments section, CREST was able to exercise only 33 branches, a single execution path, in Miniz's decompression routine. An inspection of the decompression routine reveals that memory pointers are used extensively, something that CREST is unable to reason about. This is likely what kept CREST from producing useful input for the program.

Three factors were identified during testing that likely could have substantially increased the effectiveness of the whitebox fuzzer presented here. However, pursuing any of these three things would have been far beyond the scope of this research. First, the fuzzer could have been more closely integrated with the branch-coverage algorithms and the utilized theorem prover, Z3. This presents itself as a source of potentially more complex, but novel research topics. Second, CREST-

Z3 could have been extended to be able to reason about floating-point operations better, which would have allowed it to be used on the JPEG converter that was discarded as experimental software. Research has been completed and is available on how this can be done [28]. Third, CREST could have been modified to be able to reason about memory pointers better; poor memory reasoning is the suspected reason why CREST failed to concolically generate test cases for the second Miniz experiment. Research is also available on how this can be accomplished [29].

In addition, a simple exploratory analysis of the monetary tradeoffs associated with blackbox and whitebox fuzzing was completed. This analysis is effectively the business problem complementary to the research problem focused on in this thesis. Three models were derived for the techniques presented in chapter 3: a model for the monetary cost of blackbox fuzzing a piece of software, a model for the monetary cost of whitebox fuzzing a piece of software, and a model for the cost associated with whitebox fuzzing a piece of software after blackbox fuzzing had already been completed.

Although not highly scientific in nature, the modeling done in chapter 4 could potentially be extended and used as the framework for a higher-quality publication in the future.

APPENDIX A: SOURCE CODE

fuzzer.hpp

```
/*
 * File:   fuzzer.hpp
 * Author: toby
 *
 * Created on March 1, 2013, 6:00 PM
 */

//
//
//using namespace std;
#ifndef FUZZER_HPP
#define       FUZZER_HPP

#define DEBUG 1

#include <cstdlib>
#include <cerrno>
#include <cstring>
#include <cerrno>
#include <dirent.h>
#include <limits.h>
#include <time.h>

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <string>
#include <vector>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sstream>


using namespace std;
//using std::string;
//using std::ofstream;
//using std::strerror;
//using std::cerr;
//using std::endl;

class Fuzzer {
public:
    /*_fuzzTargetPath: command to call the target program
 _sessionRoot: a path where multiple sessions will be stored. Will contain
    input files and folders for the results of various runs
 _sessionName: the name of the folder that this fuzz session's results
    will be stored in*/
    string sut_path, session_root, session_name, input_file_folder;
    ofstream central_log_file, timing_log_file;
    bool logExitSuccess;
```

```
    bool quit_after_bug_triggered;
    char * sourceFileBinary; //the current source file gets read into this to be
manipulated--not that efficient
    int _sizeOfSourceFileBinary;
    string central_log_file_path, timing_log_file_path;

    string cur_input_file;
    int total_number_of_fuzzed_executions;
    int total_crashes;
    int initial_time; //time when the fuzzer starts

    /*Initiates a fuzz session. A fuzz session will be associated with one
     program and one log file, with associated results stored in a single
     directory.

     If the log file already exists, results will be appended to the end.*/
    Fuzzer(string sut_path, string session_name, string session_root,
            string input_file_folder, bool quit_after_bug_triggered);


    ~Fuzzer();



    vector<string> createInputFileVector();


    void fuzzInt(int sessions_per_file, int iterations_per_session);
    vector<int> getIntegerInputFileContents(string file_path);
    void fuzzFileMutate(vector<int>* session_file_contents);
    int executeSUTWithInputs(vector<int>* session_file_contents);
    int createRandomArgument(double interesting_argument_bias);

    void fuzzChar(int sessions_per_file, int iterations_per_session);
    vector<char> getCharInputFileContents(string file_path);
    void fuzzFileMutate(vector<char>* session_file_contents);
    int executeMinizWithInputs(vector<char>* session_file_contents);
    char createRandomChar(double interesting_argument_bias);



    void setLoggingSuccessfulExecutions(bool choice);


};

#endif /* FUZZER_HPP */
```

fuzzer.cpp

```
#include "fuzzer.hpp"

using namespace std;

Fuzzer::Fuzzer(string sut_path, string session_name, string session_root,
        string input_file_folder, bool quit_after_bug_triggered) {
    logExitSuccess = false;
    this->sut_path = sut_path;
    this->session_root = session_root;
    this->session_name = session_name;
    this->input_file_folder = input_file_folder;
    this->quit_after_bug_triggered = quit_after_bug_triggered;
    this->central_log_file_path = session_root + "/" + session_name + ".details";
    this->timing_log_file_path = session_root + "/" + session_name + ".timing";
    srand(time(NULL));

    total_number_of_fuzzed_executions = 0;
    total_crashes = 0;
    initial_time = time(NULL);

    /*TODO: copy input files to this directory*/
    //how to deal with naming conflicts--should be rare
    /*
    string dir = this->session_root + "/" + this->session_name;
    int rename_count = 0;
    while (mkdir(dir.c_str(), 0777) == -1) {
        cerr << "mkdir failed" << endl;
        session_name = session_name + "$";
        dir = session_root + "/" + session_name;
        rename_count++;
        if (rename_count == 100) {
            cerr << "inordinate number of renames; exiting failure" << endl;
            exit(EXIT_FAILURE);
        }
    }

    //make the directory usable
    chmod(dir.c_str(), 0777);
     */

    //TODO check if directory exists
    central_log_file.open(central_log_file_path.c_str(), ios::out | ios::app);
    timing_log_file.open(timing_log_file_path.c_str(), ios::out | ios::app);

    //write a confirmation to the log file
    //check if the central log file is open
    if (central_log_file.is_open() && timing_log_file.is_open())
        central_log_file << "%%%%%Beginning session " << session_name << endl;
    else {
        cerr << strerror(errno) << endl;
```

```
        exit(EXIT_FAILURE);
    }

}

Fuzzer::~Fuzzer() {
    delete[] sourceFileBinary;
    central_log_file.close();
}

void Fuzzer::setLoggingSuccessfulExecutions(bool choice) {
    logExitSuccess = choice;
}

void Fuzzer::fuzzInt(int sessions_per_file, int iterations_per_session) {
    //create an array of all the files in "input_file_folder"
    vector<string> inputFileVector;
    inputFileVector = createInputFileVector();

    for (int cur_file_index = 0;
            cur_file_index < inputFileVector.size(); cur_file_index++) {
        //loadInputFileToMemory(inputfiles[currentfile])
        vector<int> cur_file_contents;
        cur_file_contents = getIntegerInputFileContents(input_file_folder + "/" +
inputFileVector.at(cur_file_index));

        /*Having no arguments will cause arithmetic exceptions elsewhere in the
program.*/
        if (!(cur_file_contents.size() > 0)) {
            cerr << "The current input file contains no data. Continuing to the next one.
"
                    "Offending file: " << input_file_folder + "/" +
                    inputFileVector.at(cur_file_index) << endl;

            continue;
        }

        cur_input_file = input_file_folder + "/" + inputFileVector.at(cur_file_index);

        /*At the beginning of each session, we will always be working with
         the original input file generated by crest-z3*/
        for (int cur_session = 0; cur_session < sessions_per_file; cur_session++) {
            /*Copy the unaltered input file that is in memory to a separate
             location where it can be worked on*/
            vector<int> session_file_contents = cur_file_contents;

            for (int cur_iteration = 0; cur_iteration < iterations_per_session;
cur_iteration++) {
                //modify one token of the input file
                fuzzFileMutate(&session_file_contents);

                //test our small example program on the newly generated inputs
                executeSUTWithInputs(&session_file_contents);
            }
        }
```

```
    }
}

/**
 *
 * @return a vector containing paths to all of the files in the specified
 * input_file_folder
 */
vector<string> Fuzzer::createInputFileVector() {
    DIR* dir;
    struct dirent *ent;
    string* str;
    if ((dir = opendir(input_file_folder.c_str())) != NULL) {
        /*Add all of the file paths to the vector to return*/
        vector<string> file_paths;
        while ((ent = readdir(dir)) != NULL) {
            str = new string(ent->d_name);
            //skip hidden files and directories
            if (str->at(0) == '.' || str->at(str->length() - 1) == '~')
                continue;
            file_paths.push_back(*str);
        }
        closedir(dir);
        return file_paths;
    } else {
        cerr << "could not open directory " << input_file_folder << endl;
        exit(EXIT_FAILURE);
    }
}

vector<int> Fuzzer::getIntegerInputFileContents(string file_path) {
    ifstream input_file(file_path.c_str(), ios::in);
    vector<int> file_contents;

    //check that the file is open
    if (input_file.is_open() == false) {
        cerr << "Input file " << file_path << " failed to open" << endl;
        exit(EXIT_FAILURE);
    }

    int cur_int;
    string line;
    //make the file into a vector of ints
    while (input_file.eof() == false) {
        getline(input_file, line);
        //corner case: we read in a blank line
        if (line.size() == 0)
            continue;
        cur_int = atoi(line.c_str());
        file_contents.push_back(cur_int);
    }
    input_file.close();

    return file_contents;
}
```

```
/** Mutates the inputs that we are providing to our sample program that takes
 * a list of integers as input. This method haphazardly attempts to try the
 * following interesting inputs more often than random ones: MAX_INT, MIN_INT,
 * 0.
 *
 * @param session_file_contents The inputs to be mutated
 */
void Fuzzer::fuzzFileMutate(vector<int>* session_file_contents) {
    int argument_to_fuzz = rand() % session_file_contents->size();
    (*session_file_contents)[argument_to_fuzz] = createRandomArgument(0.01);
}

/**Returns a new integer with "interesting_argument_bias" chance of being
 an "interesting" number. An "interesting number is INT_MAX, INT_MIN, or 0.
 Otherwise, simply returns a random number.*/
int Fuzzer::createRandomArgument(double interesting_argument_bias) {
    /*verify that interesting_argument_bias is between 0 and 1*/
    if (!(interesting_argument_bias >= 0.00 && interesting_argument_bias <= 1.00)) {
        cerr << "interesting_argument_bias was not in the range" <<
                "[0.00, 1.00]" << endl;
        exit(EXIT_FAILURE);
    }

    /*Chance of intentionally choosing an "interesting" value
     RAND_MAX is the same as INT_MAX on this system*/
    int chance = RAND_MAX * interesting_argument_bias;
    //srand(time(NULL));
    if (rand() < chance) { //return an "interesting" value
        int r = rand() % 3;
        switch (r) {
            case 0:
                return INT_MAX;
            case 1:
                return INT_MIN;
            case 2:
                return 0;
        }
    } else { //return an ordinary random value
        /*rand() only returns positive ints, so we need to fix this*/
        int r = rand() % 2;
        switch (r) {
            case 0:
                return rand();
            case 1:
                return (-1) * rand();
        }
    }
}

int Fuzzer::executeSUTWithInputs(vector<int>* session_file_contents) {
    const int MAX_ARG_SIZE = 128; //each argument can be 128 characters long
    const int NUM_ARGS = 1 + session_file_contents->size();
    int elapsed_time;
    bool did_program_crash = false;
```

```
pid_t child_pid; //process ID of the child process that will be spawned
int status; //if it is an error code, we write it down

/*allocate memory for each argument to be passed to the SUT*/
char* args[NUM_ARGS];
for (int i = 0; i < NUM_ARGS; i++) {
    args[i] = (char*) malloc(sizeof (char) * MAX_ARG_SIZE);
}

/*copy the arguments from the vector to the character array*/
stringstream temp_string;
for (int cur_arg = 0; cur_arg < NUM_ARGS - 1; cur_arg++) {
    temp_string.str(string()); //clear the stream
    temp_string << (*session_file_contents)[cur_arg];
    strcpy(args[cur_arg], temp_string.str().c_str());
}

/*null terminate the argument array*/
args[NUM_ARGS - 1] = (char *) 0;

elapsed_time = time(NULL) - initial_time;
child_pid = fork();
if (child_pid == 0) //child process
{
    execv(sut_path.c_str(), args);
} else if (child_pid < 0) //failed to fork
{
    cerr << "Failed to fork" << endl;
} else //parent process
{

    do {
        //Don't block waiting and report the status of stopped children
        int w = waitpid(child_pid, &status, WUNTRACED | WCONTINUED);
        if (w == -1) {
            cerr << "waitpid returned -1" << endl;
            goto breakwaitloop;
        }
        if (WIFEXITED(status)) {
            cout << "exited, status=" << WEXITSTATUS(status) << endl;
            timing_log_file << elapsed_time << "\t" <<
                    total_crashes << "\t" <<
                    total_number_of_fuzzed_executions << endl;
        } else if (WIFSIGNALED(status)) //process was terminated by OS
        {
            did_program_crash = true;
            int crash_id = rand();
            total_crashes++;
            timing_log_file << "<crash id=" << crash_id << "> "
                    << elapsed_time << " </crash>\t" << total_crashes
                    << "\t" << total_number_of_fuzzed_executions << endl;

            /*Record the following:
                    1. The output that caused the program to crash
                    2. The wall clock time at which it crashed
```

```
                        3. The kill signal
                        4. TODO: The location of the crash dump*/
                cout << "%%%Program killed by signal " << WTERMSIG(status) << endl;

                int kill_signal = WTERMSIG(status);

                central_log_file << "<crash>" << endl;
                central_log_file << "\t<num_arguments> " << NUM_ARGS - 1 << "
</num_arguments>" << endl;
                central_log_file << "\t<sut_path> " << sut_path << "</sut_path>" << endl;
                central_log_file << "\t<time> " << elapsed_time << "</time>" << endl;
                central_log_file << "\t<kill_signal> " << kill_signal << "</kill_signal>"
<< endl;
                central_log_file << "\t<input_file_name> " << cur_input_file <<
"</input_file_name>" << endl;

                //print the arguments on separate lines for easy processing
                central_log_file << "\t<argument>" << endl;
                for (int cur_arg = 0; cur_arg < NUM_ARGS - 1; cur_arg++) {
                    central_log_file << "\t\t" << (*session_file_contents)[cur_arg] <<
endl;
                }
                central_log_file << "\t</argument>" << endl;

                central_log_file << "</crash>" << endl;

                /*crashing a program takes a relatively long time--repeated
                 executions will likely crash again and again at the same
                 spot*/
                if (quit_after_bug_triggered)
                    goto breakwaitloop;
            } else if (WIFSTOPPED(status)) {
                cout << "stopped by signal " << WSTOPSIG(status) << endl;
            } else if (WIFCONTINUED(status)) {
                cout << "continued" << endl;
            }

        } while (!WIFEXITED(status) && !WIFSIGNALED(status));

breakwaitloop:
        ;
    }

    for (int i = 0; i < NUM_ARGS; i++) {
        free(args[i]);
    }


    total_number_of_fuzzed_executions++;


    return did_program_crash;
}

void Fuzzer::fuzzChar(int sessions_per_file, int iterations_per_session) {
```

```
    //create an array of all the files in "input_file_folder"
    vector<string> inputFileVector;
    inputFileVector = createInputFileVector();

    //for each input file provided
    for (int cur_file_index = 0;
            cur_file_index < inputFileVector.size(); cur_file_index++) {
        //loadInputFileToMemory(inputfiles[currentfile])
        vector<char> cur_file_contents;
        cur_file_contents = getCharInputFileContents(input_file_folder + "/" +
inputFileVector.at(cur_file_index));

        /*Having no arguments will cause arithmetic exceptions elsewhere in the
program.*/
        if (!(cur_file_contents.size() > 0)) {
            cerr << "The current input file contains no data. Continuing to the next one.
"
                    "Offending file: " << input_file_folder + "/" +
                    inputFileVector.at(cur_file_index) << endl;

            continue;
        }

        cur_input_file = input_file_folder + "/" + inputFileVector.at(cur_file_index);

        /*At the beginning of each session, we will always be working with
         the original input file generated by crest-z3*/
        for (int cur_session = 0; cur_session < sessions_per_file; cur_session++) {
            /*Copy the unaltered input file that is in memory to a separate
             location where it can be worked on*/
            vector<char> session_file_contents = cur_file_contents;

            for (int cur_iteration = 0; cur_iteration < iterations_per_session;
cur_iteration++) {
                //modify one token of the input file
                fuzzFileMutate(&session_file_contents);

                //test our small example program on the newly generated inputs
                executeMinizWithInputs(&session_file_contents);
            }
        }
    }
}

vector<char> Fuzzer::getCharInputFileContents(string file_path) {
    ifstream input_file(file_path.c_str(), ios::in);
    vector<char> file_contents;

    //check that the file is open
    if (input_file.is_open() == false) {
        cerr << "Input file " << file_path << " failed to open" << endl;
        exit(EXIT_FAILURE);
    }

    //make the input string into
```

```
    char c;
    while (input_file.good()) {
        c = input_file.get();
        if (c != EOF)
            file_contents.push_back(c);
        else
            break;
    }

    input_file.close();

    return file_contents;
}

void Fuzzer::fuzzFileMutate(vector<char>* session_file_contents) {
    int argument_to_fuzz = rand() % session_file_contents->size();
    (*session_file_contents)[argument_to_fuzz] = createRandomChar(0.3);
}

char Fuzzer::createRandomChar(double interesting_argument_bias) {
    /*verify that interesting_argument_bias is between 0 and 1*/
    if (!(interesting_argument_bias >= 0.00 && interesting_argument_bias <= 1.00)) {
        cerr << "interesting_argument_bias was not in the range" <<
                "[0.00, 1.00]" << endl;
        exit(EXIT_FAILURE);
    }

    /*Chance of intentionally choosing an "interesting" value
     RAND_MAX is the same as INT_MAX on this system*/
    int chance = RAND_MAX * interesting_argument_bias;
    //srand(time(NULL));
    if (rand() < chance) { //return an "interesting" value
        int r = rand() % 3;
        switch (r) {
            case 0:
                return CHAR_MAX;
            case 1:
                return CHAR_MIN;
            case 2:
                return 0;
        }
    } else { //return an ordinary random value
        /*rand() only returns positive ints, so we need to fix this*/
        int r = rand() % 2;
        switch (r) {
            case 0:
                return (char) rand();
            case 1:
                return (-1) * ((char) rand());
        }
    }
}

/*We will always be providing miniz with just one argument: the string
 to compress*/
```

```
int Fuzzer::executeMinizWithInputs(vector<char>* session_file_contents) {
    const int NUM_ARGS = 2;
    int elapsed_time;
    bool did_program_crash = false;
    pid_t child_pid; //process ID of the child process that will be spawned
    int status; //if it is an error code, we write it down

    /*allocate memory for each argument to be passed to the SUT*/
    char* args[NUM_ARGS];
    args[0] = (char*) malloc(sizeof (char) * (session_file_contents->size() + 1));

    /*copy the arguments from the vector to the character array*/
    stringstream temp_string;
    temp_string.str(string()); //clear the stream
    for (int i = 0; i < session_file_contents->size(); i++)
        temp_string << (*session_file_contents)[i];
    memcpy(args[0], temp_string.str().c_str(), temp_string.str().size());

    /*null terminate the argument array*/
    args[NUM_ARGS - 1] = (char *) 0;

    //execute the program
    elapsed_time = time(NULL) - initial_time;
    child_pid = fork();
    if (child_pid == 0) //child process
    {
        execv(sut_path.c_str(), args);
    } else if (child_pid < 0) //failed to fork
    {
        cerr << "Failed to fork" << endl;
    } else //parent process
    {

        do {
            //Don't block waiting and report the status of stopped children
            int w = waitpid(child_pid, &status, WUNTRACED | WCONTINUED);
            if (w == -1) {
                cerr << "waitpid returned -1" << endl;
                goto breakwaitloop;
            }
            if (WIFEXITED(status)) {
                cout << "exited, status=" << WEXITSTATUS(status) << endl;
                timing_log_file << elapsed_time << "\t" <<
                        total_crashes << "\t" <<
                        total_number_of_fuzzed_executions << endl;
            } else if (WIFSIGNALED(status)) //process was terminated by OS
            {
                did_program_crash = true;
                int crash_id = rand();
                total_crashes++;
                timing_log_file << "<crash id=" << crash_id << "> "
                        << elapsed_time << " </crash>\t" << total_crashes
                        << "\t" << total_number_of_fuzzed_executions << endl;

                /*Record the following:
```

```
                        1. The output that caused the program to crash
                        2. The wall clock time at which it crashed
                        3. The kill signal
                        4. TODO: The location of the crash dump*/
                cout << "%%%Program killed by signal " << WTERMSIG(status) << endl;

                int kill_signal = WTERMSIG(status);

                central_log_file << "<crash>" << endl;
                central_log_file << "\t<num_arguments> " << NUM_ARGS - 1 << "
</num_arguments>" << endl;
                central_log_file << "\t<sut_path> " << sut_path << "</sut_path>" << endl;
                central_log_file << "\t<time> " << elapsed_time << "</time>" << endl;
                central_log_file << "\t<kill_signal> " << kill_signal << "</kill_signal>"
<< endl;
                central_log_file << "\t<input_file_name> " << cur_input_file <<
"</input_file_name>" << endl;

                //print the arguments on separate lines for easy processing
                central_log_file << "\t<argument> ";
                for(int i = 0; i < session_file_contents->size(); i++)
                        central_log_file << (*session_file_contents)[i];
                central_log_file << endl;
                central_log_file << " </argument>" << endl;

                central_log_file << "</crash>" << endl;

                /*crashing a program takes a relatively long time--repeated
                 executions will likely crash again and again at the same
                 spot*/
                if (quit_after_bug_triggered)
                    goto breakwaitloop;
            } else if (WIFSTOPPED(status)) {
                cout << "stopped by signal " << WSTOPSIG(status) << endl;
            } else if (WIFCONTINUED(status)) {
                cout << "continued" << endl;
            }

        } while (!WIFEXITED(status) && !WIFSIGNALED(status));

breakwaitloop:
        ;
    }

    for (int i = 0; i < NUM_ARGS; i++) {
        free(args[i]);
    }

    total_number_of_fuzzed_executions++;

    return did_program_crash;
}
```

WORKS CITED

[1] J. Burnim, "Heuristics for Scalable Dynamic Test Generation," in *Automated Software Engineering*, 2008.

[2] B. Arkin, "Adobe Reader and Acrobat Security Initiative," Adobe Systems Incorporated, 20 May 2009. [Online]. Available: http://blogs.adobe.com/asset/2009/05/adobe_reader_and_acrobat_secur.html. [Accessed 5 November 2012].

[3] J. Neystadt, "Automated Penetration Testing with White-Box Fuzzing," Microsoft, February 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc162782.aspx#Fuzzing_topic9. [Accessed 27 March 2013].

[4] N. Tillmann and P. de Halleux, "Pex and Moles Public Slides," 2008. [Online]. Available: http://research.microsoft.com/en-us/projects/pex/. [Accessed 27 March 2013].

[5] M. Aslani, N. Chung, J. Doherty, N. Stockman and W. Quach, "Comparison of Blackbox and Whitebox Fuzzers in Finding Software Bugs," 2008.

[6] B. P. Miller, L. Fredriksen and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM,* vol. 33, no. 12, pp. 32-44, 1990.

[7] 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, 2000.

[8] P. Wibbeler, "Fuzzing Reader -- Lessons Learned," Adobe Systems Incorporated, 1 December 2009. [Online]. Available: http://blogs.adobe.com/asset/2009/12/fuzzing_reader_-_lessons_learned.html. [Accessed 16 September 2012].

[9] S. Jana and V. Shmatikov, "Abusing File Processing in Malware Detectors for Fun and Profit," in *2012 IEEE Symposium on Security and Privacy*, San Francisco, 2012.

[10] "Month of Browser Bugs," July 2006. [Online]. Available: http://browserfun.blogspot.com/.

[11] Microsoft, "SDL Process: Verification," Microsoft, [Online]. Available: http://www.microsoft.com/security/sdl/discover/verification.aspx. [Accessed 5 November 2012].

[12] G. Holmes, "Cisco Security and the Layered Defense Approach," Cisco Systems, 29 October 2012. [Online]. Available: http://blogs.cisco.com/security/cisco-security-and-the-layered-defense-approach/. [Accessed 5 November 2012].

[13] P. Godefroid, M. Y. Levin and D. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the 2nd international workshop on Random testin*, Atlanta, 2008.

[14] "Peach Fuzzing Platform," [Online]. Available: http://peachfuzzer.com. [Accessed 8 November 2012].

[15] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European software engineering conference*, New York, 2005.

[16] J. King, "Symbolic execution and program testing," *Communications of the ACM,* vol. 19, no. 7, pp. 385-394, 1976.

[17] P. e. al., "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008.

[18] Microsoft, "Z3," 15 February 2013. [Online]. Available: http://z3.codeplex.com/. [Accessed 27 March 2013].

[19] P. Godefroid, "From Blackbox Fuzzing to Whitebox Fuzzing towards Verification," July 2010. [Online]. Available: research.microsoft.com. [Accessed 1 November 2012].

[20] "How Samba Was Written," [Online]. Available: http://www.samba.org/ftp/tridge/misc/french_cafe.txt.

[21] J. Cabello, H. Yin, Z. Liang and D. Song, "Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, New York, 2007.

[22] L. Zhiqiang, X. Jiang, D. Xu and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Lin, Zhiqiang, et al. Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, 2008.

[23] P. Boonstoppel, C. Cadar and D. Engler, "RWset: Attacking path explosion in constraint-based test generation," in *Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, 2008.

[24] B. Elkarablieh, P. Godefroid and M. Levin, "Precise pointer reasoning for dynamic test generation," in *Proceedings of the eighteenth international symposium on software testing and analysis*, Chicago, 2009.

[25] V. Ganesh, T. Leek and M. Rinard, "Taint-based directed whitebox fuzzing," in *IEEE 31st International Conference on Software Engineering*, Vancouver, 2009.

[26] "crest: automatic test generation tool for C," [Online]. Available: https://code.google.com/p/crest/.

[27] B. Miller, L. Fredriksen and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM,* vol. 33, no. 12, 1990.

[28] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *Proceedings of the 19th international symposium on Software testing and analysis*, Trento, 2010.

[29] B. Elkarablieh, P. Godefroid and M. Levin, "Precise pointer reasoning for dynamic test generation," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, Chicago, 2009.

[30] P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Chicago, 2005.