The Pennsylvania State University
The Graduate School

STATIC INSTRUMENTATION FOR PERFORMANT BINARY FUZZING

A Thesis in
Computer Science and Engineering
by
Eric Pauley

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2020

The thesis of Eric Pauley was reviewed and approved* by the following:

Patrick D. McDaniel
William L. Weiss Professor of Information and Communications Technology
Thesis Advisor

Danfeng Zhang
Professor of Computer Science and Engineering

Chitaranjan Das
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

Abstract

Rapid advancements in fuzz testing have achieved the ability to quickly and comprehensively find security-critical faults in software systems. Yet, the most advanced of these techniques rely on access to application source code, which is often unavailable in practice. In this paper, we explore techniques to replicate the depth and efficiency of source-code available fuzzers via static binary instrumentation. Developing such instrumentation is difficult because compilation is a lossy process, and much of the source-level semantics leveraged by these techniques are not available in binaries. We recover much of this information via heuristic control flow reconstruction, a shadow stack for function identification, and a novel technique for instrumenting comparison instructions. We evaluate ReFuzz on the LAVA-M dataset, achieving the same effectiveness as a best-in-class source-available fuzzer with a 3.4× execution time overhead (lower than existing dynamic fuzzing approaches). In this way, we show that techniques for binary fuzzing may approach the functional ability of source-available fuzzing.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE1255832 and the National Science Foundation Grant No. CNS-1564105. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Chapter 1
Introduction

Fuzz testing (fuzzing), an automated technique that generates interesting[1] program inputs from known good ones, is a powerful way to discover corner cases and bugs in programs that are often missed by manually-written test cases. By tracking inputs that achieve high code coverage, and employing heuristics to generate new inputs, fuzzers efficiently explore program states. Recent advances in fuzzing have led to more complete coverage of program states [2], as well as more effective techniques for generating new inputs [2–6]. Many popular security-critical projects now use fuzz testing to prevent vulnerabilities from affecting end users [7], and software CVE discoveries can often be attributed to crashes found by fuzzers [3].

Fuzzers often leverage source code access to generate efficient fuzzing instrumentation and rapidly discover bugs [2,3,8]. For instance, Chen et al. recently introduced Angora [2], which uses source-level typing and control flow information to generate efficient fuzzing instrumentation. In many cases, the assumption of access to application source code makes sense. For example, a software developer who uses fuzzing to identify and fix bugs in their own software would naturally have access to source code. However, in other scenarios this assumption is far less certain. There are broadly two scenarios in which fuzzing might target executables without source code: (1) adversarial exploit generation against third-party software, and (2) software assurance on legacy systems or proprietary software for which no source is available.

When source code is not available, many of the most powerful fuzzing techniques are no longer available. The primary reasons for this are twofold:

1. Fuzzers work by inserting instrumentation into programs. Modern compilers provide

[1] What consititutes a unique or "interesting" input has not been precisely defined [1], and in some cases is a hyperparameter of the fuzzer. In most cases, fuzzers consider an input interesting if its control flow is unique.

an intermediate representation, which allows instrumentation to be inserted while providing control flow and typing information. This feature makes instrumentation with source code relatively simple. Compilation is a lossy process, and the lack of control flow information in binary code makes static binary instrumentation more difficult. Binary fuzzers [5,6,9–12] to date have all relied on dynamic instrumentation frameworks such as Pin [13] or QEMU [14], which incur performance penalties.

2. Fuzzers employ heuristics to discover and evaluate new interesting inputs. The most successful heuristics to date employ control-flow and typing information from the application source code to both discover and evaluate inputs. Extant binary instrumentation does not sufficiently recover information lost during compilation, so techniques that use that information cannot be applied.

In this work, we present new techniques for binary fuzzing, using source-available fuzzers on binaries without source code, allowing extant fuzzing techniques to efficiently track and discover interesting program inputs and bugs. Our tool, ReFuzz, instruments binaries for fuzzing using techniques previously limited to source-available fuzzers. ReFuzz builds on recent published work in binary instrumentation [15], and implements additional approaches relevant to fuzzing instrumentation. ReFuzz-instrumented binaries can be fuzzed by existing tools with minimal modification.

Our work in static binary fuzzing faces three central challenges: (1) static control flow instrumentation with full coverage, (2) tracking function call context without making assumptions about memory layout, and (3) instrumenting comparisons with equivalent accuracy to source-level approaches. In addressing these challenges, ReFuzz instruments binaries with source-level accuracy while reducing performance overhead over previous approaches.

We evaluate ReFuzz against extant source- and binary-level fuzzers. On a collection of manually-inserted bugs in C programs, ReFuzz finds more bugs than any other fuzzer tested, including a bug that is compiler-dependent and so is missed by fuzzers that instrument source code. On the LAVA-M corpus [16], ReFuzz found substantially the same bugs as Angora's dynamic taint tracking mode without access to source code. On LAVA-M, ReFuzz incurred a 3.4× overhead compared to Angora, which is lower than other binary fuzzing approaches using dynamic instrumentation. ReFuzz's approximate 5× binary size overhead is also an acceptable trade-off for the ability to fuzz without source code. In some cases, ReFuzz's binary instrumentation finds more bugs on average than Angora's published results regardless of mode used. Our evaluation demonstrates

that ReFuzz achieves comparable bug-finding performance to modern source-level fuzzers without access to source code.

In summary, we make the following contributions:

1. We develop approaches for statically instrumenting binaries for fuzzing with source-level accuracy.

2. We demonstrate techniques for minimizing performance overhead of binary fuzzing instrumentation.

3. We evaluate ReFuzz on manually- and artificially-generated bugs in programs, matching the bugs found by a state-of-the-art source-available fuzzer while minimizing performance overhead.

ReFuzz allows the most advanced fuzzing techniques available to be used on binaries. This capability has broad implications for software assurance: the community is no longer limited to auditing the security of open source software alone. Researchers and software testers can use binary fuzzers to audit proprietary and legacy software without requiring source code from developers. Further, binary fuzzing has adversarial implications, as withholding source code no longer protects programs from exploitation by fuzzers. This challenges the notion of security through obscurity that often motivates closed-source software development practices.

Chapter 2
Background

Fuzz testing is a specialized approach to randomized test-case generation. Early works in randomized test case generation built on the assumption that program inputs often follow a format, defined by a context-free grammar (CFG) [17]. Given a specification of a given program's input CFG, test cases can be generated by following CFG rules until only terminals (string literals) exist. Randomly generated strings that satisfy a given program's CFG are semi-valid, and exercise portions of a program past the initial parsing stage.

Test-case generation based on CFGs faces two major limitations: (1) specifying the CFG requires manual analysis of the program under test, and (2) inputs that satisfy a CFG for a program may still be trivially invalid, as the behavior of meaningful programs is generally not context-free.

Fuzz testing takes a different approach to randomized test-case generation that does not suffer from the above limitations. Fuzzers employ a *fuzzing loop*, which repeatedly selects a valid program input (provided by the user) to randomly mutate (*fuzz*) into a new input [18]. Intuitively, minor modifications to valid program inputs should generate inputs that are also valid, while potentially causing the program to behave differently. While fuzzing can be enhanced by improving how inputs are mutated, random mutation of valid inputs is fundamental to the technique of fuzzing [19].

Black-box fuzzers implement the most basic approach to fuzzing, mutating inputs without knowledge of the program under test [20]. Recent fuzzing approaches leverage increased access to the program under test to effectively mutate program inputs. These *grey-box fuzzers* instrument the program to monitor execution and determine which input mutations invoke new behavior. We broadly classify grey-box fuzzers into source-available fuzzers (i.e., requiring source code access) and binary fuzzers (i.e., using binary-level instrumentation).

2.1 Source-available fuzzing

When source code is available, a grey-box fuzzer can take advantage of rich information in the source to instrument the program under test for fuzzing. This instrumentation might measure code coverage, track information flow, and record values of variables for use by the fuzzer. We primarily explore source-available techniques as implemented by Angora [2]. While contemporary fuzzers such as RedQueen [6] achieve competitive performance, Angora's flexible approach makes it emblematic of state-of-the-art fuzzing techniques.

Source-available fuzzers analyze and instrument programs at compile-time, relying on high-level information from source code. This allows them to extract useful information about the program's behavior that is not readily apparent in compiled binaries. Prominent source techniques include:

1. *Code coverage instrumentation.* The control flow coverage of each program execution is recorded. If two inputs cause similar code coverage, only one is kept. This effectively curates a minimal set of inputs that trigger all discovered program behavior.

   Code coverage is difficult to quantify as complete coverage by one metric might not exercise all possible program behavior. Coverage instrumentation for fuzzing must approximate program behavior with minimal overhead. To achieve this, fuzzers such as AFL [3], Driller [4], and Angora [2] measure control flow edge coverage under each input, storing approximate coverage in a fixed-size bitmap for performance reasons. Angora extends this metric by additionally tracking the context (i.e., the function call stack) of these edges.

2. *Taint tracking.* The information flow of each input byte is tracked.This allows the fuzzer to only mutate input bytes that directly affect other instrumentation. This instrumentation can be performed at compile-time (e.g., using LLVM's [21] DataFlowSanitizer) or using a dynamic binary instrumentation tool such as Intel's Pin [13].

3. *Comparison instrumentation.* A program's control flow is the result of its individual control flow instructions. As a consequence, fuzzers can readily discover new program inputs by mutating to affect any given conditional instruction. LAF-Intel [8] and Angora [2] record the inputs and results of each comparison, though they employ varying methods to mutate inputs based on these values.

```
1 int main(int argc, char** argv) {
2     long int val = 0;
3     fread(&val, 2, 1, stdin);
4     if ((val - 1234) * (val - 1230) < 0)
5         val = *(volatile int*)NULL;
6     return val;
7 }
```

(a) A C program that crashes when `val` $\in [1231, 1233]$.

Input	`val`	f	Description
`0 0`	0	1517821	❶ Initial Input
`1 0`	1	1515358	❷a Compute Gradient at (`0 0`)
`255 0`	255	954526	
`0 1`	256	952573	
`0 255`	65280	4102146301	
`0 1`	256	952573	❸a Descend Gradient
`0 2`	512	518397	
`255 4`	1279	2206	
`252 8`	2300	1140621	
`254 4`	1278	2113	
`252 4`	1276	1933	
`248 4`	1272	1597	
`240 4`	1264	1021	
`224 4`	1248	253	
`192 4`	1216	253	
`224 5`	1504	73981	
`225 4`	1249	286	❷b Compute Gradient at (`224 4`)
`223 4`	1247	222	
`224 5`	1504	73981	
`224 3`	992	57597	
`216 4`	1240	61	❸b Descend Gradient
`215 4`	1239	46	
`213 5`	1237	22	
`209 3`	1233	-3	
`209 3`	1233	-3	❹ Input Solves Conditional

(b) Inputs passed to solve the conditional. ($f = (\texttt{val} - 1234) \times (\texttt{val} - 1230)$)

Figure 2.1: Inputs passed to explore a conditional. The program is instrumented by ReFuzz and fuzzed by Angora.

Based on the above instrumentation, several source-level fuzzers differ primarily in how they craft new inputs from the collected information. LAF-Intel [8] links comparison instrumentation and code coverage, artificially inflating the coverage of an input when it partially solves a conditional. This is done by splitting each comparison into multiple nested comparisons, each of which is more likely to be solved by random mutation. Angora employs gradient descent, an optimization technique, to solve conditionals directly.

Gradient descent models each comparison as a constraint on some abstract function of the input. The goal, then, is to modify the input such that the truthiness value of this constraint changes. Based on an initial concrete program input, each relevant byte of the input is mutated independently. This allows the fuzzer to compute a gradient of the function with respect to each byte of the input. Based on the constraint, the fuzzer follows this computed gradient to minimize the concrete value of the function. Figure 2.1 shows an example application of gradient descent to discover new program behavior. At ❶, taint tracking is used to determine which input bytes affect the conditional. Step ❷a numerically computes the gradient of the function f with respect to these input bytes by running the program with each byte incremented or decremented. Step ❸a follows this gradient to minimize f. This process continues until either the conditional has been successfully solved (❹), or following the gradient no longer reduces f. In this case, the gradient is recomputed with new inputs and followed (❷b and ❸b) until the conditional is solved. Through repeated application of this procedure, the fuzzer solves conditionals concretely, discovering new program behavior and potential bugs.

2.2 Binary Fuzzing Instrumentation

When only binary code is available, instrumenting for grey-box fuzzing becomes more difficult because source-level information is removed by the compilation process. This is further complicated when binaries are obfuscated or debug symbols are removed. Binary fuzzers seek to approximate the techniques of source-available fuzzers without source code. In general, one can perform either dynamic or static instrumentation to support binary-level fuzzing:

1. *Instrument dynamically.* Extant binary fuzzers [5,6,9–12] employ this method, using a binary instrumentation framework such as Pin [13], QEMU [14], or DynInst [11]. Dynamic instrumentation tools have inherent performance overhead, as they must interpret the executable and determine instrumentation points at runtime. While this overhead can be reduced using just-in-time compilation techniques, it cannot be eliminated entirely.

2. *Add instrumentation statically.* A program can be rewritten to include fuzzing instrumentation within the executable code. This requires performing static analysis on the executable without runtime information, extracting instruction and control-flow information, and reassembling a new program. While static instrumentation

avoids the runtime overhead of dynamic instrumentation, it is difficult because control flow and typing information is lost during compilation, and cannot be inferred in general without running the program. ReFuzz demonstrates techniques for static binary fuzzing instrumentation.

In a near-simultaneous work, Dinesh et al. [22] explore applications of static instrumentation for fuzzing. Their work focuses primarily on ensuring soundness during the rewriting process, rather than matching the performance of source-available fuzzers. In contrast, our work explores challenges relating specifically to applying state-of-the-art fuzzing techniques to binaries.

Some binary fuzzers focus on reproducing source-level techniques using approximation. AFL-DynInst [11] instruments dynamically for fuzzing by AFL [3], and Steelix [12] implements comparable techniques to LAF-Intel [8] without compile-time instrumentation. Such works generally aim to find the same crashes per execution as their source-available counterparts, while minimizing execution time overhead.

For techniques that do not benefit from source instrumentation, fuzzers first demonstrate techniques on binaries. RedQueen [6] and VUzzer [5] are directly implemented using dynamic instrumentation.

2.3 Static Binary Instrumentation

Our work in binary fuzzing builds directly on recent developments in the space of static binary instrumentation, which incurs several key challenges.

Application binaries contain executable code and data. One of the greatest challenges to static binary instrumentation is determining the meaning of these bytes without actually running the program, especially when binaries are obfuscated and debug info is removed. A rewriter must obtain a correct disassembly of the executable and modify the instructions without breaking original functionality. Different rewriting frameworks approach these problems with varying success.

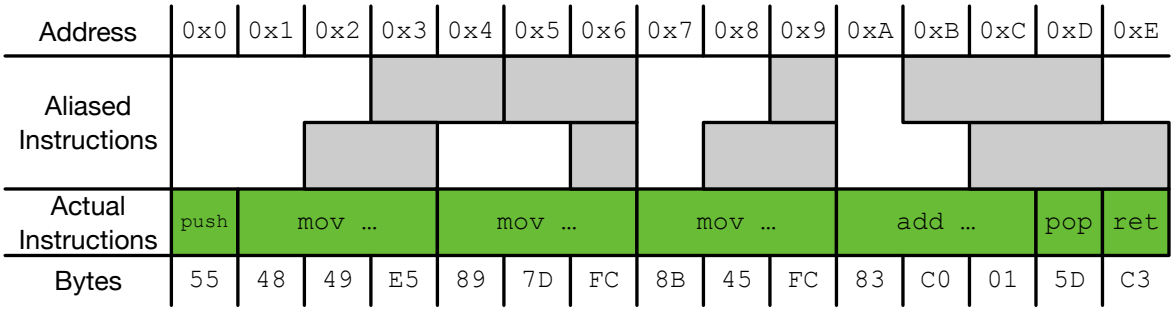
Determining valid executable offsets in a program is essential to extracting actual executed code. While this is straightforward for a CPU architecture that uses word-aligned instructions, it is more difficult for a variable-width instruction set such as x86 [24]. Figure 2.2 shows an example function and its resulting binary, along with instructions that exist in the resulting binary but not in the original program.

Most compilers align valid instructions based on function boundaries, and Uroboros uses this analysis to extract valid instructions [25, 26]. The authors of Uroboros note

```
1 int foo(int a)
2     return a + 1;
```

```
0 55          push rbp
1 48 89 e5    mov  rbp, rsp
4 89 7d fc    mov  [rbp-0x4], edi
7 8b 45 fc    mov  eax, [rbp-0x4]
A 83 c0 01    add  eax, 0x1
D 5d          pop  rbp
E c3          ret
```

(a) A simple C function `foo` compiled with GCC (`-O0`)



(b) Assembled binary. Actual instructions map to source assembly. Aliased instructions are valid instructions at other offsets.

Figure 2.2: x86 machine code is unaligned; original instructions are aliased by additional valid instructions at most byte offsets. Determining used instructions statically is undecidable [23].

that their tool can become more effective as function identification improves. However, even the most advanced function boundary extraction methods often fail to identify boundaries precisely under adversarial conditions such as obfuscation [27, 28]. Binary fuzzers are an adversarial tool, and as such must be resistant to these obfuscation techniques. Using a rewriting tool that depends on function boundaries is not viable.

In recent years binary analysis tooling has placed more emphasis on determining valid instruction offsets without relying on function boundary identification [29]. Although, as Wartell et al. note [23], sound static disassembly is undecidable in general, these techniques can correctly disassemble most programs, and as a result rewriters such as RAMBLR [30] have been developed that rely on these new instruction identification techniques. RAMBLR makes fewer assumptions about instruction offsets, and so works correctly on more programs, though it is not entirely resistant to obfuscation and still relies on heuristics to determine instruction locations.

An alternate approach has been proposed by Bauman et al. [15]. They argue that, instead of relying on heuristics to determine what instructions are disassembled, a rewriter can simply disassemble all valid instructions in the binary. This constitutes a superset of the actual useful code in the executable, motivating the technique's name of *Superset Disassembly*. Bauman et al. implement this concept in MULTIVERSE, which rewrites binaries without using heuristics. Though binaries rewritten in this fashion have relatively high size overhead, they introduce low execution time overhead while ensuring that all possible execution paths are instrumented. REFUZZ builds on MULTIVERSE, analyzing each instruction individually and inserting fuzzing instrumentation.

Chapter 3
Binary Rewriting for Fuzzing

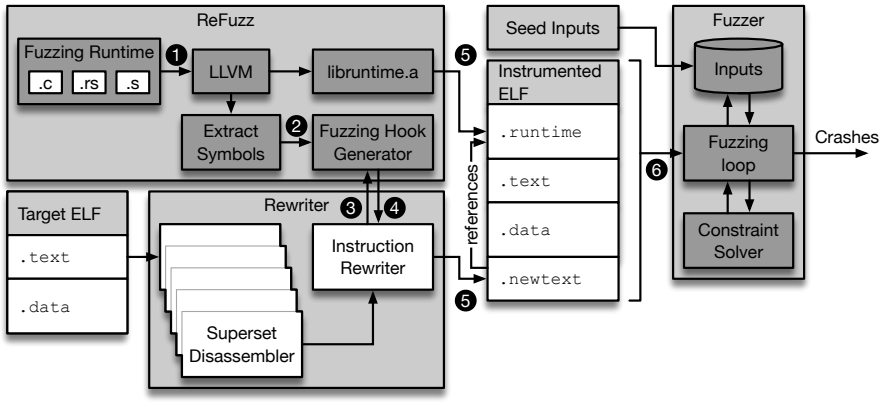


Figure 3.1: System for fuzz testing using ReFuzz. ReFuzz generates fuzzing instrumentation within Multiverse.

We investigate static binary instrumentation for fuzzing using binary rewriting. While static instrumentation is more difficult to achieve soundly than dynamic instrumentation, we show that static instrumentation can be used to achieve fuzzing techniques previously only available with source code.

Our tool, ReFuzz, integrates with an existing binary rewriter (Multiverse [15][1]) to instrument stripped binaries for fuzzing, inserting binary approximations of compile-time instrumentation. The instrumented program contains the original binary with inline instrumentation, and an instrumentation runtime. Figure 3.1 presents this integration, which consists of several steps:

❶ The *fuzzing runtime* (code provided by the fuzzer) is compiled using LLVM [21]. This runtime is normally written in a high-level language such as C or Rust, and

[1] Multiverse rewrites instructions at all byte offsets, ensuring that the entire program is instrumented without relying on heuristics.

linked with the target program at compile-time. In contrast, ReFuzz produces an independent statically-linked binary. This step allows the runtimes of existing fuzzers to be used with minimal modification.

❷ The symbols exported by the fuzzing runtime are extracted. These symbols will then be available to inline instrumentation.

❸ Each possible instruction in the target ELF file is processed by ReFuzz as a candidate for instrumentation. ReFuzz generates instrumentation assembly for control flow instructions and comparisons, and inserts calls to functions in the fuzzing runtime to record information about these instructions.

❹ The rewriter reassembles a new text segment containing original instructions and inline fuzzing instrumentation.

❺ The fuzzing runtime and rewritten text segments are inserted into a new ELF file, along with the original sections from the target binary. The rewritten text segment contains valid static references to the instrumentation runtime, allowing inline instrumentation in `.newtext` to access functionality written in a high-level language.

❻ The final instrumented ELF is passed to the fuzzer, which fuzzes the program as if it had been instrumented at compile-time.

Binary rewriting allows programs to be instrumented for fuzzing while maintaining performance and compatibility with existing fuzzers. Because the resulting program requires minimal modification to the fuzzer itself, future improvements to the fuzzer itself can apply to ReFuzz-instrumented programs automatically. Additionally, ReFuzz's ability to instrument binaries statically is a key strength over existing binary fuzzing techniques.

3.1 Challenges to Static Instrumentation

Compilation creates compact programs that run efficiently on processors. However, it greatly complicates analysis and instrumentation, as information about program behavior (i.e., data and control flow) is irreversibly lost. To statically instrument programs for fuzzing, ReFuzz must reverse-engineer, infer, or approximate some source-level information. Several challenges in this space are unique to fuzzing:

1. *Coverage Measurement.* Source-available fuzzers instrument control flow edges during compilation. This allows them to easily measure code coverage at runtime. Likewise, extant binary fuzzers record this information dynamically. In contrast, when statically rewriting binaries for fuzzing, inserting this information is not straightforward. The locations of basic blocks in the program cannot be soundly inferred. ReFuzz implements coverage instrumentation that maintains precision with minimal runtime overhead.

2. *Context Sensitivity.* Recent work [2] has shown that tracking coverage using control-flow edges alone is not sufficient to fully exercise a program. In a given function, for instance, the same control flow may be interesting in one context but not in another. Recent source-level fuzzers additionally track the *context* of control flow edges based on the current function call stack. The compiler exposes function call information to instrumentation, and so adding this is straightforward with source code. When rewriting binaries, however, information on the stack cannot be easily interpreted by instrumentation. ReFuzz uses a shadow stack to track this context.

3. *Inferring Comparisons.* As noted in Section 2.1, source-level fuzzers record the inputs and results of comparisons. This information is used to guide mutation strategies, quickly discovering inputs that trigger new behavior. Information on comparisons is also made available to compiler instrumentation. At the binary level, these comparisons cannot, in general, be inferred statically, as they can span multiple instructions and be separated by arbitrary intermediate control flow. Dynamic approaches have approximated comparisons. ReFuzz tracks comparisons more effectively than extant dynamic solutions by mirroring the processor state, with accuracy comparable to source-level instrumentation.

3.2 Approach

3.2.1 Coverage Measurement

Fuzzers work by finding new and interesting program behavior. Since new program behavior is generally caused by different control flow, measurement of control flow coverage is essential to modern fuzzers [2, 3, 6]. For each execution of a program, the set of control flow edges reached is recorded, and the fuzzer keeps inputs that maximize the set of control flow edges reached. When instrumenting for this at the source-code level,

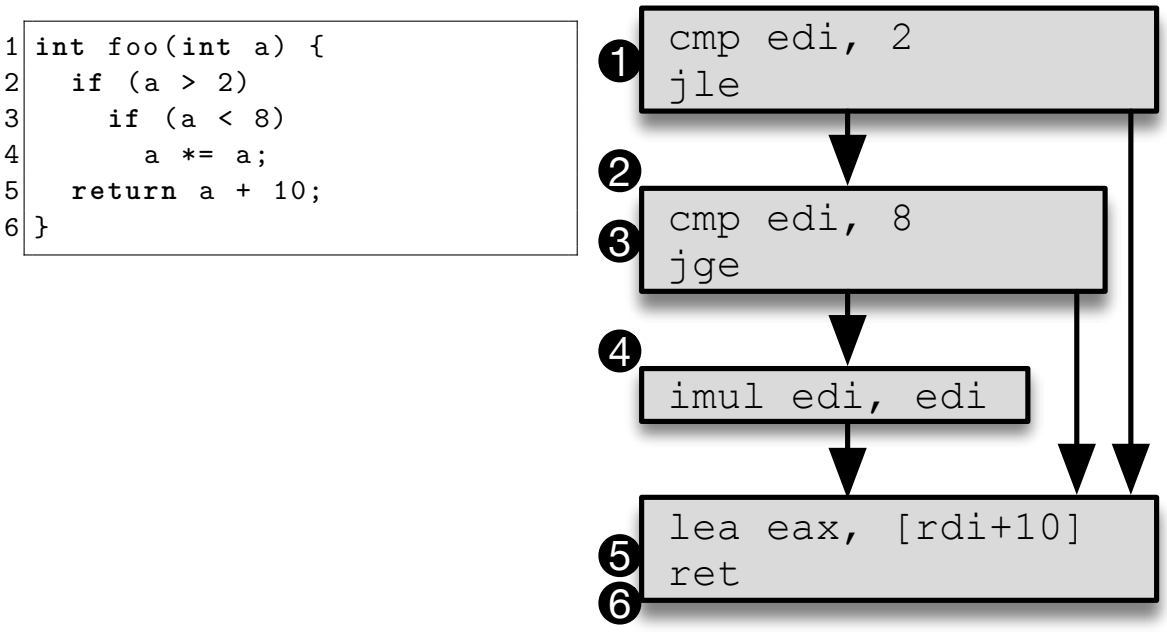


Figure 3.2: Disassembly and control flow of a simple function. Basic blocks representing lines 4 and 5 in the C source are not separated by any control flow instructions. Circled numbers are points where ReFuzz inserts coverage instrumentation.

direct control flow edges are readily apparent during compilation.

When instrumenting binaries statically for control flow, this method is not available because a program's basic blocks or control flow targets cannot be soundly inferred. ReFuzz aims to ensure that all control flow edges are instrumented despite this limitation. The key insight is that, while failing to instrument a given control flow edge could prevent interesting inputs from being considered, inserting too much instrumentation only incurs an execution time overhead. Consider the set of control flow edges tracked by the fuzzer. If a control flow edge is duplicated in this set (i.e., counted as two different edges), this will not have affect which inputs are kept.

As a result of this insight, we strategically over-instrument all possible control flow instructions. ReFuzz inserts coverage instrumentation before and after each control flow instruction without determining basic block boundaries. In Figure 3.2, instrumentation is inserted at points ❶-❻ around each control flow instruction. Control flow edges are recorded as pairs of these points (e.g., (❶, ❺) when $a = 0$).

By over-instrumenting, ReFuzz ensures that all control flow edges are instrumented. This does, however, incur an execution time overhead. These duplicate edges must be recorded and processed by the fuzzer, causing each execution to take longer. For instance,

in Figure 3.2, the edge (❷, ❸) is a duplicate control flow edge because it is contained within a single basic block. Because control flow edges are stored in a limited-size bitmap, duplicate edges might occupy a slot used by some other edge, preventing the fuzzer from tracking other true edges.This is an acceptable trade-off for the ability to fuzz test binaries, and techniques for mitigating overhead of coverage tracking has been considered by prior works [3, 8].

3.2.1.0.1 Indirect Jumps In some cases, control flow is accomplished via indirect jumps, which cannot be inferred statically. For example, a C `switch` statement is often compiled into a table that is used by an indirect jump. Other indirection (e.g., function pointers) may not be soundly considered even with source code available. To account for these cases, ReFuzz determines indirect jump destinations at runtime and records appropriate control flow edges. This ensures that all control flow is tracked, but may fill data structures if indirect jumps are heavily used.

3.2.2 Context Sensitivity

Control flow edge coverage alone does not fully describe program behavior. For instance, for an example in a later section (Figure 5.1c) a fuzzer might have full control flow edge coverage, but still not discover the bug. To mitigate this issue, fuzzers have augmented control flow edge coverage to be context-sensitive, recording the call context of each control flow edge. When instrumenting source code, this call context is tracked alongside other local information on the stack [2]. While adding new information to the program stack during compilation is possible, this cannot be done on a binary without modifying the stack layout and affecting the original program.

Unlike existing context-sensitive fuzzers, ReFuzz uses a shadow stack to track function call context. Each call and return is instrumented to modify this shadow stack, and current context is reported to the fuzzer along with control flow edges. This successfully tracks call context for programs that use x86-64 `call` and `ret` instructions, but may cause inaccuracy when these instructions are not used for making calls and returns, or are used for purposes other than calls and returns. For instance, a program might repeatedly use call instructions as direct jumps to confuse context tracking. Approaches to anti-fuzzing obfuscation are discussed in Section 7.1.

```
1 int min(int a, int b) {
2   if (a > b)
3       return b;
4   return a;
5 }
```

```
0 39 f7        cmp    edi,esi # Compare
2 89 f0        mov    eax,esi
4 0f 4e c7     cmovle eax,edi # Conditional
7 c3           ret
```

Figure 3.3: Disassembly of a `min` function. Control flow is split into two instructions, separated by an unrelated instruction.

Table 3.1: Comparisons and constraints in x86 assembly. Each x86 conditional instruction maps to a constraint that can be solved using gradient descent. (Derived from [2])

Comparison	f	Constraint	Instruction
$a < b$	$f = a - b$	$f < 0$	`JAE`, `JBE`, `JGE`, `JLE`
$a > b$	$f = b - a$	$f < 0$	
$a <= b$	$f = a - b$	$f <= 0$	`JA`, `JB`, `JC`, `JG`, `JL`
$a >= b$	$f = b - a$	$f <= 0$	
$a! = b$	$f = -abs(a - b)$	$f < 0$	`JNE`
$a == b$	$f = abs(a - b)$	$f == 0$	`JE`

3.2.3 Inferring Comparisons

Recent advances in both source-available [2, 8] and binary [5, 6, 12] fuzzing depend on instrumentation to record the input values and results of each comparison, as well as the type of comparison being performed (Table 3.1). This information can be used to keep inputs that partially satisfy conditionals [8, 12], detect magic byte values [5], or infer relationships between input and program state [6].

Source-level fuzzers that instrument comparisons generally do so during compilation, in which complete information on a given comparison is available at a single point. As a result, inserting this instrumentation is relatively straightforward. Dynamic binary instrumentation is more complicated, as comparisons in x86 are split across two instructions (Figure 3.3). First, a `cmp` (*compare*) instruction is executed, which fills in the `FLAGS` register with all possible comparison results. Later, a *conditional* instruction is executed, such as a conditional jump or move. This instruction uses results from the most recently run instruction that filled the flags register. Existing approaches to binary

comparison instrumentation only consider the compare instruction. This approach has two weaknesses: (1) while many instructions populate the `FLAGS` register, the results are not always used (false positives), and (2) the type of comparison being performed is defined by the conditional instruction, whose information is not collected.

Because existing works in binary fuzzing primarily consider comparisons for equality, there is little need to know what type of comparison is being performed. Additionally, tools reduce overhead by only instrumenting instructions that are usually used for control flow [5]. However, ReFuzz aims to instrument comparisons for arbitrary fuzzing applications, including previously source-based techniques such as gradient descent, which is too computationally expensive to perform on irrelevant comparisons and requires knowledge of the type of comparison performed.

Our solution to this challenge emulates CPU's behavior on computing and using `FLAGS`. When a compare instruction populates the `FLAGS` register, comparison inputs are temporarily stored in memory. For comparisons that are never used, this represents minimal performance overhead. If a conditional instruction accesses `FLAGS`, these values are reported to the fuzzer by the instrumentation, along with the type of comparison performed (based on the conditional instruction executed). This information is then used by the fuzzer to specifically mutate inputs toward solving the conditionals. In the case of Angora, solutions are found using gradient descent. This approach reduces the number of false positive comparisons reported to the fuzzer, and provides more complete information than existing approaches. Because false positives are reduced, we can also greedily instrument all instructions that populate `FLAGS` without substantial performance overhead.

Chapter 4
Implementation

ReFuzz is implemented as a modified version of the Multiverse binary rewriter. By default, ReFuzz produces instrumentation compatible with the Angora fuzzer, though instrumentation compatible with AFL and LAF-Intel has also been tested. ReFuzz improves the instrumentation capabilities of Multiverse by supporting an instrumentation runtime, written in a high-level language. In addition, performance considerations affected how fuzzing instrumentation was written.

4.1 Instrumentation Runtime

Fuzzing requires a compiled runtime that is invoked during program execution. This runtime communicates with the fuzzing loop, a process that repeatedly invokes the program under test with different inputs. While this runtime can be simple if only basic coverage information is collected, measuring conditionals is more complex. Source-level instrumentation tooling includes this runtime while compiling the binary. When instrumenting an existing binary, however, the instrumentation runtime cannot be as easily incorporated because standard linking procedures are not designed to work on already-linked binaries.

We developed a lightweight framework that allows for an instrumentation runtime to be inserted into a rewritten binary alongside the instrumented code. Symbols from the runtime are then made available to instrumentation hooks that are assembled and inserted inline with the original program. Instrumentation is generated in two steps:

- *Compilation.* Any instrumentation code that does not need to be inserted inline with existing code is compiled. The executable to be rewritten is analyzed and a free region in virtual memory is identified. The compiled instrumentation is then

linked into non-relocatable executable code, which is copied into the output binary. This step supports any source files that are compatible with LLVM. Because it is not possible to reliably edit dynamic library information for the fuzzed program, this instrumentation may not rely on any dynamic libraries; C and Rust code is compiled statically using `musl`, a statically linked implementation of libc.

- *Rewriting.* Instrumentation hooks that go inline with the fuzzed program are inserted before and after relevant instructions (i.e., those that affect control flow). These hooks are small assembly snippets, which are assembled using Keystone [31][1] as the fuzzed program is being rewritten. Before rewriting occurs, symbols are extracted from the instrumentation runtime, allowing these snippets to be linked against the larger instrumentation library. The rewritten instrumentation can therefore access complex functionality, even though it contains only a few contiguous instructions.

The compiled runtime and the rewritten program are output as one executable binary. This allows existing fuzzing tools to use the instrumented program with minimal modification.

4.2 Instrumentation Performance

Fuzzing a program involves passing many inputs into it to explore new behavior and potential bugs. The effectiveness of a fuzzer is, therefore, largely related to two factors: How much information can be obtained about a program from each execution, and how rapidly program executions can be performed (throughput). Source-level instrumentation can be inserted using a compiler's Intermediate Representation (IR) to achieve high throughput on binaries. Furthermore, statically inserted fuzzing instrumentation at the IR level is inserted before optimization is performed; therefore, the following optimizations can optimize the instrumentation for a specific program. On binaries, however, ReFuzz must modify compiled binaries directly and optimizations cannot be easily performed.

One key advantage to instrumenting using an IR is the ability to efficiently use registers. Fuzzing tools that leverage LLVM bitcode can add abstract instructions that are then mapped to unused processor registers. In contrast, ReFuzz can make no assumptions

[1]The Keystone Assembler is an adaptation of the LLVM assembly backend to support custom assembly workflows. ReFuzz uses Keystone to reference the runtime symbol table from instrumentation hooks.

about a program's register use. Since instrumentation code necessarily modifies registers, each instrumentation hook must save the processor state before executing and restore it afterwards. This presents a substantial performance overhead. ReFuzz reduces the impact of this by reducing the register footprint of its instrumentation. Further improvement can be achieved by using static analysis to find free registers (Section 6.1.3).

4.2.1 Taint Tracking

Fuzzing using gradient descent requires determining what portions of the input influence each comparison. This is done using taint tracking, which measures information flow dynamically. Taint tracking instrumentation can either be inserted during compilation, or at runtime using a dynamic instrumentation tool such as Pin [13][2]. In the case of binary rewriting for fuzzing, instrumentation during compilation is not available. For simplicity, ReFuzz uses existing dynamic instrumentation in Angora's fuzzing loop, which is based on libdft [32].

[2]Inserting taint tracking instrumentation during rewriting is also possible, but is outside the scope of this work (Section 6.2).

Chapter 5
Evaluation

Our evaluation aims to demonstrate ReFuzz's ability to instrument binaries for fuzzing with comparable effectiveness to source-level fuzzers. We focus on the following questions: (1) can ReFuzz find the same classes of bugs targeted by source-level fuzzers? (2) what are the performance trade-offs of binary instrumentation on code coverage and bugs found? and (3) how does instrumentation affect binary size, and does this negatively affect performance?

5.1 Fuzzing Instrumentation

We first evaluate ReFuzz on four sample programs (Shown in Figure 5.1) to confirm function and demonstrate the types of conditionals that can be solved to find bugs. These four inputs represent successively more complex programs for bug finding. We compare ReFuzz's performance against AFL [3], LAF-Intel [8], and Angora [2] on four programs:

1. `simple` contains a trivial buffer-overrun bug. Passing an input of sufficient length overwrites the return address on the stack, causing the crash. This was found quickly by each tested tool. Finding this bug does not require constraint solving as implemented by Angora and ReFuzz, as random mutations are sufficient to trigger it.

2. `magic` contains a comparison of the input against a 32-bit magic value. This is similar to the bugs inserted in the LAVA-M corpus, though in this case the magic bytes are compared directly against the input. AFL cannot successfully find this bug in reasonable time, while the other three tools successfully find the bug.

Table 5.1: Time taken to find a crash in each example program. Pairs without a time did not complete successfully within 60 s.

Program	Time to find crash with each fuzzer (s)			
	ReFuzz	Angora	LAF-Intel	QAFL
`simple`	2.7	5.6	0.2	0.3
`magic`	1.8	0.9	38.9[1]	–
`context`	4.0	3.6	–	–
`undef`	1.9	–	–	–

[1] LAF-Intel inconsistently finds the crash within 60 s.

3. `context` has a similar magic byte comparison to that of `magic`, but the bug only manifests within the first call to `foo`; note that since a and b are unsigned integers, $a - b$ is an unsigned subtraction and its result is always nonnegative. LAF-Intel cannot consistently trigger this bug. ReFuzz and Angora both implement context sensitive branch counts, and so both can find this bug.

4. `undef` contains a null-pointer dereference that is easily discernible at compile-time. While both GCC and Clang do not optimize this out by default, the addition of the instrumentation code used by Angora causes further optimization passes to remove the bug. Since Angora's instrumented version no longer contains the bug, Angora cannot find it even though it occurs in the uninstrumented program. In contrast, ReFuzz instruments at the binary level and so faithfully reproduces the functionality of the uninstrumented executable.

Each program was fuzzed by each tested fuzzer for up to one minute. Comparing the time taken by each program to find these bugs (Table 5.1) demonstrates the effectiveness of fuzzing using binary rewriting. ReFuzz found all the bugs that Angora found in a comparable amount of time. LAF-Intel and AFL, which use simpler heuristics to measure and discover new test cases, did not successfully find bugs in the harder sample programs. This shows that ReFuzz is finding similar classes of bugs to Angora.

In some cases, ReFuzz can find bugs in programs that are not found by Angora. In the `undef` program, the bug may be optimized out by some compilers, including the instrumentation pass used by Angora. Bugs due to undefined behavior can be hidden during testing only to appear in production releases, making this bug especially insidious. Instrumentation during compilation inherently modifies the program under test; so any

Table 5.2: Median number of bugs found by each fuzzer on each LAVA-M executable in one hour.

Program	Number of bugs found			
	ReFuzz	Angora	LAF-Intel	QAFL
`base64`	45	43	42	0
`md5sum`	59	56	6	0
`uniq`	29	29	16	0
`who`	258	258	2	0

bugs that are compilation-dependent may not be reproducible using a source-level fuzzer. In contrast, binary fuzzing can be performed on software in its release configuration, and explicitly does not modify the behavior of the base program. This is a key advantage of binary rewriting for fuzzing.

5.2 Performance on Fuzzing Corpora

We continue by comparing the performance of ReFuzz with other fuzzers on a standard bug corpus. For this we use the LAVA-M corpus [16], which is the standard for evaluation in the fuzzing community [2, 5, 6].

Developed by Dolan-Gavitt et al. in 2016, the LAVA-M corpus is a set of utilities from GNU Coreutils (`base64`, `uniq`, `md5sum`, and `who`) with artificially-generated bugs. To generate buggy code, a data-flow analysis is performed on application source code. The analysis finds points in the source code where a function of the input has a specified number of mutations. At these points, values are compared against a generated magic value, such that a memory corruption bug occurs under specific inputs. This technique creates bugs that are non-trivial to reproduce, since the magic values are not directly present in crashing inputs.

We evaluated four fuzzers (ReFuzz, Angora, LAF-Intel, and AFL) on the LAVA-M programs. For each fuzzer-program pair, the program was fuzzed over 20 trials for one hour each. Each fuzzing trial was run in a single thread on an Intel Xeon 6136 with 384GB of RAM. Seed inputs for each program were identical across all fuzzers[1].

Table 5.2 shows the median number of bugs found by each fuzzer in one hour. ReFuzz finds many more bugs than LAF-Intel and AFL, and roughly as many bugs as Angora[2]

[1] All seeds were derived from Angora's published evaluation procedures.

[2] Angora was tested using Pintool-based taint tracking, not compile-time taint tracking. This was done

Table 5.3: Median time overhead of ReFuzz vs. Angora.

Program	Minutes to find bugs		Overhead
	ReFuzz	Angora	
`base64`	11	6	1.8×
`md5sum`	34	11	3.1×
`uniq`	5	1	5.0×
`who`	42	9	4.7×
Overall	92	27	3.4×

on all four LAVA-M programs. This demonstrates that ReFuzz, a binary fuzzer, is comparable in bug-finding capability to those source-available fuzzers.

We additionally analyze the runtime performance of ReFuzz. Figure 5.2a shows the median number of bugs found over time by the four fuzzers tested, and Table 5.3 shows time taken by ReFuzz to find as many bugs as Angora. Qualitatively, programs instrumented with ReFuzz show similar bug-finding behavior to Angora, with new inputs being found roughly linearly until the program has been covered fully. ReFuzz has a 3.4× runtime overhead compared to Angora across the entire corpus. For comparison, Steelix [12], a binary adaptation of LAF-Intel [8], has a 7× overhead compared it its compile-time counterpart. Because fuzz testing is often performed in parallel across many high-powered servers, this is an acceptable trade-off for the capability of finding bugs without access to source code.

To understand the source of overhead on LAVA-M, we additionally compare the bugs found by each fuzzer normalized by invocations of the program under test. This comparison (Figure 5.2b) shows that ReFuzz's overhead is primarily due to individual program invocations taking longer. This overhead can be partially attributed to static binary rewriting inefficiencies (Section 4.2). Additionally, more sophisticated static analysis might reduce this overhead further (Section 6.1).

We also compared the specific bugs found by ReFuzz and Angora across all trials. One bug was found by Angora in at least one trial, but not found by ReFuzz in any trial. ReFuzz found 5 bugs not found by Angora[3]. The substantial overlap in found bugs demonstrates that ReFuzz instruments executables for fuzzing correctly using techniques previously limited to source code. Further, the fact that ReFuzz found

to match the tracking used by ReFuzz so differing performance is solely due to static instrumentation. See Section 6.2.

[3]All 6 non-overlapping bugs were found in the `who` program.

additional bugs not found using source-level instrumentation again suggests that binary fuzzing could, in some cases, have superior bug-finding performance to source-available fuzzing.

5.2.1 Binary Size Overhead

The binary instrumentation techniques used by ReFuzz consider every possible instruction in the original program. Because each instruction must be rewritten in the new binary, instrumented programs can be over 100× the size of the original (Figure 5.3). While this is one of the key weaknesses discussed in the Multiverse paper [15], the overhead has a limited effect on fuzzing due to the way fuzzing processes are created, as well as the layout of the instrumented program. In particular, fuzzers work by loading the program under test into memory a single time. The program then forks off copies with each test input. Since Linux uses Copy-On-Write for forked processes, the text segment (which is not modified at runtime) never gets replicated after the first program execution. The program is loaded into memory each time that taint tracking is run, but this represents a sufficiently small fraction of fuzzing time so that it does not affect overall performance.

While Multiverse (and by extension ReFuzz) must rewrite every instruction in the program for soundness purposes, contiguous instructions in the original program are mostly contiguous in the resulting program. In the best case, when all actual instructions in the original program are contiguous, the rewritten program will output these instructions first before all aliased instructions. At runtime, aliased instructions are never used, and so have minimal effect on cache locality of the program during fuzzing.

```c
int main(int argc, char **argv) {
  char buf[10];
  gets(buf);
  return buf[0] != NULL;
}
```

(a) `simple` - A buffer overrun can be caused by calling `gets`

```c
int main(int argc, char **argv) {
  unsigned int val = 0;
  fread(&val, 4, 1, stdin);

  if (val == 0x12345678)
    val = *(volatile int *)NULL;
  return val;
}
```

(b) `magic` - A specific input causes a null-pointer exception

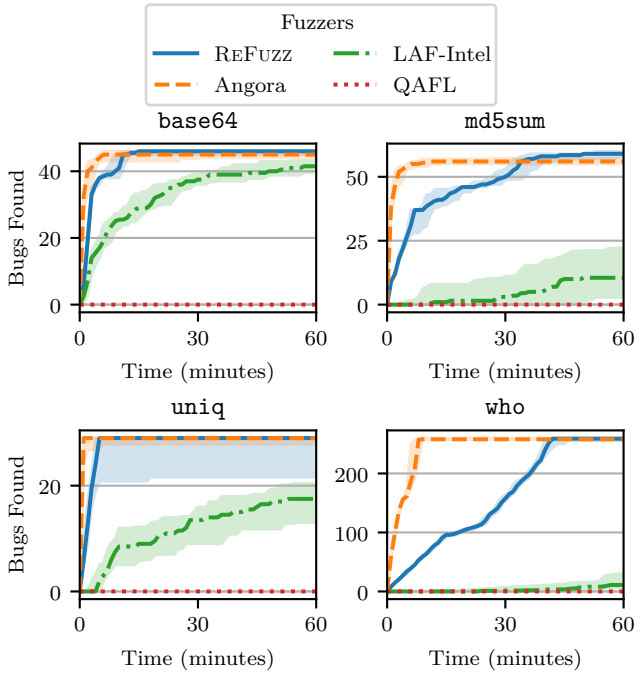
```c
__attribute__((noinline)) volatile
int foo(unsigned int a, unsigned int b) {
  if (a - b < 0x1000)
    if (a < 0x60000100)
      *(volatile int *)NULL;
  return 1;
}

int main(int argc, char **argv) {
  unsigned int a = 0;
  unsigned int ret = 0;
  fread(&a, 4, 1, stdin);

  ret += foo(a, 0x59239472);
  ret += foo(a, 0x70000000);
  ret += foo(a, 0x80000000);
  ret += foo(a, 0x90000000);
  ret += foo(a, 0xa0000000);
  return ret;
}
```

(c) `context` - The bug is only triggered in the first call to `foo`. Note that integers are unsigned.

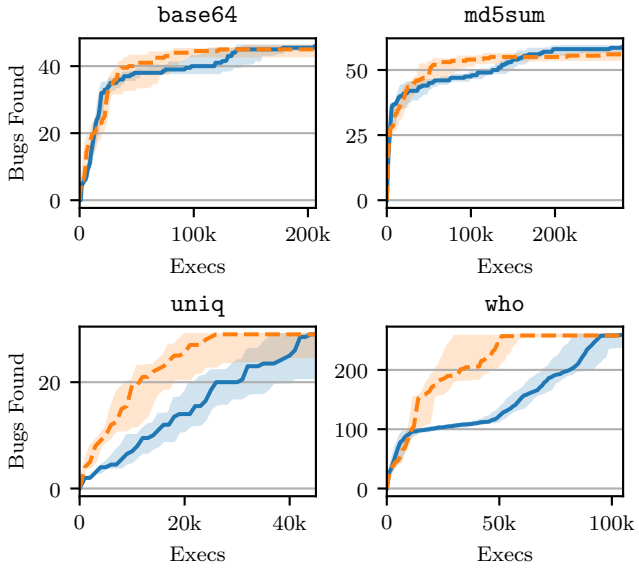
```c
__attribute__((noinline))
int foo(unsigned int a, unsigned int b) {
  if (a - b < 0x1 && a < 0x60000100)
    return *(int *)(a - b);
  return 1;
}

// Same as in `context`
int main(int argc, char **argv) {...}
```

(d) `undef` - The bug may be optimized out by some compilers

Figure 5.1: C programs with progressively more complex bugs



(a) Number of bugs found over time.



(b) Bugs found vs. program executions (ReFuzz and Angora).

Figure 5.2: Bugs found in the LAVA-M corpus by four fuzzers. Shaded areas represent 60% intervals across 20 trials, with lines being the median number of bugs found.

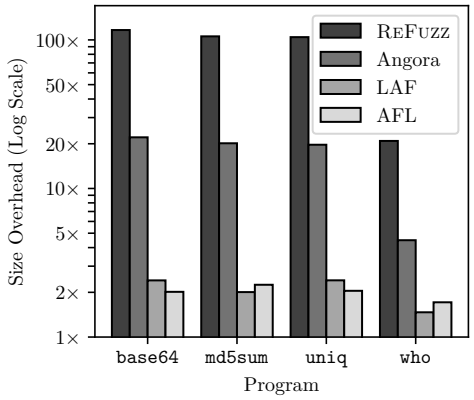


Figure 5.3: Size overhead of programs instrumented by each fuzzer. ReFuzz incurs a substantial size increase due to binary rewriting.

Chapter 6
Discussion

In addition to demonstrating novel adaptation of fuzzing techniques to binaries, ReFuzz is a complex application of binary rewriting. As such, several limitations of existing binary rewriting tools were encountered. These limitations might motivate future work in designing binary rewriting tools.

6.1 Tooling for Binary Rewriting

ReFuzz is based on a recently published binary rewriting tool, Multiverse [15]. As such, analysis was partially limited by the capabilities of this tool, and we found several opportunities for improvement in binary rewriting that might motivate future work. Here we discuss control flow recovery during rewriting as well as general performance considerations.

6.1.1 Control Flow Recovery

Extant binary rewriters allow for individual instructions to be instrumented or modified, but do not expose control flow information during this process[1]. Without this information, instrumentation must be added to all instructions that could potentially be useful for fuzzing. For instance, many arithmetic instructions in x86 modify the `FLAGS` register, though in most cases the effects of these operations are ignored. It is impossible to know at the instruction level whether an instruction's effect on the `FLAGS` register is used or not. Additionally, instrumentation on individual instructions cannot determine what registers currently contain important values. Instrumentation must therefore save and

[1]We consider rewriters that do not rely on symbol or debug information, which simplify recovery of control flow.

restore all registers that are modified during instrumentation, including `FLAGS`, which is deeply connected to the architectural state and is slow to restore.

Recovery of even partial control flow information would solve the above issues. Future heuristic-free binary rewriters might create partial control flow graphs of instructions. This could be performed soundly for non-branching instructions, as well as branch instructions with a known destination. Though the control flow graph would not be complete, sound control flow going forward just a single instruction would, in many cases, determine free registers that can be used by instrumentation code.

6.1.2 Rewriting Performance

ReFuzz works by adding instrumentation to all instructions that affect control flow. In x86 programs, these instructions constitute a large part of the program, and so the resulting instrumented binary can be over 100× the original size (Section 5.2.1). This led to performance issues during the rewriting process, which took upwards of 20 minutes for programs of size 1MB. This time was primarily spent assembling instrumentation hooks. Because conventional linkers could not be used on rewritten programs these instrumentation hooks had to be assembled for each program offset. The speed of this rewriting process could be greatly improved if programs were lifted to assembly source files, which could then be assembled and linked using off-the-shelf tooling. While this approach has been used by rewriters that rely on heuristics to determine instruction offsets, no work has applied it to heuristic-free rewriting. Because instrumentation only needs to be performed once, the performance of the rewriter itself is not a high priority. Further, existing techniques could be easily parallelized to improve performance.

6.1.3 Optimization of Rewritten Programs

Binary rewriters currently work by rewriting each instruction individually. Though this ensures correctness, it may lead to poor runtime performance. By isolating instructions that execute in series, and rewriting them as a single unit, a rewriter might be able to optimize the rewritten program to increase performance. Ideally, these optimizations could be as powerful as those applied during the compilation process, potentially increasing the performance of a rewritten program beyond that of the original. This could also reduce binary size overhead. While techniques for optimized rewriting exist, only a subset of these can likely be applied to binaries soundly without making assumptions about compilation.

6.2 Taint Tracking Approaches

To investigate the applicability of binary rewriting for fuzzing, we used the existing Pintool-based taint tracking included in Angora. This taint tracking mode is experimental, and currently achieves worse code coverage than LLVM-based taint tracking. Because we evaluate both ReFuzz and Angora using Pintool-based tracking, our reported results for the `who` executable differ substantially from those published by Chen et al. [2]. Because dynamic instrumentation for taint tracking has received relatively less attention from the research community, we argue that this weakness is mostly attributable to the experimental nature of Pintool tracking, whereas LLVM-based tracking has received substantial development.

Whereas dynamic insertion of coverage and conditional information incurs a substantial performance overhead, dynamic taint tracking has little effect on the final performance of the fuzzer. This is because, for each conditional, taint tracking is only performed once, followed by many executions of coverage measurement when solving the constraints in the conditional instruction.

Extant works have investigated techniques to improve the effectiveness of taint tracking at the instruction level. Chua et al. [33] proposed a system to infer the taint semantics of assembly instructions by passing a representative set of inputs, which could lead to more accurate dynamic taint tracking approaches. While some information on dataflow may be lost during compilation, we expect that dynamic instrumentation for taint tracking will likely reach performance parity with compile-time instrumentation for fuzzing purposes.

Due to broader applicability, mature taint tracking systems are available for use at compile-time and using dynamic instrumentation. Between these, only the dynamic approach is possible when instrumenting binaries. However, as static binary instrumentation matures there is opportunity to insert taint tracking instrumentation statically. Unlike dynamic instrumentation, a static system could infer taint semantics within each basic block offline, reducing the number of instructions that need to be instrumented at runtime. More efficient taint tracking approaches might also allow this information to be used more substantially in the fuzzing process. For instance, information flow is currently only computed on program inputs that achieve new control flow coverage. If this process were less computationally expensive, one could instrument programs to discover new data propagation that does not necessarily result from modified control flow.

Chapter 7
Related Work

7.1 Fuzzing-specific Obfuscation

By making fewer assumptions for rewriting and instrumentation, ReFuzz aims to be resistant to general-purpose obfuscation techniques. However, it does not address recent advances in program obfuscation specifically targeted at fuzzing.

Initial work in preventing binary fuzzing aimed to detect the specific environments created by common fuzzers such as AFL [34]. While this might frustrate basic effort to attack a given program, the detection methods could be bypassed by changing the fingerprint of the fuzzer. Other efforts reduced the usefulness of fuzzer-found crashes by injecting unexploitable crashes into application source code [35].

Recent works in fuzzing prevention more directly target the techniques used by modern fuzzers. Güler et al. [36] modify programs to hide crashes from fuzzers and prevent constraints from being solved. Jung et al. [37] prevent fuzzing by polluting coverage and taint tracking information, while purposefully imposing a runtime overhead during fuzzing. These methods thwart existing fuzzing approaches, though countermeasures may still be possible, especially when binary analysis techniques are available. For instance, runtime overhead might be overcome by saving intermediate program states during fuzzing, allowing later portions of a program to be executed repeatedly while avoiding artificial delays.

As source and binary fuzzers reach performance parity, withholding source code no longer protects a program from automated exploitation. As such, obfuscation techniques become more important to protect sensitive code bases.

7.2 Other Approaches to Fuzzing

While methods that consider individual concrete executions have shown to be the most effective in practice, other techniques also show promise in specific scenarios. For instance, Driller [4] uses a combination of concrete and symbolic execution to solve conditionals. This approach is compelling: as technologies in symbolic execution [38] and constraint solving [39] are developed, the fuzzer automatically improves.

Other approaches note that the non-convex nature of program behavior is largely similar to problems faced by the machine learning discipline [40]. Programs naturally have *discontinuities*, regions of the input space for which there is a rapid change in program behavior for small input perturbations. Machine learning algorithms face a similar phenomenon in the form of local minima, which thwart training attempts by preventing optimization from working fully. Similarly to ML, smoothing approaches potentially allow a fuzzer to optimize inputs and solve conditionals effectively.

Chapter 8
Conclusions

Our work demonstrates that fuzzing techniques need not be limited by access to source code. Our evaluation shows that ReFuzz achieves source-level fuzzing accuracy with minimal performance overhead. Further, the techniques implemented by our work are generalizable to future works in fuzzing. The software assurance community can use ReFuzz to audit closed-source software similarly to open projects. Yet, binary fuzzing also has negative implications for software security, as withholding source code and obfuscating binaries no longer protects against automated exploitation. As such, advances in binary fuzzing motivate future work in fuzzing-oriented obfuscation.

Bibliography

[1] Klees, G., A. Ruef, B. Cooper, S. Wei, and M. Hicks (2018) "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 2123–2138.

[2] Chen, P. and H. Chen (2018) "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 711–725.

[3] Zalewski, M. (2010), "American Fuzzy Lop: a security-oriented fuzzer," `http://lcamtuf.coredump.cx/afl`.

[4] Stephens, N., J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna (2016) "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." in *NDSS*, vol. 16, pp. 1–16.

[5] Rawat, S., V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos (2017) "VUzzer: Application-aware Evolutionary Fuzzing." .

[6] Aschermann, C., S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz "REDQUEEN: Fuzzing with Input-to-State Correspondence," .

[7] (2019), "google/oss-fuzz," .
URL `https://github.com/google/oss-fuzz`

[8] (2016), "Circumventing Fuzzing Roadblocks with Compiler Transformations," `https://lafintel.wordpress.com`.

[9] Thompson, P. (2015), "aflpin," `https://github.com/mothran/aflpin`.

[10] (2015), "AFL-QEMU," `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[11] (2015), "AFL-dyninst," `https://github.com/Cisco-Talos/moflow/tree/master/afl-dyninst`.

[12] Li, Y., B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu (2017) "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, pp. 627–637.

[13] Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005) "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, ACM, pp. 190–200.

[14] Bellard, F. (2005) "QEMU, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46.

[15] Bauman, E., Z. Lin, and K. W. Hamlen (2018) "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proc. NDSS*, pp. 40–47.

[16] Dolan-Gavitt, B., P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan (2016) "LAVA: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 110–121.

[17] Hanford, K. V. (1970) "Automatic generation of test cases," *IBM Systems Journal*, **9**(4), pp. 242–257.

[18] Forrester, J. E. and B. P. Miller (2000) "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th USENIX Windows System Symposium*, vol. 4, Seattle, pp. 59–68.

[19] Godefroid, P. (2020) "Fuzzing: hack, art, and science," *Communications of the ACM*, **63**(2), pp. 70–76.

[20] Oehlert, P. (2005) "Violating assumptions with fuzzing," *IEEE Security & Privacy*, **3**(2), pp. 58–62.

[21] Lattner, C. and V. Adve (2004) "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, p. 75.

[22] Dinesh, S., N. Burow, D. Xu, and M. Payer "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," .

[23] Wartell, R., Y. Zhou, K. W. Hamlen, and M. Kantarcioglu (2014) "Shingled graph disassembly: Finding the undecideable path," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, pp. 273–285.

[24] Balakrishnan, G. and T. Reps (2010) "WYSINWYX: What you see is not what you eXecute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **32**(6), p. 23.

[25] Wang, S., P. Wang, and D. Wu (2015) "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 627–642.

[26] ——— (2016) "Uroboros: Instrumenting stripped binaries with static reassembling," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, IEEE, pp. 236–247.

[27] Song, D., D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena (2008) "BitBlaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, Springer, pp. 1–25.

[28] Bao, T., J. Burket, M. Woo, R. Turner, and D. Brumley (2014) "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 845–860.

[29] Wang, F. and Y. Shoshitaishvili (2017) "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, IEEE, pp. 8–9.

[30] Wang, R., Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna (2017) "Ramblr: Making Reassembly Great Again." in *NDSS*.

[31] "Keystone – The Ultimate Assembler," `http://www.keystone-engine.org/`.

[32] Kemerlis, V. P., G. Portokalidis, K. Jee, and A. D. Keromytis (2012) "libdft: Practical dynamic data flow tracking for commodity systems," in *ACM Sigplan Notices*, vol. 47, ACM, pp. 121–132.

[33] Chua, Z. L., Y. Wang, T. Baluta, P. Saxena, Z. Liang, and P. Su (2019) "One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics." in *NDSS*.

[34] GÖRANSSON, D. and E. EDHOLM "Escaping the Fuzz," .

[35] Hu, Z., Y. Hu, and B. Dolan-Gavitt (2018) "Chaff Bugs: Deterring Attackers by Making Software Buggier," *arXiv preprint arXiv:1808.00659*.

[36] Güler, E., C. Aschermann, A. Abbasi, and T. Holz (2019) "ANTIFUZZ: impeding fuzzing audits of binary executables," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1931–1947.

[37] Jung, J., H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim (2019) "FUZZIFICATION: anti-fuzzing techniques," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1913–1930.

[38] Cadar, C., D. Dunbar, D. R. Engler, et al. (2008) "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, pp. 209–224.

[39] De Moura, L. and N. Bjørner (2008) "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 337–340.

[40] She, D., K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana (2019) "NEUZZ: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 803–817.