Performant Binary Fuzzing without Source Code using Static Instrumentation

Eric Pauley
University of Wisconsin–Madison
epauley@cs.wisc.edu

Gang Tan
The Pennsylvania State University
gtan@psu.edu

Danfeng Zhang
The Pennsylvania State University
zhang@cse.psu.edu

Patrick McDaniel
The Pennsylvania State University
mcdaniel@cse.psu.edu

***Abstract*—Advancements in fuzz testing have achieved the ability to quickly and comprehensively find security-critical faults in software systems. Yet, some of these techniques rely on access to 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 RWFUZZ on the LAVA-M dataset, achieving the same effectiveness as a best-in-class source-available fuzzer with a $3.4\times$ 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.**

I. INTRODUCTION

Fuzz testing (fuzzing), an automated technique that generates interesting 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 [11], as well as more effective techniques for generating new inputs [31], [24], [22], [11], [6]. Many popular projects now use fuzz testing to prevent vulnerabilities from affecting end users [5], and software vulnerability discoveries can often be attributed to crashes found by fuzzers [31].

Fuzzers often leverage source code access to generate efficient fuzzing instrumentation and rapidly discover bugs [31], [3], [11]. For instance, Chen et al. recently introduced Angora [11], 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 is not available, many techniques are no longer available. The primary reasons for this are twofold:

1) Fuzzers work by inserting instrumentation into programs. Modern compilers provide 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 [6], [22], [25], [2], [1], [19] to date have all relied on dynamic instrumentation frameworks such as Pin [20] or QEMU [10], 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, RWFUZZ, instruments binaries for fuzzing using techniques previously limited to source-available fuzzers. RWFUZZ builds on recent published work in binary instrumentation [9], and implements additional approaches relevant to fuzzing instrumentation. RWFUZZ-instrumented binaries can be fuzzed by existing tools with minimal modification. This is a substantial departure from works such as Angora [11] and RedQueen [6], which seek to introduce *new* fuzzing techniques rather than allow proven ones to be applied in broader contexts.

Our work in static binary fuzzing faces three challenges: (1)

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, RWFUZZ instruments binaries with source-level accuracy while reducing performance overhead over previous approaches.

We evaluate RWFUZZ against source- and binary-level fuzzers. On manually-inserted bugs in C programs, RWFUZZ finds more bugs than any other fuzzer tested, including a bug that is compiler-dependent and so is missed by source-based fuzzers. On the LAVA-M corpus [13], RWFUZZ found similar bugs to Angora's Pintool-based taint tracking mode without access to source code. On LAVA-M, RWFUZZ incurred a $3.4\times$ overhead compared to Angora, which is lower than other binary fuzzing approaches using dynamic instrumentation. In some cases, RWFUZZ's binary instrumentation finds more bugs on average than Angora's published results regardless of mode used. Our evaluation demonstrates that RWFUZZ achieves comparable bug-finding performance to source-level fuzzers, without requiring 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 RWFUZZ on manually- and automatically-generated bugs in programs, matching the bugs found by a modern source-available fuzzer with minimal overhead.

RWFUZZ 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.

II. 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) [16]. 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 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 [14]. 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 [15].

Black-box fuzzers mutate inputs without knowledge of the program [21]. Recent approaches use increased access to the program to effectively mutate inputs. These *grey-box fuzzers* instrument the program to monitor execution and determine which mutations invoke new behavior. We classify grey-box fuzzers into source-available fuzzers (i.e., requiring source code access) and binary fuzzers (i.e., using binary instrumentation).

A. 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 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 [11]. While contemporary fuzzers such as REDQUEEN [6] achieve competitive performance, Angora's flexible approach makes it emblematic of state-of-the-art 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*. Control flow coverage of each program execution is recorded. If two inputs cause similar coverage, only one is kept. This effectively curates a minimal set of inputs that trigger all discovered behavior.
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 [18] DataFlowSanitizer) or using a dynamic binary instrumentation tool such as Intel's Pin [20].
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 [3] and Angora [11] record the inputs and results of each comparison, though they employ varying methods to mutate inputs based on these values.

Based on this instrumentation, many source-level fuzzers differ in how they craft new inputs from the collected information. LAF-Intel [3] 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. Concrete values used in comparisons are recorded, along with the offsets in the input that contribute to the conditional. This requires precise instrumentation of each conditional, as well as taint tracking to determine which input bytes of the input affect execution.

B. 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 stripped 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 [6], [22], [25], [2], [1], [19] use a binary instrumentation framework such as Pin [20], QEMU [10], or DynInst [1]. Dynamic instrumentation tools have inherent performance overhead, as they must interpret the executable and determine instrumentation points at runtime. This overhead can be reduced, though not eliminated, using just-in-time compilation.
2) *Add instrumentation statically*. A program can be statically rewritten to include fuzzing instrumentation within the executable. This requires static analysis to extract instruction and control-flow information. 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. RWFUZZ demonstrates techniques for static binary fuzzing instrumentation.
   Dinesh et al. [12] explore applications of static instrumentation for fuzzing. They focus on ensuring soundness and performance during the rewriting process. In contrast, our work explores challenges relating specifically to applying source-available fuzzing techniques to binaries.

Some binary fuzzers focus on reproducing source-level techniques using approximation. AFL-DynInst [1] instruments dynamically for AFL [31], and Steelix [19] implements comparable techniques to LAF-Intel [3] dynamically. Such works generally aim to find the same crashes per execution as their source-available counterparts, while minimizing execution time overhead. When techniques do not benefit from source instrumentation, fuzzers demonstrate techniques on binaries. REDQUEEN [6] and VUzzer [22] are directly implemented using dynamic instrumentation.

C. Static Binary Instrumentation

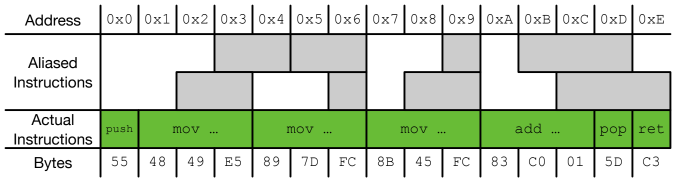
Our work in binary fuzzing builds on developments in static binary instrumentation, which incurs several key challenges.

Binaries contain code and data. One of the greatest challenges to static instrumentation is determining the meaning of these bytes without running the program, especially when binaries are stripped of debug info. A rewriter must obtain a correct disassembly of the executable and modify the instructions without breaking original functionality. Different frameworks approach this problem with varying success [28], [29], [27].

```
int foo(int a)
    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.

Fig. 1: x86 machine code is unaligned: original instructions are aliased by additional valid instructions. Determining used instructions statically is undecidable [30].

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 [7]. Figure 1 shows an example function and its resulting binary, along with instructions that exist in the binary but not in the original program.

Most compilers align instructions based on function boundaries, and UROBOROS uses this to extract valid instructions [28], [29]. The authors note that their tool can become more effective as function identification improves. However, even advanced function boundary extraction methods fail to identify boundaries precisely under adversarial conditions such as disguised function alignment [23], [8]. Binary fuzzers are an adversarial tool, and must be resistant to these techniques. Using a rewriting tool that depends on function boundaries is not viable.

Recent binary analysis tooling has placed emphasis on determining instruction offsets without relying on function identification [26]. Although, as Wartell et al. note [30], sound static disassembly is undecidable in general, these techniques can disassemble most programs. As a result rewriters such as RAMBLR [27] have been developed that rely on these new techniques. RAMBLR makes fewer assumptions about instruction offsets, so works correctly on more programs, though it is not entirely resistant to obfuscation and relies on heuristics to determine instruction locations.

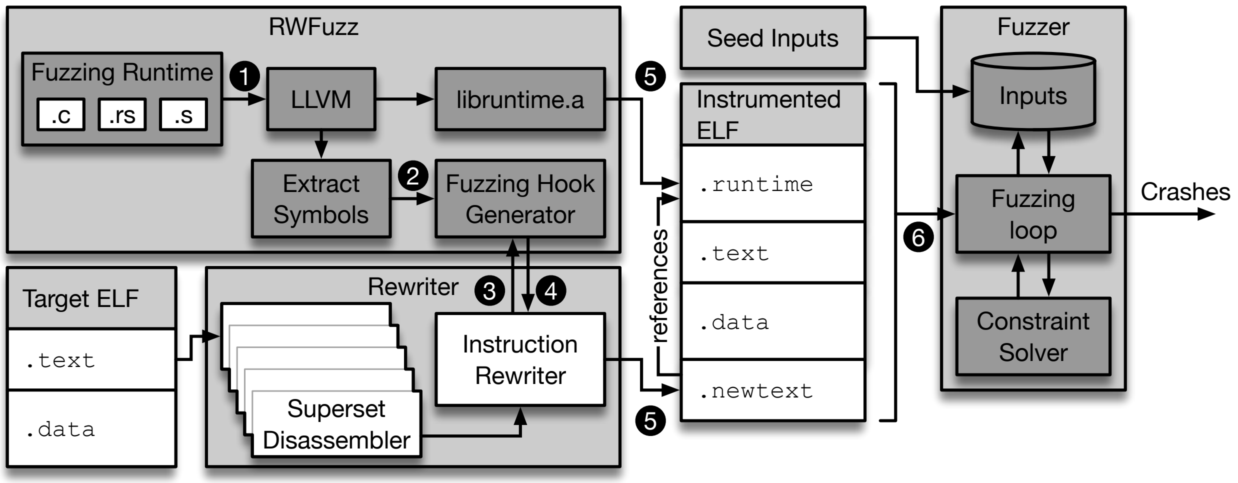


Fig. 2: System for fuzz testing using RWFUZZ. RWFUZZ generates fuzzing instrumentation within MULTIVERSE.

An alternate approach has been proposed by Bauman et al. [9]. They argue that, instead of relying on heuristics to determine what instructions are disassembled, a rewriter can disassemble all valid instructions in the binary. This is a superset of the actual useful code in the executable, motivating name *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. RWFUZZ builds on MULTIVERSE, analyzing each instruction individually and inserting fuzzing instrumentation.

III. BINARY REWRITING FOR FUZZING

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, RWFUZZ, integrates with an existing binary rewriter (MULTIVERSE [9][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 sourced from an existing fuzzer. Figure 2 presents this integration, which consists of several steps:

❶ The *runtime* (provided by the fuzzer) is compiled using LLVM [18]. This runtime is written in a high-level language such as C or Rust, and linked with the target program at compile-time. In contrast, RWFUZZ produces a statically-linked binary. This step allows the runtimes of existing fuzzers to be used with minimal modification.

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

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

❸ Each possible instruction in the target ELF file is processed by RWFUZZ as a candidate for instrumentation. RWFUZZ generates instrumentation assembly for control flow instructions and comparisons, and inserts calls to functions in the 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 a fuzzer, which fuzzes the program as if it were compile-time instrumented.

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 RWFUZZ-instrumented programs automatically. RWFUZZ's ability to instrument binaries statically is a key strength over existing binary fuzzing techniques.

A. Challenges to Static Instrumentation

Compilation creates compact programs that run efficiently. However, it complicates instrumentation, as information about data and control flow is lost. To statically instrument programs for fuzzing, RWFUZZ must reverse-engineer, infer, or approximate source-level information. Several challenges in this space are unique to fuzzing:

1) *Coverage Measurement*. Source-available fuzzers instrument control flow edges during compilation, allowing code coverage measurement. Likewise, extant binary fuzzers record this information dynamically. Inserting this information statically in binaries is not straightforward, as the

```
1 int foo(int a) {
2   if (a > 2)
3     if (a < 8)
4       a *= a;
5   return a + 10;
6 }
```

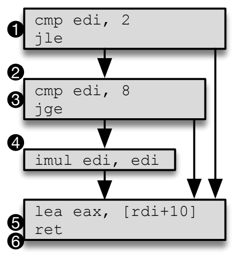


Fig. 3: Disassembly and control flow of a simple function. Basic blocks representing lines 4 and 5 in the C source are not separated by control flow instructions. Circled numbers are points where RWFUZZ inserts coverage instrumentation.

locations of basic blocks in the program cannot be soundly inferred. RWFUZZ implements coverage instrumentation that maintains precision with minimal runtime overhead.

2) *Context Sensitivity*. Recent work [11] has shown that tracking control-flow edges alone is insufficient to fully exercise a program, opting to additionally track the *context* of control flow edges based on the current function call stack. The compiler exposes function call information, so adding this is straightforward with source code. Binary instrumentation cannot soundly access this information, so RWFUZZ uses a shadow stack to track context.
3) *Inferring Comparisons*. Source-level fuzzers record the inputs and results of comparisons, which guides mutation strategies. Information on comparisons is also made available to compiler instrumentation. At the binary level, these comparisons cannot, in general, be inferred statically. RWFUZZ tracks comparisons more effectively than extant dynamic solutions by mirroring the processor state, with accuracy comparable to source-level instrumentation.

B. Approach

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 [31], [11], [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, 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. RWFUZZ 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 the set (i.e., counted as two different edges), this will not affect which inputs are kept.

As a result of this insight, we strategically over-instrument all possible control flow instructions. RWFUZZ inserts coverage instrumentation before and after each control flow instruction without determining basic block boundaries. In Figure 3, 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, RWFUZZ 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, 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 [31], [3].

a) 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, RWFUZZ 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.

2) Context Sensitivity: Control flow edge coverage alone does not fully describe program behavior. In an example (Figure 5c) 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 [11]. 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, RWFUZZ 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. Function addresses saved in the shadow stack are used to compute a function call context (by chained hashing of each call in the stack), which is then incorporated into coverage information. In this way (e.g., Figure 5c), calls to functions in different contexts are treated separately for coverage purposes.

3) Inferring Comparisons: Advances in source-available [3], [11] and binary [22], [19], [6] fuzzing depend on instrumentation to record the input values and results of each comparison,

```
int min(int a, int b) {
  if (a > b)
      return b;
  return a;
}
```

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

Fig. 4: Disassembly of a `min` function. Control flow is split into two instructions, separated by an unrelated instruction.

TABLE I: Comparisons and constraints in x86 assembly. Each x86 conditional instruction maps to a constraint that can be solved using gradient descent. (Derived from [11])

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

as well as the type of comparison being performed (Table I). This information can be used to keep inputs that partially satisfy conditionals [3], [19], detect magic byte values [22], 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. Dynamic binary instrumentation is more complicated, as comparisons in x86 are split across two instructions (Figure 4). 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, facing 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 [22]. However, RWFuzz 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 emulates CPU behavior on computing `FLAGS`. When a compare instruction populates the `FLAGS` register, comparison inputs are stored in memory. If a conditional instruction accesses `FLAGS`, these values are reported to the fuzzer, along with the type of comparison performed (based on the instruction executed). This information is then used by the fuzzer to specifically mutate inputs. In the case of Angora, solutions are found using gradient descent. This approach reduces the 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.

IV. Implementation

RWFuzz is implemented as a modified version of the Multiverse binary rewriter. By default, RWFuzz produces instrumentation compatible with the Angora fuzzer, though instrumentation compatible with AFL and LAF-Intel has also been tested. RWFuzz 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.

A. 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.* Instrumentation that does not need to be inserted inline 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 in the output binary. 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 [4] 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 thus 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.

B. *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, RWFUZZ 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, RWFUZZ can make no assumptions 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. RWFUZZ reduces the impact of this by using a minimal set of unique registers for its instrumentation.

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 [20]. In the case of binary rewriting for fuzzing, instrumentation during compilation is not available. For simplicity, RWFUZZ uses existing dynamic instrumentation in Angora's fuzzing loop, which is based on libdft [17].

V. EVALUATION

Our evaluation aims to demonstrate RWFUZZ's ability to instrument binaries for fuzzing with comparable effectiveness to source-level fuzzers. We focus on the following questions: (1) can RWFUZZ 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?

A. *Fuzzing Instrumentation*

We first evaluate RWFUZZ on four sample programs (Shown in Figure 5) 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 RWFUZZ's performance against AFL [31], LAF-INTEL [3], and Angora [11] on four programs:

```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

Fig. 5: C programs with progressively more complex bugs

1) `simple`: a trivial buffer-overrun bug. This was found quickly by each tested tool. Finding this bug does not require constraint solving as implemented by Angora (instrumented from source or using RWFUZZ), as random

TABLE II: 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)			
	RWFUZZ	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 a crash within 60 s.

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

Program	Number of bugs found				
	RWFUZZ	Angora	LAF-Intel	QAFL	RetroWrite[1]
`base64`	45	43	42	0	2
`md5sum`	59	56	6	0	0
`uniq`	29	29	16	0	2
`who`	258	258	2	0	0

[1] Median bugs found over 5 trials of 24 hours [12].

TABLE IV: Median time overhead of RWFUZZ vs. Angora.

Program	Minutes to find bugs		Overhead
	RWFUZZ	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×

mutations are sufficient to trigger it.

2) `magic`: contains a comparison against a 32-bit magic value. This is similar to the bugs inserted in the LAVA-M corpus. AFL cannot successfully find this bug in reasonable time, while the other tools successfully find the bug.
3) `context`: similar to `magic`, but only crashes within the first call to `foo`; 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. RWFUZZ and Angora both support context-sensitive branch counts, so both can find this bug.
4) `undef`: a null-pointer dereference 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. Angora cannot find it even though it occurs in the uninstrumented program. RWFUZZ instruments at the binary level and so reproduces the uninstrumented functionality.

Each program was fuzzed by each tested fuzzer for one minute. Comparing the time taken by each program to find these bugs (Table II) demonstrates the effectiveness of fuzzing using binary rewriting. RWFUZZ found all bugs that Angora found in comparable 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 RWFUZZ is finding similar classes of bugs to Angora.

In some cases, RWFUZZ 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 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.

B. Performance on Fuzzing Corpora

We continue by comparing the performance of RWFUZZ with other fuzzers on a standard bug corpus. For this we use the LAVA-M corpus [13]. Angora's optimization towards LAVA-M makes it an ideal corpus for evaluating RWFUZZ. 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 bugs are introduced as magic values based on dataflow analysis, and are non-trivial to reproduce, since the magic values are not directly present in crashing inputs.

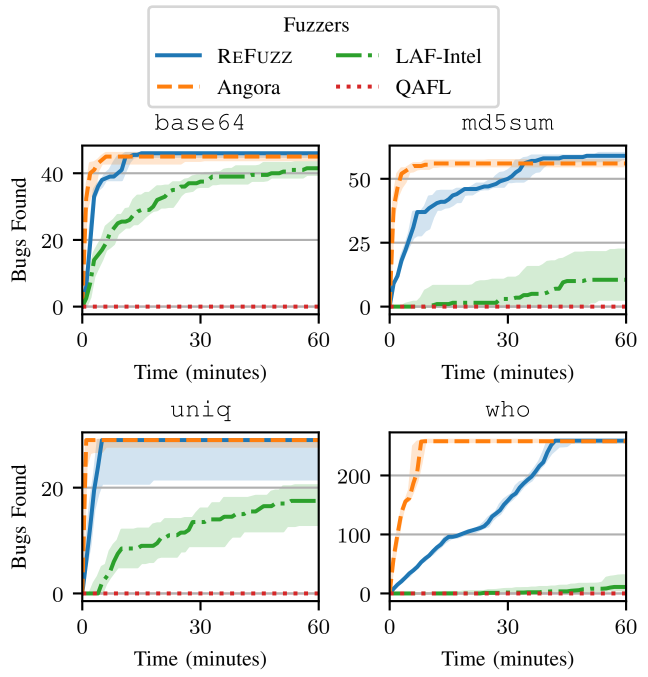
We evaluated four fuzzers (RWFUZZ, Angora, LAF-INTEL, and AFL) on the LAVA-M programs. We additionally compare against the published results of RetroWrite [12], though the two tools have differing goals. For each fuzzer-program pair, the program was fuzzed over 20 trials for one hour each. Each trial 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[2].

Table III shows the median number of bugs found by each fuzzer in one hour. RWFUZZ finds many more bugs than LAF-INTEL and AFL, and roughly as many bugs as Angora[3] on all four LAVA-M programs. This demonstrates that RWFUZZ's instrumentation is comparable to equivalent source-available instrumentation. Additionally, because RetroWrite implements AFL-style instrumentation, RWFUZZ's instrumentation outperforms it in terms of actual bugs found.

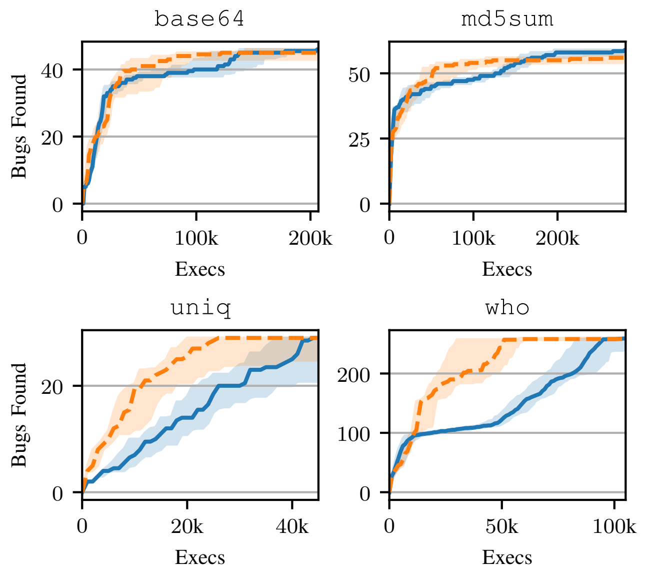
We also analyze the runtime performance of RWFUZZ. Figure 6a shows the median number of bugs found over time by the four fuzzers tested, and Table IV shows time taken by RWFUZZ to find as many bugs as Angora. Programs instrumented with RWFUZZ show similar behavior to Angora, with new inputs found roughly linearly until the program is covered fully. RWFUZZ has a 3.4× runtime overhead compared

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

[3] Angora was tested using Pintool-based taint tracking, not compile-time taint tracking. This was done to match the tracking used by RWFUZZ so differing performance is solely due to static instrumentation.



(a) Number of bugs found over time.



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

Fig. 6: 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.

to Angora across the corpus. For comparison, Steelix [19], a binary adaptation of LAF-Intel [3], has a 7× overhead compared it its compile-time counterpart. Because fuzz testing is often performed in parallel across many servers, this is an acceptable trade-off for the capability of finding bugs without access to source code.

We additionally compare the bugs found by each fuzzer normalized by invocations of the program under test. This comparison (Figure 6b) shows that RWFuzz's overhead is primarily due to individual program runs taking longer. The overhead can be partially attributed to static binary rewriting inefficiencies (Section IV-B).

We also compared the specific bugs found by RWFuzz and Angora across all trials. One bug was found by Angora and not by RWFuzz, while RWFuzz found 5 bugs not found by Angora. The substantial overlap in found bugs demonstrates that RWFuzz instruments executables for fuzzing correctly using techniques previously limited to source code.

VI. Conclusions

Our work demonstrates that fuzzing techniques need not be limited by access to source code. Our evaluation shows that RWFuzz 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 RWFuzz to audit closed-source software similarly to open projects. Yet, binary fuzzing also has negative implications for software security, as withholding source code no longer protects against automated exploitation. As such, advances in binary fuzzing motivate future work in fuzzing-oriented obfuscation.

References

[1] "AFL-dyninst," https://github.com/Cisco-Talos/moflow/tree/master/afl-dyninst, 2015.
[2] "AFL-QEMU," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2015.
[3] "Circumventing fuzzing roadblocks with compiler transformations," https://lafintel.wordpress.com, 2016.
[4] "Keystone – The Ultimate Assembler," http://www.keystone-engine.org/, 2016.
[5] "google/oss-fuzz," Nov. 2019. [Online]. Available: https://github.com/google/oss-fuzz
[6] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.
[7] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.
[8] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 845–860.
[9] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proc. NDSS*, 2018, pp. 40–47.
[10] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
[11] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.
[12] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 128–142. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2020.00009
[13] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.
[14] 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*, vol. 4. Seattle, 2000, pp. 59–68.
[15] P. Godefroid, "Fuzzing: hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[16] K. V. Hanford, “Automatic generation of test cases,” *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

[17] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, “libdft: Practical dynamic data flow tracking for commodity systems,” in *ACM Sigplan Notices*, vol. 47, no. 7. ACM, 2012, pp. 121–132.

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

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

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[21] P. Oehlert, “Violating assumptions with fuzzing,” *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[22] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing.” in *NDSS*, vol. 17, 2017, pp. 1–14.

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

[24] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.” in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[25] P. Thompson, “aflpin,” https://github.com/mothran/aflpin, 2015.

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

[27] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, “Ramblr: Making reassembly great again.” in *NDSS*, 2017.

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

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

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

[31] M. Zalewski, “American fuzzy lop: a security-oriented fuzzer,” https://lcamtuf.coredump.cx/afl/, 2010.

Acknowledgement

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE1255832. 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.