THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PERFORMANT BINARY FUZZING WITHOUT SOURCE CODE USING STATIC INSTRUMENTATION

ERIC PAULEY
FALL 2019

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Computer Science and Electrical Engineering
with honors in Computer Science

Reviewed and approved* by the following:

Patrick Drew McDaniel
William L. Weiss Professor of Information and Communications Technology
Thesis Supervisor

Danfeng Zhang
Professor of Computer Science and Engineering
Thesis Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

Fuzz testing (fuzzing), a technique for automatically finding exploitable bugs in programs, has seen increased popularity in the security community. While fuzzing techniques can efficiently discover new program behavior, modern fuzzing techniques are largely limited to the analysis of programs with source code available. We investigate the application of state-of-the-art fuzzing techniques to binary programs without source code, using static binary rewriting to modify the programs without recompiling them. Our tool, REFUZZ, allows off-the-shelf binaries to be analyzed using fuzzing techniques that were previously limited to source code. We evaluate our tool against source-available and binary-level fuzzers, and find that REFUZZ can discover similar and, in some cases, more bugs than a recently-published source-level fuzzer. Our work demonstrates the value of binary analysis techniques for fuzzing, and realizes a tool that will allow the security community to meaningfully analyze more software.

Table of Contents

List of Figures

List of Tables

Acknowledgements

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) is a promising technique for identifying bugs in programs. Recent advances in fuzzing can detect complex corner cases in software that would otherwise be difficult to manually trigger. As a result, fuzzing has become an essential part of assurance for security-critical software.

Fuzzers are an important technique in the threat space of software exploitation. While most fuzzers are used by application developers to discover bugs and improve the security of their software, fuzzers can also be used by adversaries to discover and exploit bugs in other systems. When used by an adversary for software exploitation, the threat space of fuzzers can be considered in terms of the adversary's access to software, as well as the complexity of the desired exploit. Whereas developers who fuzz their own software usually have access to source code, an adversary who wishes to exploit software may have no access to source code, debug symbols, or unobfuscated binaries.

To date, most fuzzers focus on software assurance, and so limit themselves to fuzzing applications with source code to achieve strong performance. In contrast, binary fuzzing faces several additional challenges. Source-level semantics, such as basic block information, are not readily available, and in many cases obfuscation of released binaries can further frustrate static analysis. As such, extant approaches in binary fuzzing rely on dynamic instrumentation. This vastly increases the overhead associated with fuzzing, and prevents the analyses that make modern source-level fuzzers so effective.

In this work, we adapt fuzzing techniques that were previously only possible with source code to binaries. We introduce the application of static binary instrumentation to fuzzing, achieving comparable soundness guarantees to dynamic binary instrumentation with low runtime overhead. Additionally, we demonstrate techniques that approximate state-of-the-art source-level fuzzing analysis on stripped binaries. Our tool, REFUZZ, functions as an instrumentation frontend for the Angora fuzzer [10], replacing the existing instrumentation that is inserted during compilation using

only the compiled binary. Our technique uses heuristic-free binary rewriting techniques to produce correct instrumentation even in the presence of obfuscation.

We evaluate REFUZZ on the programs tested in the original Angora paper, namely the LAVA-M dataset. REFUZZ performs strongly on the LAVA-M dataset without access to program source code.

In summary, we make the following contributions:

1. We demonstrate how fuzzing techniques vary within the larger field of software exploitation by describing a threat space for automated software exploitation. In this threat model, fuzzers play an important role in the initial analysis of a target program.

2. We adapt the source-level analyses used by Angora to stripped binaries. We use binary analysis techniques to achieve comparable analysis without relying on source code.

3. We demonstrate the first use of static binary instrumentation for fuzzing, using binary rewriting techniques to produce correct binaries without heuristics.

4. We evaluate our tool, REFUZZ, on several example programs and synthetic bug corpora.

Chapter 2

Background

Software exploitation involves finding inputs to a piece of software that cause unintended or exploitable behavior. The generation of these exploits has increasingly been performed automatically, a process known as automated exploit generation [9]. These techniques can be arranged into a threat space based on their requisite access to the target software and the strength of the exploit obtained. Figure 2.1 demonstrates the space of possible threat models in automated software exploitation. An adversary can have degrees of access to the software ranging from ability to pass inputs and receive outputs to full source access. Using this capability, the adversary aims to achieve an exploit of varying strength.

The adversarial goals in our threat space are arranged in strictly increasing order of strength, and techniques that achieve strong exploits often rely on weaker exploits found by some other means. For instance, BOPC [13] uses binary access to escalate a write-anywhere primitive (an integrity violation) into remote code execution. This integrity violation must be produced via some other means, such as manual analysis of a crash found by a binary fuzzer.

Fuzzing is an important tool in software exploitation because it is usually the first step in an exploit chain, running on programs with no prior knowledge and finding exploitable crashes. As a consequence, fuzzing has seen increased attention in recent years, with modern fuzzers performing deeper program analysis and finding complex exploitable bugs without manual effort.

The most basic form of fuzzing is black-box fuzzing, a technique that involves generating random inputs and passing them to a program to discover crashes [18]. More recent approaches to fuzzing have maintained the same adversarial goal (i.e., discovering crashing inputs) while relying on increased access to the program under test to increase effectiveness. These *grey-box fuzzing* approaches add instrumentation to the program to monitor execution. AFL [29] popularized the most prominent such technique, measuring code coverage to craft inputs that exercise more of the program and therefore discover more bugs. Nearly all fuzzers now employ some form of this coverage tracking, which has been adapted for use with our without source code available.

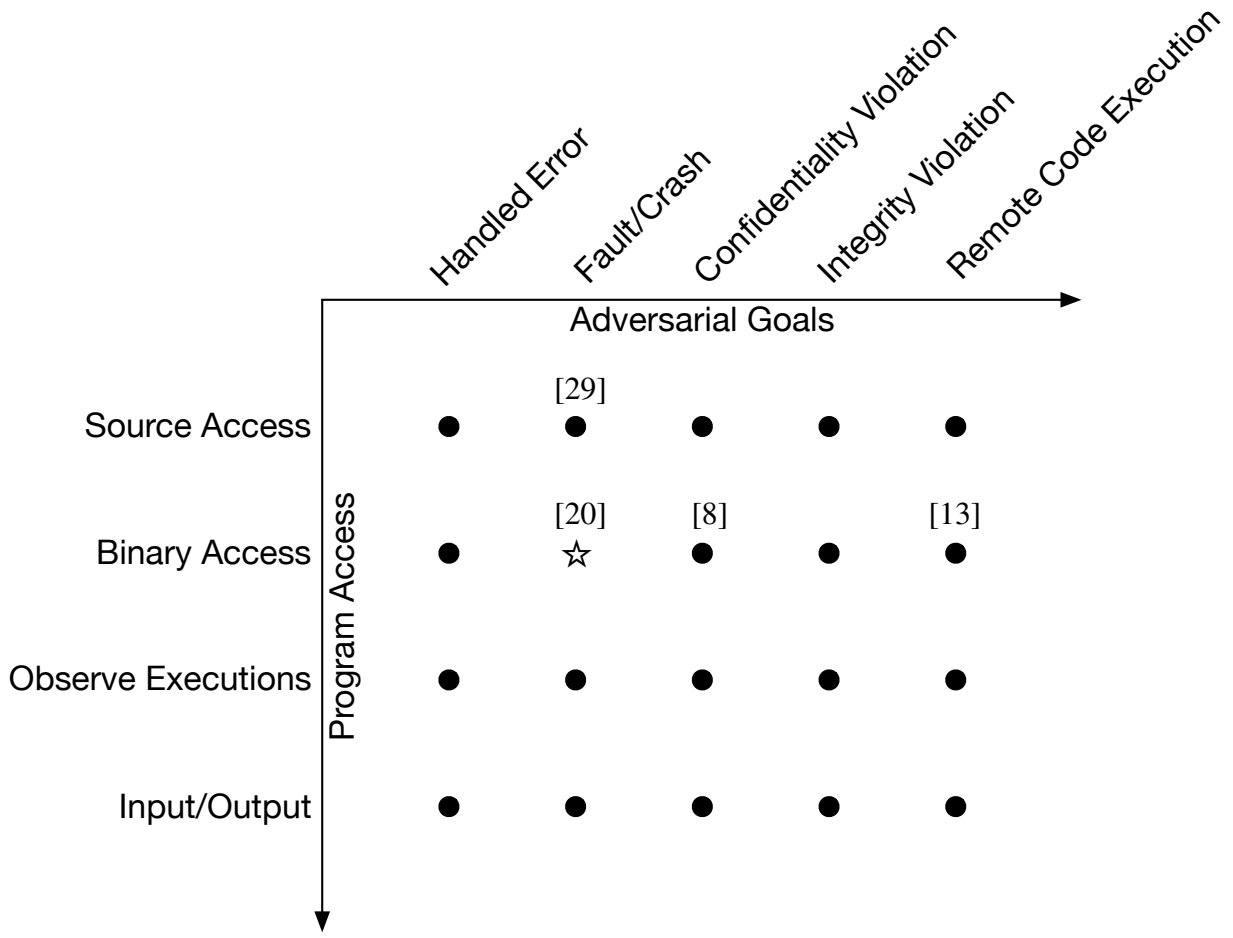


Figure 2.1: The threat space of automated software exploit generation. This work describes a novel method for discovery of crashes with binary access (represented by the star). While prior works [20] have considered this threat model, we apply stronger techniques previously limited to source code.

Further advances in fuzzing have focused on mutating inputs to achieve more code coverage. Initially, work in this space proceeded by splitting conditionals to make them more likely to be solved by random the random mutation in AFL, both through source code [3] or dynamic binary instrumentation [12, 16]. Approaches using symbolic execution have also been proposed [22], though real-world programs are often too complex for this analysis to achieve strong results. Further techniques have also been explored, such as extracting magic values from executables [19] or applying knowledge of binary formats to produce valid inputs [4].

2.1 Recent Advances in Fuzzing

Recent approaches to fuzzing leverage advanced program analysis techniques to fuzz executables effectively. Our work builds on new fuzzing techniques implemented by Angora [10]. Angora's techniques make few assumptions about the underlying executable, yet can quickly discover new execution paths and uncover complex bugs. While contemporary fuzzers such as REDQUEEN [4]

have achieved competitive performance, Angora's combination of performance and generalizability make it arguably the best source-level fuzzer available.

To understand the challenges and opportunities in applying source-level fuzzing techniques to binaries, one must first understand the techniques used by Angora. Like most source-available fuzzers, Angora analyzes and instruments programs at compile-time. This allows it to extract useful information about the program's behavior that is not readily apparent in compiled binaries. Angora's main source-dependent instrumentation employs the following techniques:

1. *Dynamic taint tracking*. Angora tracks how each byte of the input affects the application state. This allows the fuzzer to only mutate input bytes that directly affect other instrumentation.

2. *Comparison instrumentation*. Each time a comparison is made, Angora records the compared values, the type of comparison made, and the pertinent input bytes that influence the conditional.

3. *Function exploitation*. Functions often take pointer arguments, which can cause undefined or buggy behavior when set to `null`. Angora instruments known exploitable functions to record parameters, allowing the fuzzer to mutate the input to cause `null` pointers to be passed. This instrumentation relies on manually-produced descriptions of exploitable function parameters, but otherwise is not application-specific.

The above instrumentation records points in the execution where program behavior could be modified by a mutated input. The fuzzer then uses gradient descent, an optimization technique, to iteratively mutate the program input based on this instrumentation. This effectively solves constraints numerically, discovering new program behavior and potential bugs.

Angora effectively discovers code coverage through source-level instrumentation, relying on high-level information from source code. Instrumentation is added around comparisons during compilation, which records the operands of these comparisons. Because the instrumentation is added before optimization, comparisons have not yet been optimized. Instrumentation of these comparisons is more difficult to achieve on binaries because the process of optimizing comparisons may cause them to be combined, replaced with equivalent logic that is harder to reason about, or removed entirely.

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) Software assurance on legacy systems or proprietary software for which no source is available and (2) adversarial exploit generation against third-party software. Our work explores the adversarial space of binary fuzzing, and seeks to solve the following challenge:

> *Challenge:* Apply the fuzzing techniques implemented by Angora to obfuscated and non-source-available binaries with minimal performance reduction.

Chapter 3

Binary Rewriting for Fuzzing

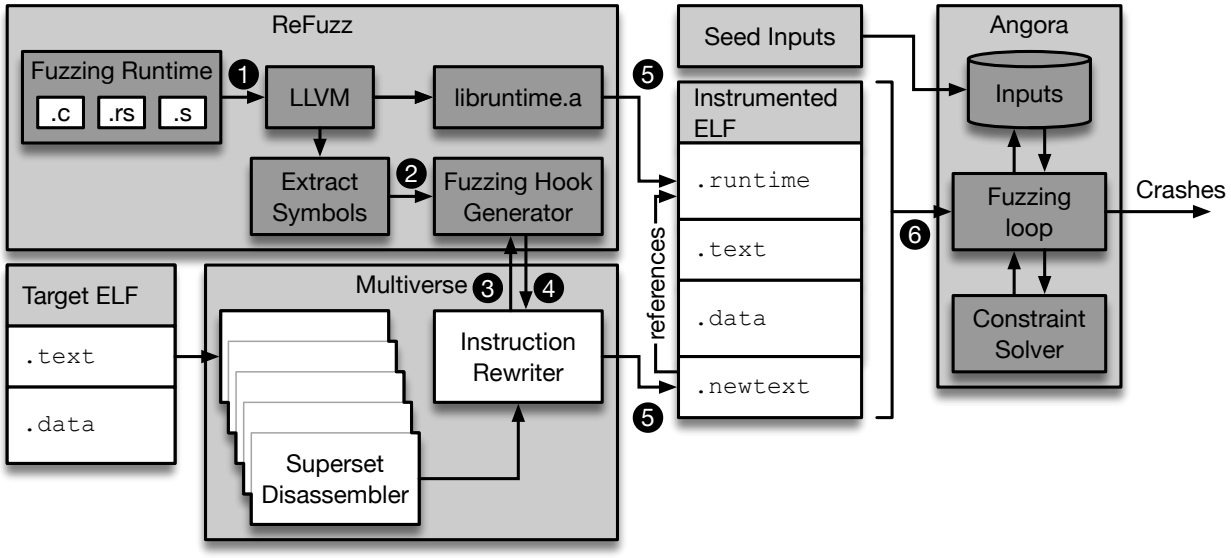


Figure 3.1: System for inserting fuzzing instrumentation using MULTIVERSE. REFUZZ functions as an instrumentation module with MULTIVERSE.

Fuzz testing starts with known good program inputs, and measures the program's response to those inputs. A fuzzer then uses those measurements to mutate the inputs, with the goal of discovering new program states and bugs. Central to this process is instrumenting the code to measure runtime behavior. Most fuzzers perform this instrumentation during compilation of source code. When source code is not available, there are three potential options:

1. *Don't instrument*. The program can be run without instrumentation, and the inputs can be

mutated randomly without regard to program behavior. This is referred to as black-box fuzzing, and can be performed on binaries without modification. AFL [29] supports this out of the box.

2. *Instrument dynamically*. Extant binary fuzzers employ this method, using a binary instrumentation framework such as Pin [23], QEMU [2], or DynInst [1]. 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 completely.

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

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 applicable with source code available.

Our tool, REFUZZ, integrates with MULTIVERSE to instrument stripped binaries for fuzzing, inserting comparable instrumentation to that of Angora's compile-time instrumentation. Figure 3.1 illustrates this integration, which consists of several steps:

❶ Portions of the runtime written in a high-level language such as C or Rust are compiled using LLVM [15]. This produces a statically-linked runtime library.

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

❸ MULTIVERSE processes each possible instruction in the target ELF file. These instructions are passed to REFUZZ as candidates for instrumentation. REFUZZ generates instrumentation assembly for control flow instructions and comparisons, and inserts calls to runtime functions that record these instructions.

❹ Instruction instrumentation is returned to MULTIVERSE, which reassembles a new text segment containing original instructions and inline 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 call complex functionality defined in `.runtime`.

❻ The final instrumented ELF is passed as input to Angora's fuzzing loop, which applies conditional optimization to expand code coverage and discover bugs.

We begin by describing our instrumentation technique more generally. This requires reliable rewriting of the target application without symbol information, and insertion of an instrumentation runtime along with rewritten code. There are also additional performance challenges that arise when

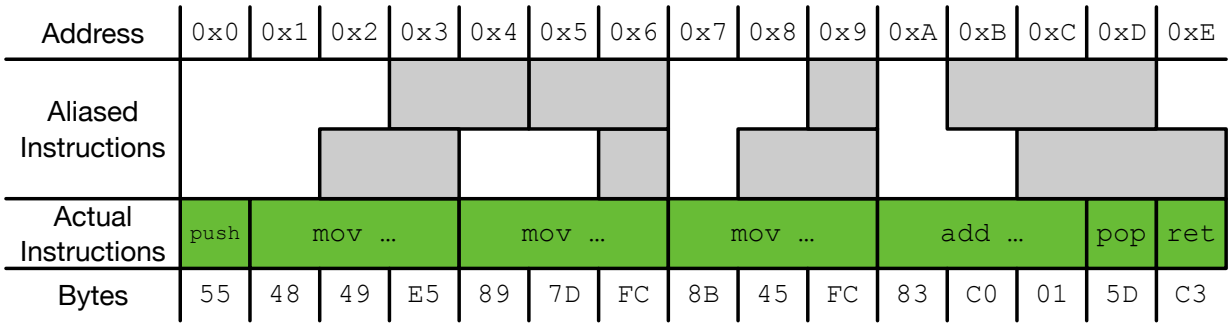
using a rewriter for fuzzing. We then discuss the challenges associated with implementing specific fuzzing analyses on top of our instrumentation. REFUZZ's ability to extract high-level semantics without source code available is a key strength over existing binary fuzzing techniques.

3.1 Challenges to Binary Rewriting

```
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 3.2: Binary code in x86 is unaligned, and original instructions are aliased by additional valid instructions at most byte offsets. Determining the actual instructions statically is undecidable [28].

Application binaries contain executable code and data. One of the greatest challenges to static binary rewriting 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 in a variety of ways with varying success. REFUZZ's choice of approaches is instrumental to its ability to reliably fuzz binaries.

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 [5]. Figure 3.2 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 all valid instructions based on function boundaries, and the UROBOROS authors use this analysis to extract valid instructions [26, 27]. The authors of UROBOROS note that their tool can become more effective as function boundary identification improves. However, even the most advanced function boundary extraction methods often fail to identify boundaries precisely under adversarial conditions such as obfuscation [6, 21]. 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 was, therefore, not viable.

In recent years binary analysis tooling has placed more emphasis on determining valid instruction offsets without relying on function boundary identification [24]. Although, as Wartell et al. note [28], fully sound static disassembly is undecidable in general, these techniques can correctly disassemble most programs, and as a result rewriters such as RAMBLR [25] have been developed that rely on this new instruction identification. RAMBLR makes fewer assumptions when determining instruction locations, and so works correctly on more programs, though it is not completely resistant to obfuscation and still relies on heuristics to determine instruction locations.

In 2018 an alternate approach was proposed by Bauman et al. [7]. They argue that, instead of relying on heuristics to determine what instructions should be disassembled, a rewriter can simply disassemble all valid instructions in the binary. This set of code 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. While binaries rewritten in this fashion have relatively high size overhead, they introduce acceptable execution time overhead while ensuring that all possible execution paths are instrumented. REFUZZ builds on MULTIVERSE, analyzing each instruction individually and inserting appropriate instrumentation.

3.2 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. To build an instrumentation framework for fuzzing, we selected an existing tool for binary rewriting, then developed additional tooling to link the fuzzing runtime into the resulting binary.

Based on the rewriting techniques implemented by MULTIVERSE, we developed a lightweight framework that allows for an instrumentation runtime to be inserted into a rewritten binary alongside the instrumented code. We also developed a linking technique that allows references from instrumentation hooks (the assembly snippets inserted inline with original code) to the runtime. To this end, 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, including Angora's fuzzing instrumentation library, which is written in Rust. Because it is not possible to reliably edit dynamic library information for the fuzzed program, this instrumentation may not rely on any dynamic libraries, so the 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 each instruction. This instrumentation takes the form of small assembly snippets, which are assembled 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.

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

3.2.1 Performance Considerations

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 (semantic strength) and how rapidly program executions can be performed (throughput). Whereas source-level instrumentation can be inserted using a compiler's Intermediate Representation to achieve high throughput on binaries, REFUZZ must modify compiled binaries directly. Statically inserted fuzzing instrumentation is inserted before optimization is performed, so the instrumentation can be optimized to the specific program. Binary instrumentation does not have this advantage, so there is an inherent overhead in instrumenting binaries.

One key advantage to instrumenting using an intermediate representation is the ability to efficiently use registers. Fuzzing tools that leverage LLVM bitcode can add abstract instructions that are then mapped to processor registers. In contrast, REFUZZ can make no assumptions about a program's use of registers. As such, each instrumentation stub inserted must save and restore the processor state. Storing the state of all registers presents a substantial performance overhead. To reduce the impact of this, REFUZZ uses calling conventions that reduce the number of registers used. This allows instrumentation stubs to be simple and fast.

3.3 Challenges to Static Fuzzing Instrumentation

Angora's instrumentation depends on precise analysis of program source code to implement strong fuzzing techniques. In addition to difficulties relating to binary instrumentation in general, Angora's use of source code presents additional challenges in achieving comparable analysis on binaries alone, especially when binaries are obfuscated and debug information has been removed. This section surveys the challenges faced in adapting fuzzing-specific analysis to stripped binaries.

```
1 int foo(int a) {
2     if (a > 20)
3         a *= a;
4     return a + 10;
5 }
```

```
0 83 ff 14     cmp    edi,0x14
  ❶
3 7e 03        jle    0x8
  ❷
5 0f af ff     imul   edi,edi
8 8d 47 0a     lea    eax,[rdi+0xa]
  ❸
b c3           ret
  ❹
```

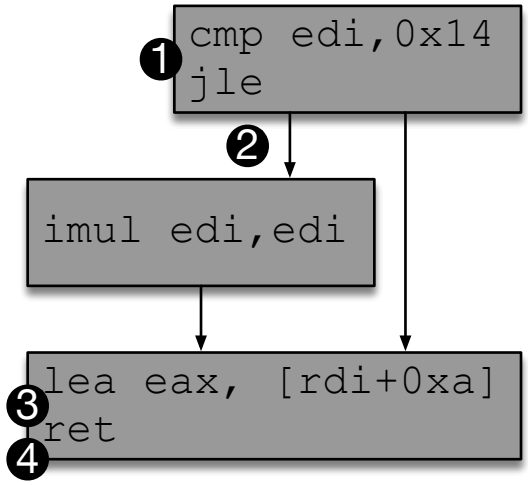


Figure 3.3: Disassembly and control flow of a simple function. Basic blocks representing lines 3 and 4 in the C source are not separated by any control flow instructions. Circled numbers represent points where REFUZZ inserts converage instrumentation.

3.3.1 Measuring Coverage

Coverage measurement is essential to modern fuzzers [4, 10, 29], whether implemented using static or dynamic instrumentation. In each of these cases, however, basic block coverage is relatively easy to directly measure or infer. When statically rewriting binaries for fuzzing, control flow information is not available. To overcome this limitation, we approximate basic block coverage by instrumenting for coverage at strategic points in the binary.

Because basic blocks in x86 assembly generally end with a control flow instruction, we achieve a strong approximation of basic block coverage by simply inserting a coverage point before each control flow instruction. This, however, does not successfully differentiate basic blocks that are adjacent in virtual address space, where one basic block executes after the previous one. This occurs frequently in code such as in Figure 3.3. In this example, basic blocks are not always separated by

control flow instructions. AFL [29] encounters a similar problem when instrumenting programs at the assembly-code level, though rudimentary information about basic blocks is still present in the form of comments produced by the compiler. In contrast, REFUZZ must achieve strong branch coverage while instrumenting each x86 instruction individually.

To approximate basic block coverage, REFUZZ makes an assumption that holds true for most well-behaved x86 programs: Basic blocks either begin or end with a control flow instruction. We leverage this assumption by inserting coverage instrumentation before and after each control flow instruction. In Figure 3.3, for example, instrumentation is inserted at ❶ and ❷, around the `jle` instruction, and at ❸ and ❹ around the `ret` instruction. In this example, there is now at least one instrumentation point in each basic block, allowing the fuzzing loop to accurately measure coverage.

Context sensitive branch count Recent work has shown [10] that tracking coverage at the basic block level alone is not sufficient to achieve maximal coverage. This is because functions called in one context may never be exploitable, but may be when called in another way (Shown in Figure 5.1c). Angora tracks the call context of conditionals and basic blocks to successfully exploit these branches. When source code is available, this can be trivially implemented by pushing and popping a unique ID to a stack before and after each call instruction. When statically instrumenting binaries, however, this is not possible. Functions must be called with their original return addresses kept intact, as return addresses may be used to access data in the original binary.

To bypass the above limitation, REFUZZ instruments call and return instructions to push and pop context. However, call instructions are not necessarily paired with instrumented return instructions. In calls to shared libraries, for example, the return is executed by the shared library and so is not instrumented. While this specific case can fixed by not instrumenting calls to the procedure lookup table, other edge cases cannot be easily detected during rewriting. As such, function context is inherently less accurate when instrumenting on binaries.

3.3.2 Inferring Comparisons

```
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 3.4: Disassembly of a `min` function. The control flow logic is split between two instructions, separated by another unrelated instruction.

The main strength of gradient-descent fuzzing comes from instrumenting comparisons, which

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 [10])

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

are then solved using optimization techniques. Information on comparisons is readily available from LLVM bitcode, so little additional analysis is needed to add instrumentation. In the binary domain, this is not the case. Comparisons are performed across at least two instructions in x86. First, a variant of the `cmp` instruction is called, which fills in the `FLAGS` register with all possible comparison results. Next, 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. This representation of comparisons in x86 binaries frustrates instrumentation efforts.

Not only are comparisons separated into multiple instructions, but the control flow between these instructions is not decidable during rewriting. In addition to `cmp` instructions, arithmetic instructions also set comparison results based on computations, along with many other instructions. Further, the first phase of a comparison (setting the `flags` register) need not occur in the instruction immediately preceding the second-phase instruction. In fact, comparison flags can be stored and loaded later, meaning that the two phases of a comparison can be arbitrarily separated at execution time. Figure 3.4 shows examples of comparisons in x86 assembly. Note that, since the control flow of a binary executable cannot be soundly determined, identifying these comparisons during rewriting is undecidable in general.

To solve this problem of instrumenting comparisons we take a similar approach to the processor itself. The compare and conditional instructions in x86 interact via the `FLAGS` register: compare instructions populate this register and conditional instructions read from it (Note that, in REFUZZ we consider compare instructions to be any that store information in `FLAGS` that can later be used by a conditional instruction). Likewise, we store information about comparisons when a compare instruction is called, then subsequently retrieve and instrument based on this information when a conditional instruction is called.

Comparisons in x86 can be modeled in two steps, as shown in Table 3.1. First, an instruction such as `cmp` computes a function f of the two compared expressions. The truth value of various constraints is then computed on this function, and stored in the `FLAGS` register as *status flags*, such as Overflow (`OF`), Carry (`CF`), and Zero (`ZF`). Our instrumentation of these comparison instructions consists of storing the value of f in a thread-local variable. When the condition instruction is executed, it performs an operation, such as jumping or writing registers, based on the stored flags. Whereas source-level gradient descent fuzzers infer their constraints from the source code, we infer constraints from these conditional instructions. At runtime, each time a conditional instruction is executed we log both the constraint (derived from the instruction opcode), and the value of f, which

was previously stored. The fuzzing loop can then use these log entries to perform gradient descent on any conditional instruction.

Comparison instructions in x86 can be separated into two types: (1) conventional comparison instructions, whose purpose is to populate the status flags for use in conditional instructions (e.g. `cmp` and `test`) and (2) implicit comparison instructions, which compute and store a function of arguments to an output, as well as setting the status flags (e.g. `sub` and `and`). Implicit comparison instructions can easily be instrumented by storing the value of the output register, then retrieving this value when a conditional instruction is executed. Conventional comparison instructions must be instrumented by calling the equivalent implicit comparison instruction, which stores the intermediate value. Each conventional comparison has an equivalent implicit comparison instruction that stores the resulting value as well as setting status flags.

3.3.3 Taint Tracking

Fuzzing using gradient descent requires isolating what portions of the program input influence each conditional expression. This is done using taint tracking, which measures this dynamically for each program execution. Taint tracking instrumentation can either be inserted during compilation, or at runtime using a dynamic instrumentation tool such as Pin [17]. In the case of rewriting for fuzzing, instrumentation during compilation is not available, so dynamic instrumentation must be used. REFUZZ leverages existing dynamic instrumentation in Angora's fuzzing loop, which is based on libdft [14].

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 only needs to be performed once, followed by many executions of coverage measurement when solving the constraints in the conditional instruction.

Chapter 4

Implementation

We implemented REFUZZ as a modified version of MULTIVERSE. We now describe the implementation challenges associated with inserting instrumentation at the instruction level.

Whereas source-level fuzzers such as Angora instrument using LLVM-IR before registers are allocated, REFUZZ must instrument on stripped binaries and so has limited access to control flow and register information. This complicates the instrumentation process. To account for this, instrumentation stubs inserted into the rewritten program must restore all register contents, including the x86 `FLAGS` register. The instrumentation code in Figure 4.1 is representative of two approaches to instrumenting functions. In 4.1a, no runtime function calls are needed, and so the instrumentation can be optimized to reduce register footprint. In 4.1b, a runtime call is needed and so execution state must be saved and restored.

```
1 mov [__cmp_arg1], rax
2 mov rax, rax
3 xchg rax, [__cmp_arg1]
4 mov [__cmp_arg2], rax
5 mov rax, qword ptr [rsp + 0x18]
6 xchg rax, [__cmp_arg2]
7 mov dword ptr [__cmp_size], 8
8 mov dword ptr [__cmp_stored], 1
```

(a) `cmp rax, qword ptr [rsp + 0x18]`: Arithmetic instructions are avoided so the `FLAGS` register is unaffected.

```
 1 lea rsp, [rsp-0x100]
 2 mov [rsp], r11
 3 mov [rsp+8], rax
 4 mov [rsp+16], rdi
 5 lahf
 6 seto  al
 7 mov edi, 0x4f65
 8 call __count_edge
 9 add al, 0x7f
10 sahf
11 mov rdi, [rsp+16]
12 mov rax, [rsp+8]
13 mov r11, [rsp]
14 lea rsp, [rsp+0x100]
```

(b) `jmp 0x40044a` : Calling an instrumentation runtime function requires saving registers and flags.

Figure 4.1: Example instrumentation stubs from instrumenting the `base64` executable. Instrumentation code must always maintain the execution state of the processor.

Chapter 5

Evaluation

REFUZZ aims to demonstrate strong fuzzing techniques on binaries with throughput competitive with source-level instrumentation. We evaluate this goal through the following questions: (1) can REFUZZ find the same classes of bugs targeted by source-level fuzzers, and (2) what are the performance trade-offs of binary fuzzing on code coverage and bugs found.

5.1 Fuzzing Instrumentation

We first evaluate REFUZZ on four artificial workloads (Shown in Figure 5.1) to 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 [29],

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	AFL
`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 does not consistently find the crash within 60s.

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`	22	44	44	0
`uniq`	24	14	19	0
`md5sum`	0[a]	56	22	0
`who`	69	259	8	0

[a] REFUZZ did not find any bugs in `md5sum` due to an instrumentation error.

LAF-INTEL [3], and Angora [10] on these four programs (Results in Table 5.1):

1. `simple` contains a trivial buffer-overrun bug. This was found quickly by all four tools. Notably, constraint solving as implemented by Angora and REFUZZ does not exploit this bug, and instead 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 dataset, though in this case the magic bytes are compared directly against the input. AFL cannot successfully exploit this bug in reasonable time, while the other three tools successfully find the bug.

3. `context` has a similar magic byte comparison to that of `magic`, but the bug is only exploitable within the first call to `foo`. LAF-INTEL cannot consistently trigger this bug. REFUZZ and Angora both implement context sensitive branch counts, and so both successfully explore 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 the bug to no longer be exploitable. 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.

Testing on the above programs demonstrates the effectiveness of fuzzing using binary rewriting. A program instrumented at the binary level can achieve comparable fuzzing techniques to extant source-level fuzzing tools, including gradient descent to solve conditionals and context sensitivity. Additionally, instrumenting at the binary level preserves the behavior of programs that might otherwise by modified by the instrumentation process (as is the case in `undef`). In this respect, fuzzing using binary rewriting can actually be more effective than through source-level instrumentation.

5.2 Performance on Fuzzing Corpora

We continue by comparing the performance of REFUZZ with other fuzzers on the LAVA-M bug corpus [11]. This corpus consists of four programs from GNU Coreutils (`base64`, `uniq`,

`md5sum`, and `who`) that have had memory corruption bugs artificially inserted. These bugs take the form of comparing some function of the input with a 32-bit number, and triggering a segmentation fault of the number matches exactly.

We compared REFUZZ with Angora[1], LAF-INTEL, and AFL in QEMU mode (QAFL), which instruments binaries dynamically for fuzzing. Of these, only QAFL and REFUZZ work on binaries without source code. The 4 fuzzers were run for 1 hour against each of the 4 programs over 7 trials, and discovered crashes were compared against the unmodified corpus executables to count the number of unique crashes over time.

As shown in Table 5.2 and Figure 5.2, performance of REFUZZ on the LAVA-M corpus was mixed. In cases where instrumentation completed successfully, REFUZZ found consistently more bugs than QAFL, the only other tool tested that functions without source code. REFUZZ finds fewer bugs than Angora on the dataset. We hypothesize that this may be an implementation weakness in REFUZZ rather than a theoretical one, as the bugs in the LAVA-M dataset are all similar in construction and REFUZZ exhausted all instrumented conditionals during the hour of testing. This implies that there may be additional conditionals in the programs that were not correctly instrumented.

[1] Angora was tested using Pintool-based taint tracking, not compile-time taint tracking. This was done to match the taint tracking used with REFUZZ, so the difference in found bugs is attributed solely to the static instrumentation.

```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 is only triggered in the first call to `foo`

```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

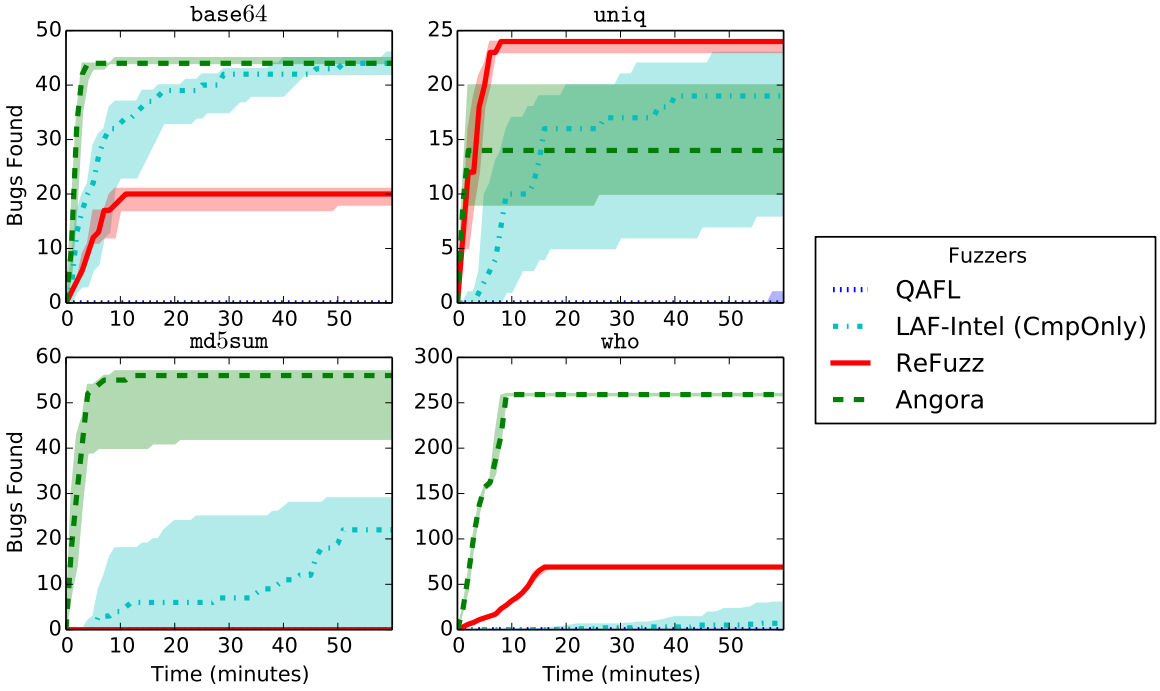


Figure 5.2: Number of bugs found by each fuzzer in the LAVA-M corpus over time. Shaded areas represent range across 7 trials, with lines being the median number of bugs found.

Chapter 6

Discussion and Future Work

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

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

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, allow for determining a subset of unused registers.

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 as large as $50\times$ the original size. 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 stubs. Because conventional linkers could not be used on rewritten programs these instrumentation stubs 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.

Chapter 7

Conclusions

In this work, we introduced a novel application of static binary rewriting to fuzzing. Our tool, REFUZZ, instruments programs for fuzzing without source code available, and implements comparable fuzzing techniques to those used by recently published works in the space. We evaluated REFUZZ on a set of example programs and a corpus of artificially generated bugs, showing that fuzzing using REFUZZ finds more bugs than other tested binary fuzzing techniques, though not as many as source-based instrumentation. We also found that, in some cases, fuzzing by statically instrumenting binaries finds bugs that would not be found via source instrumentation. We conclude that static binary instrumentation is a promising approach to fuzzing, though techniques could be improved by developing new tooling for static binary rewriting.

Bibliography

[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] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. Redqueen: Fuzzing with input-to-state correspondence.

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

[6] 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)*, pages 845–860, 2014.

[7] E. Bauman, Z. Lin, and K. W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proc. NDSS*, pages 40–47, 2018.

[8] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*. IEEE.

[9] D. Brumley, S. K. Cha, and T. Avgerinos. Automated exploit generation, Sept. 15 2015. US Patent 9,135,405.

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

[11] 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)*, pages 110–121. IEEE, 2016.

[12] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. Citeseer.

[13] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882. ACM, 2018.

[14] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Acm Sigplan Notices*, volume 47, pages 121–132. ACM, 2012.

[15] 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*, page 75. IEEE Computer Society, 2004.

[16] 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*, pages 627–637. ACM, 2017.

[17] 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*, volume 40, pages 190–200. ACM, 2005.

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

[19] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. 2017.

[20] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[21] 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*, pages 1–25. Springer, 2008.

[22] 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*, volume 16, pages 1–16, 2016.

[23] P. Thompson. https://github.com/mothran/aflpin. `https://github.com/mothran/aflpin`, 2015.

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

[25] 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.

[26] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[27] S. Wang, P. Wang, and D. Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 236–247. IEEE, 2016.

[28] 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*, pages 273–285. Springer, 2014.

[29] M. Zalewski. American fuzzy lop: a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl`, 2010.

Academic Vita – Eric Pauley

Research

Penn State University 2017 – Present

Programming Language Security

Build practical language security techniques, including probabilistic confidentiality and advanced fuzzing techniques, using LLVM. Apply techniques from statistical learning and optimization to precisely model and exploit program behavior.

Penn State University 2015 – 2016

Bioinformatics

Published novel DNA processing algorithm, showing a large increase in speed over extant solutions.

Work Experience

Sendtric LLC 2015 – Present

Co-founder – Email Countdowns

Started a company offering email countdown timers for marketers; created efficient platform that has served *billions* of emails. Continuing to develop service, which is the most popular of its kind with customers ranging from small businesses to Fortune 50. Managing business operations and facilitating client success.

Microsoft Summer 2018

Software Engineering Internship

Designed experimental system for pairing PCs and mobile devices, easing user onboarding while providing more accurate telemetry to developers.

Personal Project Winter 2017

High-Frequency Trading Software

Developed and deployed software for cryptocurrency market making and arbitrage.

IBM 2017 - 2018

Systems Automation Internship

Created a global cloud for automated testing on IBM Power machines. Architected platform including scaling and security. Helped lead team on new software, slated to manage global lab resources.

Kapost Summer 2016

Software Development Internship

Developed modular and testable backend used by enterprise customers, building applications on a team using Ruby on Rails and ReactJS. Practiced efficient use of version control and agile development.

Education

2018–2020 (concurrent) **MS – Computer Science**
NSF Graduate Research Fellow
Advisor: Patrick McDaniel
Pennsylvania State University

2015 – 2019 **BS – Electrical Engineering**
BS – Computer Science
Schreyer Scholar
Pennsylvania State University

Skills & Interests

Languages	Go, Rust, C++, C, LaTeX, Python, Ruby, Javascript, Java, Verilog
Skills	Systems Engineering, Software Security, Binary Analysis, Performance Analysis
Interests	LLVM, Language Security, Fuzzing, Program Analysis, Distributed Systems, Algorithmic Trading

Public Projects

2019 **Sendtric (sendtric.com)**
Email countdown service

2017 **Go-Quantize (git.io/goquant)**
Highly-optimized image quantizer for Go

2017 **FlowCache (git.io/flowcache)**
Collaborative cache filling for Go

Competitions

2017 **HackPSU**
1st place

2016 **ProfHacks (Rowan University)**
1st place, Best developer tool

2016 **HackRU (Rutgers University)**
Most innovative embedded technology

2016 **HackPrinceton**
Best use of cloud services

2015 **Microsoft College Code Competition**
1st place at Penn State