RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization

Sushant Dinesh
Purdue University

Nathan Burow
Purdue University

Dongyan Xu
Purdue University

Mathias Payer
EPFL*

***Abstract*—Analyzing the security of closed source binaries is currently impractical for end-users, or even developers who rely on third-party libraries. Such analysis relies on automatic vulnerability discovery techniques, most notably fuzzing with sanitizers enabled. The current state of the art for applying fuzzing or sanitization to binaries is dynamic binary translation, which has prohibitive performance overhead. The alternate technique, static binary rewriting, cannot fully recover symbolization information and hence has difficulty modifying binaries to track code coverage for fuzzing or to add security checks for sanitizers.**

The ideal solution for binary security analysis would be a static rewriter that can intelligently add the required instrumentation as if it were inserted at compile time. Such instrumentation requires an analysis to statically disambiguate between references and scalars, a problem known to be undecidable in the general case. We show that recovering this information is possible in practice for the most common class of software and libraries: *64-bit, position independent code*. Based on this observation, we develop `RetroWrite`, a binary-rewriting instrumentation to support American Fuzzy Lop (AFL) and Address Sanitizer (ASan), and show that it can achieve compiler-level performance while retaining precision. Binaries rewritten for coverage-guided fuzzing using `RetroWrite` are identical in performance to compiler-instrumented binaries and outperform the default QEMU-based instrumentation by *4.5x* while triggering more bugs. Our implementation of binary-only Address Sanitizer is *3x* faster than Valgrind's memcheck, the state-of-the-art binary-only memory checker, and detects 80% more bugs in our evaluation.

I. Introduction

Most software for commodity systems is closed source, and even developers for such systems frequently rely on closed source libraries. Even on Linux, widely used applications such as Skype, the Google Hangouts plugin, or video codecs are closed source. Consequently, users (and even developers) are at the mercy of third-parties to detect and patch the inevitable security issues. While mitigations such as ASLR [1], DEP [2], Stack Canaries [3], or CFI [4], [5] protect against exploitation, they cannot pinpoint the underlying vulnerability.

To discover memory errors during testing, best practices combine a feedback-guided fuzzer with sanitization. Both require information about the execution of the test cases. Fuzzers such as AFL [6] leverage coverage to guide exploration while tools such as Address Sanitizer (ASan) [7] check memory accesses for possible violations. These tools are implemented as compiler-passes to instrument the code during compilation, resulting in low runtime overhead. Existing approaches for binary software testing either: (i) resort to blackbox fuzzing [8], resulting in shallow coverage close to the provided test cases, (ii) rely on dynamic binary translation [9], [10] to instrument the binary at prohibitively high runtime cost (e.g., 10x to 100x for AFL fuzzing in QEMU mode on LAVA-M [11]), or (iii) use unsound static rewriting based on heuristics [12], [13].

The fundamental difficulty for static rewriting techniques is disambiguating reference and scalar constants, so that a program can be "reflowed", i.e., having its code and data pointers adjusted according to the inserted instrumentation and data section changes. During assembly, labels are translated into relative offsets or relocation entries. A static binary rewriter must recover all these offsets correctly. There are three fundamental techniques to rewrite binaries: (i) recompilation [14], which attempts to lift the code to an intermediate representation; (ii) trampolines [15], [16], which relies on indirection to insert new code segments without changing the size of basic blocks; and (iii) reassembleable assembly [12], [13], which creates an assembly file equivalent to what a compiler would emit, i.e., with relocation symbols for the linker to resolve. Lifting code to IR for recompilation requires correctly recovering type information from binaries, which remains an open problem. Trampolines may significantly increase code size, and the extra level of indirection increases performance overhead. Consequently, we believe that resymbolizing binaries for reassembleable assembly is one the most promising technique for static binary rewriting.

In this paper, we show that static binary rewriting, leveraging reassembleable assembly, can produce sound and efficient code for an important class of binaries: *64-bit position-independent code (PIC)*. Notably, such binaries include third party shared libraries, the analysis of which is the most pressing use-case for such a rewriter. Our rewriting technique, called `RetroWrite`, leverages relocation information which is required for position independent code, and produces assembly files that can be reassembled into binaries. To show the versatility of `RetroWrite`, we present case studies where we perform instrumentation for AFL and ASan on binaries.

We identify two key requirements for fuzzing binaries: (i) To maximize fuzzer throughput, we need a mechanism to generate instrumented binaries that are as performant as binaries instrumented at compile-time, and (ii) such rewriting should be sound (binaries still behave as expected) and scalable to support real-world use cases. Attempts to statically instrument binaries using DynInst [17] are not widely adopted as they do not satisfy the second criteria (in our evaluation `afl-dyninst` is ineffective in finding bugs despite its high throughput, we hypothesize that the instrumentation incorrectly tracks coverage). Our `afl-retrowrite` pass statically instruments binaries that are *just as performant as their compiler instrumented counterparts*. As `RetroWrite` is fundamentally sound in rewriting the binaries it supports, it can be widely adopted as a replacement to the current QEMU-based

* Mathias Payer is on unpaid leave from Purdue until 5/2020 to support his students based on his aquired funding. Purdue CS requires this note.

instrumentation when fuzzing position-independent code. Our evaluation of `afl-retrowrite` focuses on fuzzing throughput and effectiveness on seven real-world applications: `readelf`, `bzip2`, `file`, `bsdtar`, `pngfix`, `tiff2rgba`, and `tcpdump`, and on the LAVA-M benchmarks. `afl-retrowrite` has *4.2x and 5.6x higher fuzzing throughput* compared to QEMU-based AFL instrumentation on real-world benchmarks and LAVA-M respectively. On the LAVA-M benchmarks, `afl-retrowrite` finds more bugs than binary-based solution and is on par with compiler-based counterparts.

Fuzzers depend on program crashes to detect and report bugs. Consequently, bugs that do not trigger crashes are not caught through fuzzing. ASan is the most widely used memory checker that detects memory corruption bugs when they are executed (instead of silently corrupting some memory region). ASan is implemented as a compiler pass that adds tripwires to detect memory corruption. The availability of source information allows ASan to be far superior in terms of performance and bug detection rate when compared to existing binary-only solutions [18], [10]. Valgrind memcheck, the state-of-the-art binary-only memory checker, relies on dynamic binary translation (DBT) to instrument binaries at runtime. Valgrind's overhead (about 2x - 20x) due to DBT and heavyweight instrumentation makes it unsuitable for fuzzing. To the best of our knowledge, there are no lightweight alternatives to fuzz binaries with Valgrind memcheck. We develop `ASan-retrowrite`, a Binary Address Sanitizer as a `RetroWrite` instrumentation pass to retrofit binaries with memory checks. `ASan-retrowrite` is both lightweight and finds more bugs when compared to Valgrind memcheck. Additionally, `ASan-retrowrite` is compatible with ASan, thereby enabling blackbox components of software, e.g., closed-source or legacy libraries, to be rewritten by `ASan-retrowrite` while compiling the rest of the (source) code with ASan. Compared to Valgrind memcheck, our `ASan-retrowrite` has a *3x speedup* and is only *0.65x slower* than ASan (due to the lack of compiler optimization and register pressure) on SPEC CPU2006 C benchmarks.

Using `RetroWrite` passes for AFL and ASan instrumentation is significantly better than state-of-the-art tools for fuzzing binaries in terms of performance and compatibility, integrate with source-based tools, and viable as drop-in replacements. `RetroWrite` is available as open-source at https://github.com/HexHive/retrowrite. Our contributions are:

- A static binary rewriting framework that allows sound, efficient rewriting of 64-bit PIC binaries (§ III;
- An instrumentation pass] that allows binaries to be run with AFL with the same performance as compiler-based AFL instrumentation (§ V);
- An instrumentation pass] that retrofits binaries with ASAN checks, increasing by *3x* orders of magnitude the efficiency of memory safety analysis for binaries (§ IV);
- A comprehensive evaluation of ASan and AFL instrumentation (§ VI) on benchmarks and real-world applications, followed by a discussion of limitations (§ VII).

II. Background

Here we offer a summary of the building blocks of binary analysis and binary rewriting to ensure common background and terminology along with outstanding research questions addressed by `RetroWrite`.

a) Disassembly: Disassembly is the first step in binary rewriting, and is used to recover the existing instructions for analysis or modification. Disassembling a binary compiled for a variable length instruction set architecture is hard as the disassembler has to identify the length for each instruction. With an architecture as extensive as x86, nearly every sequence of bytes can be disassembled to some valid instruction. To counter this problem, the established strategies for disassembling binaries are linear sweep and recursive descent, which are discussed extensively by Schwarz et al., [19]. Linear sweep goes through the entire `.text` section top-down and eventually disassembles the entire binary while recursive descent follows the control flow of the program and disassembles all reachable code in the binary. IDA Pro [20] has been the industry standard for disassembling and reverse engineering, but there are other viable contenders such as radare2 [21], Binary Ninja [22], and static binary analysis frameworks such as angr [23]. All these tools use recursive descent to disassemble binaries.

Beyond instruction length, many ISAs intermix code and data, making it hard to distinguish between these sections. In general, deciding whether bytes represent code or data is undecidable [24]. However, as pointed out by Andriesse et al. [25], the undecidability is driven by corner cases and disassembling executables generated by mainstream compilers, e.g., gcc, clang, and Visual Studio, is possible with high accuracy (nearly 100%), even when compiled with high optimization.

b) Binary Rewriting: Techniques can be broadly classified into two categories based on *when* they instrument:

- **Dynamic Binary Translation (DBT).** DBT translates the binary while being executed. Consequently, they leverage runtime information and do not depend on complex static analysis that may not scale. This makes them an attractive solution to rewriting large software. Several off-the-shelf solutions for DBT exist, including Intel PIN [26], DynamoRIO [27], [28], QEMU [29], DynInst [15], libdetox [30], and Valgrind [10]. The most lightweight DBT techniques, i.e., Intel PIN, libdetox, and DynamoRIO, have anywhere between $\sim$10% to $\sim$20% rewriting overhead, i.e., with no instrumentation.
- **Static Binary Rewriting.** Static rewriting translates the binary *before* it is executed. Since the instrumentation is performed offline, the rewriter can utilize complex analysis and optimize the memory and runtime overhead, similar to compiler optimizations for source code. No off-the-shelf tool exists to rewrite *arbitrary* code. However, static rewriting is an active area of research with several research prototypes [14], [16], [31], [12], [13], [32]. Existing prototypes vary by runtime and memory overheads and the characteristics of rewritten binaries.

No existing rewriter meets our design criteria for a security oriented rewriter: DBT suffers from prohibitive runtime overhead and hence reduces efficacy of software testing practices, e.g., fuzzing. Though optimizations such as inlining can reduce the overall instrumentation overhead, DBT remains prohibitively expensive, and cannot compete with static rewriting techniques which optimize instrumentation offline. Static rewriting suffers from its reliance on static analyses, which adds both imprecision and complexity. Consequently, existing static techniques do not scale. A solution that combines the scalability of DBT with the low (runtime) overhead of static rewriting that remains sufficiently precise to support security instrumentation is highly desirable. We will show that reassembleable assembly is such a solution.

c) Reassembly: The key observation of reassembleable assembly is that assembly files produced by disassemblers have a *rigid* structure, i.e., code and data are pinned to their original locations and cannot be moved. Moving code or data breaks all references in the binary, which were hard-coded from labels to specific addresses by the assembler. In contrast, a compiler-generated assembly file maintains some of the source-level abstractions, such as variable names, in the form of assembler labels. These files usually do not have hard-coded addresses as those are assigned at link time.

Reassembleable assembly creates assembly files that appear to be compiler-generated, i.e., they do not contain hard-coded values but instead assembly labels. The core process of generating reassembleable assembly is thus *symbolization*: converting reference constants into assembler labels. Symbolizing the assembly allows security-oriented rewriters to directly modify binaries, much like editing compiler-generated assembly files. Once modified, the symbolized assembly files can be assembled using any off-the-shelf assembler to generate an instrumented binary.

Reassembly was first introduced in Uroboros [13]. Wang et al. developed a set of heuristics based on common compiler idioms to classify constants as reference or scalars and symbolize reference constants into assembler labels. ramblr [12] then advanced the state-of-the-art for reassembleable assembly by identifying several simplifying assumptions in Uroboros that led to non-functional binaries, and developed static analyses to improve symbolization accuracy. Despite this, ramblr acknowledges that their rewriting strategy is unsound and develop heuristics to abort the reassembly when rewriting correctness cannot be guaranteed.

We believe that reassembly is a promising rewriting technique for our requirements: instrumentation can be inlined thereby reducing the overall overhead, while still maintaining original program characteristics in terms of control flow, instruction selection, and register and memory access patterns. As an additional benefit, reassembly allows post-processing on symbolized assembly files. Consequently, using a security rewriter built on reassembleable assembly is inherently modular. Once the framework exists for producing the reassembleable assembly, security transformations can be added as modules to transform the assembly files before they are finally reassembled to produce the instrumented binary.

However, the main drawback of reassembly-based techniques is the requirement of completeness: no constant can be misclassified as a reference or a scalar. Without being complete, there is no guarantee that the reassembled binary will function correctly. However, it has been shown that statically determining whether a constant represents a scalar or a reference is infeasible [33]. Therefore, existing techniques are empirical and use heuristics to approximate a sound static analysis. While they work in many cases, they are generally insufficient to rewrite real-world binaries. For example, ramblr (representing state-of-the-art) reports false negatives in identifying references on coreutils built for x86-64. With larger, real-world applications, we expect a large number of misclassifications, which would prevent a binary from being rewritten correctly. Fortunately, while this restriction holds for the general case, there is hope for position-independent binaries.

d) Position-Independent Code (PIC): Executables compiled to be position-independent may be loaded at any virtual address by the loader. PIC is required both for ASLR and for shared libraries. Shared objects, such as libraries, are position-independent out of necessity: different processes may have different address space layouts and libraries need to be loaded at arbitrary addresses. Traditionally, executables are compiled to be loaded at a fixed address, because PIC introduces overhead by requiring offsets to be calculated at runtime rather than compile time. However, recent architectural features, e.g, the ability to reference relative to the instruction pointer (`rip`) on `x86_64`, have made this overhead negligible. Dynamic linkers are now smarter and have additional relocations to further reduce this overhead, making their performance identical to non-PIC while improving security.

All major Linux distributions such as Ubuntu, Fedora, and Gentoo [34], [35], [36], have switched to compiling and shipping binaries as PIC by default. In the smartphone ecosystem, Android has removed support for non-PIC linking and compiled binaries have to be PIC since Lollipop [37]. Though iOS does not forbid non-PIC binaries, it strongly encourages PIC and emits warnings for non-PIC binaries [38]. As PIC improves security guarantees with minimal performance impact, we anticipate PIC to be the de-facto standard on all platforms in the future. Therefore, we focus our efforts on developing principled techniques to rewrite PIC binaries by leveraging their relocation information for symbolization.

III. RETROWRITE

Binary rewriting allows for post-compilation modification of binaries. In particular, instructions can be added or deleted to enforce new security properties or remove unwanted functionality, while still maintaining an executable binary. Consequently, binary rewriting can be an incredibly powerful mechanism for increasing security by enabling, e.g., coverage-guided greybox fuzzing, and memory checkers with near source-based performance *on binaries*. However, rewriting binaries is not as straightforward as editing source code, mainly due to the

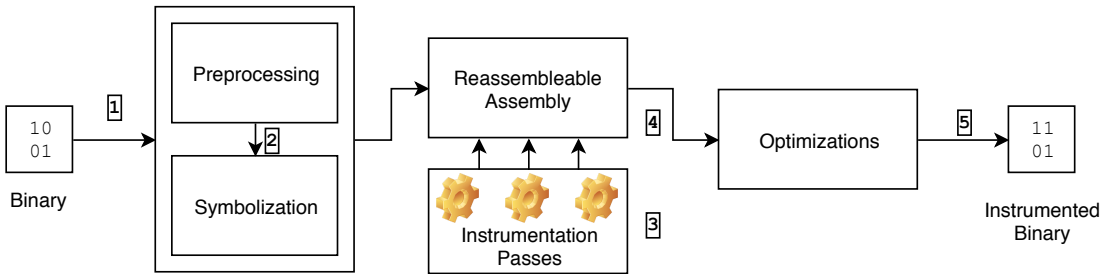

Figure 1: Overview of `RetroWrite`. Taking a binary as input, `RetroWrite` produces an instrumented version of the binary.

fact that binaries lack source-like abstractions. Binaries lack type information, and data-structure abstractions are flattened to raw memory accesses. Worse yet, the symbol information used to link the location of, e.g., functions and basic blocks to calls and jumps is lost. Adding or deleting code causes these addresses to change, breaking binaries. Using binary rewriting for security auditing therefore faces many reverse engineering challenges to recover sufficient information about the binary to enforce the desired security properties. The ideal rewriter, for security and fuzzing applications, should:

(R1) Have low runtime and memory overhead when the binary is recompiled with instrumentation (**Performance**).

(R2) Preserve the original program characteristics (barring changes made by the instrumentation). This, e.g., ensures that any discovered bugs directly translate to the original binary (**Correctness**).

(R3) Scale to real-world software (**Scalability**).

Existing DBT-based techniques do not satisfy our performance criteria (**R1**) while existing work on static binary rewriting does not satisfy at either the correctness (**R2**) or the scalability (**R3**) criteria, see § VIII for a detailed discussion of prior techniques.

A. *RetroWrite Design*

`RetroWrite` implements static rewriting through reassembleable assembly. The core operation to generate reassembleable assembly is *symbolization*, i.e., statically disambiguating between reference and scalar type for constants and replacing references with appropriate assembler labels. For PIC, we adopt a principled symbolization strategy without heuristics. `RetroWrite` leverages the relocation information in PIC binaries to reconstruct all labels that the compiler previously emitted before a binary was assembled.

`RetroWrite` is designed as a framework, with the reassembleable assembly-based rewriter at its core. Other modules can be added to the framework that instrument the generated assembly files to, e.g., track coverage for greybox fuzzers or add redzone for ASAN. Figure 1 shows an overview of the `RetroWrite` framework. `RetroWrite` takes as input a binary and produces an instrumented version, through a five step process. Specifically, `RetroWrite`'s steps are:

1) **Preprocessing.** The first step is to load sections of the binary required for reassembly, namely the text and the data sections. `RetroWrite` also loads auxiliary information, such as symbols and relocations from the binary. This step also includes disassembling using linear-sweep and recovering a best-effort control flow graph (CFG): identifying and adding edges for direct control-flow transfers. `RetroWrite` does not require heavyweight analyses to infer indirect control-flow targets, limiting analysis time and scaling to larger binaries.
2) **Symbolization.** Symbolization is the core of `RetroWrite`'s rewriting procedure. `RetroWrite` uses relocation information from the loading phase and the recovered control-flow graph to identify symbolizable constants, in both the data and code sections, and convert them to assembler labels. `RetroWrite` outputs reassembleable assembly at the end of this step. The reassembleable assembly may be further processed by other tools or instrumentation passes developed in `RetroWrite`.
3) **Instrumentation Passes.** Instrumentation passes operate on the reassembleable assembly to instrument and modify the target binary. The rewriting API is both flexible and expressive to perform heavyweight transformations on binary-code. We show the power of instrumentation passes in two case studies for AFL and ASan.
4) **Instrumentation Optimization.** `RetroWrite` analyzes the instrumentation to determine the required number of registers and side-effects. The results of these analyses are then used to: (i) selectively save (and restore) state changes, e.g., conditional flags, before (and after) every instrumentation site, and (ii) allocate registers for instrumentation.
5) **Reassembly (ASM).** As the final step, `RetroWrite` produces an instrumented assembler file. This may further be processed by other tools that operate on compiler generated assembly files before being assembled to a working binary using any off-the-shelf assemblers.

One advantage of our technique is that we do not need to lift assembly to a higher-level intermediate language, a process that requires precise modeling of the instruction set architecture (ISA). Capturing instruction semantics to lift from disassembly to an intermediate language is hard, must be implemented on a per-architecture basis, and is known to be error-prone. `RetroWrite` is lightweight and works directly on disassembly generated from any off-the-shelf disassembler.

Consequently, our design makes it straightforward to extend our rewriting framework to support multiple architectures.

Symbolization. Our symbolization procedures runs in three different phases, corresponding to symbolizing different types of references between code and data:

1) **Control Flow Symbolization**. Operands to control-flow instructions, i.e., calls and jumps, are converted to assembler labels, providing *code-to-code references*.
2) **PC-relative Addressing**. As position-independent code cannot reference fixed addresses, references are calculated relative to the program counter (on x86-64, this is `rip`). We adjust the operand of instructions that compute PC-relative addresses to use an assembler label instead. Then, at the location referenced by the instruction (calculated statically) the corresponding assembler label is defined. Such labels encompass both *code-to-code* and *code-to-data references*. This implicitly covers cases of indirect jumps and calls as function references are symbolized at the point where the address is taken, thereby concretizing the (until now imprecise) CFG.
3) **Data Relocations**. Lastly, we handle data references. In essence, we mimic the dynamic linker/loader in performing the relocations: at the offset pointed to by the relocation entry, we replace the bytes by an assembler label. The corresponding label is then defined at the address pointed to by the relocation (exact formula depends on the type of relocation). This process handles both *data-to-data* and *data-to-code references*.

Note that our approach to symbolization is fundamentally different from existing work: rather than using heuristics and analyses to categorize a constant as a scalar or a reference, we use relocations, PC-relative addressing, and recovered control flow to determine reference constants and symbolize them. Therefore, our approach is sound by construction and has zero false positives and false negatives. This means our approach is generic, and applicable to any real-world PIC binary.

B. Implementation

Our current implementation supports Linux x86-64 PIC binaries. It is implemented in about 2,000 lines of Python code and uses Capstone (a disassembly framework with support for multiple ISAs) to disassemble raw bytes into x86-64 instructions. `RetroWrite` uses pyelftools, an ELF parsing library to load ELF files and parse relocation information. The disassembly and relocation information are given to our symbolization analysis, which then synthesizes reassembleable assembly. Though our current implementation is restricted to the x86-64 architecture, other architectures supported by Capstone can be added with small engineering effort.

Generating reassembleable assembly is the first step towards rewriting binaries. Once reassembleable assembly is generated, writing instrumentation passes to safely instrument binaries at low-overhead requires three things: (i) a logical abstraction for analysis and instrumentation passes to operate on, e.g., modules, functions, or basic block level granularity, (ii) working around the ABI to ensure the instrumentation does not break the binary, and (iii) automatic register allocation to achieve compiler-like overhead.

a) Function Identification: Our reassembly step does not strictly need function start and size information. However, the rewriter and the instrumentation API greatly benefit from function information as it provides a natural way to structure analysis and instrumentation at function granularity. Our implementation uses the symbol table to identify function start and sizes. To support stripped binaries, users may reuse function identification as provided in commercial tools such as IDA Pro, or open source frameworks such as radare2, as a part of a pre-processing step. An alternative is to use other existing research in function identification [39], [40], [41], [42]. Note that the symbol information, when available, is strictly provided as a convenience to the instrumentation API, and not required by the symbolization step. Additionally, libraries contain exported symbol information even when stripped.

b) ABI Dependencies: The instrumentation API must also be aware of the ABI limitations to ensure the binary is instrumented as intended. For example, the System V ABI for x86-64 (the default ABI on Linux) specifies that leaf functions (functions that do not call other functions) may use 128-bytes below the stack-pointer as an implicit stack, without allocating it explicitly. This means that pushing or popping from the stack to save state before and after instrumentation would compromise stack-local variables, resulting in incorrect execution. To work around this limitation, `RetroWrite` discovers leaf functions through a static analysis. The instrumentation API is aware of the ABI and maintains a separate stack to save and restore state when instrumenting leaf functions. Other ABI dependencies include the calling convention, which influences the register allocation analysis as arguments may be passed in registers, and registers are used for the return value.

c) Register Allocation: Unlike a compiler-based instrumentation, binary-only tools do not have the luxury of relying on virtual registers and allowing the compiler to assign physical ones, but must choose their own physical registers. The instrumentation cannot clobber any program state, i.e., registers, conditional flags, program stack, or global state, as this can have unintended side-effects, leading to crashes or inconsistencies that are hard to debug. Therefore, the safest option is to save all state before entering instrumented code, and restore the saved state before exiting. However, this is prohibitively expensive hence undesirable.

To reduce overhead from saving program state, `RetroWrite` performs a conservative (over-approximate) intra-function liveness analysis to find all registers (and flags) that are live at instrumentation sites. In short, our liveness analysis is equivalent to a compiler-based liveness analysis with variables replaced by registers. As the analysis relies on the control-flow graph, which is incomplete (imprecise), the analysis has to over-approximate the set of live registers: we can tolerate false positives (register belongs to live set according to the analysis, but is not actually live) but not false negatives. False positives translate to fewer registers available for allocation and hence greater number of register

spills for instrumentation while false negatives lead to clobbering a register in use and consequently errors during execution. To reduce overall overhead, the instrumentation API ensures that non-live registers are allocated first before allocating live registers. Any live register that is allocated is also automatically saved before and restored after the instrumentation. Similarly, if the instrumentation clobbers conditional flags, these are detected, saved, and restored automatically as a part of the API.

IV. Binary Address Sanitizer

To demonstrate the potential security benefits of static binary rewriting, we first develop a memory checker plugin for the `RetroWrite` framework. Software written in low-level languages, i.e., languages without memory-safety guarantees, such as C and C++, is susceptible to memory corruption bugs. These bugs are the root cause of 70% of vulnerabilities in Microsoft code [43], and are frequently exploited by attackers to gain arbitrary code execution capabilities. Efforts to enforce full runtime memory safety (spatial and temporal) have been prohibitively expensive [44], [45] as they require every memory access to be checked in addition to tracking allocation information for every memory object. An alternate approach is to catch these bugs during software testing. However, these bugs may be subtle and not detectable through a crash. Even if the software crashes, isolating the root cause is usually non-trivial. Memory checkers [10], [18], [7] are a class of software testing tools that detect memory corruption bugs. They also provide a detailed backtrace of execution prior to the crash, making bug isolation and patching easier.

Valgrind is a popular dynamic binary instrumentation (DBI) framework that provides a state-of-the-art memory checker, appropriately named "memcheck". Valgrind provides an efficient mechanism for dynamic analyses to associate and track metadata to every register and memory value, i.e., shadow values. Memcheck uses Valgrind's shadow value capability to detect accesses to undefined memory locations, e.g., either uninitialized variables or out-of-bound accesses for buffers, by tracking undefined bit values. However, this is expensive, incurring anywhere from 2x to 300x overhead, making memcheck infeasible for use with frequent execution during testing, and in particular fuzzing, where higher throughput directly correlates to higher bug-finding probability.

A. Address Sanitizer Semantics

Address Sanitizer (ASan) [7] is a tripwire-based approach to detect memory corruption. In short, ASan modifies the memory allocation of an application to surround every memory object with a redzone, a forbidden region of memory whose access triggers a crash. Then, ASan instruments every memory access to check if it is an access to an allowed address, i.e., not a redzone. ASan provides probabilistic guarantees in detecting spatial and temporal memory safety violations: it increases the probability that a memory corruption triggers a crash close to the location of the bug, but is not guaranteed to detect every instance of memory corruption. ASan is implemented as a compiler pass in `gcc` and `clang`.

ASan utilizes shadow memory to keep track of allocated bytes of memory. As tracking allocation status for every single byte of memory in an application is prohibitively expensive, ASan maps eight bytes of memory to a single byte of shadow memory. The shadow byte value represents the state of memory. If an access to a memory location is forbidden, e.g., because it is deallocated or a redzone, then the corresponding shadow byte is *poisoned*, i.e., stores the value `0xff`. ASan uses two kinds of instrumentation: (i) *ASan allocation* to pad all allocations with redzones, and (ii) *ASan memory check* to terminate if an access is illegal.

B. Design

Our goal is to implement a binary version of ASan in `RetroWrite` that closely resembles and integrates seamlessly with the source-based sanitizer. `ASan-retrowrite` works with source-based solutions to instrument parts of an application where source code is unavailable, e.g., when linking against closed-source or legacy libraries, while applying existing compiler-based instrumentation where source code is available. The main difficulty in porting source ASan directly to binaries, even with a rewriting framework, is the lack of abstractions: binaries do not have any information about variables, types, or buffer bounds as these are stripped away during compilation. Recovering some of this information is possible through static analysis. But these analyses are expensive and impact scalability of systems that build on them. As our focus is on building a practical binary equivalent of ASan to aid the fuzzer in finding bugs, we trade some precision to scale to real-world software (see Table I for a policy comparison).

a) Stack: `ASan-retrowrite` instruments stack-objects at a stack frame granularity. Therefore, our instrumentation may miss bugs when the overflow is contained within the stack frame. This limitation is common to all binary-only tools as they do not have access to variable scope and type information. Several techniques have been proposed in the context of decompilation to recover this information [46], [47], [48], [49]. However, they are over-approximate which leads to false positives, limiting their usefulness in a fuzzing scenario. Adding redzones at stack frame granularity allows us to catch overflows on the stack without introducing false positives.

As an optimization, `ASan-retrowrite` does not redzone stack frames for every function. During compilation, the compiler performs a conservative analysis to identify functions that

Table I: Overview of redzone policy as implemented in ASan and `ASan-retrowrite`. `ASan-retrowrite` is equivalent to ASan on the heap, instruments at a stack frame granularity on the stack, and does not redzone global memory objects.

Memory Object	ASan	`ASan-retrowrite`
Heap	Individual	Individual
Stack	Individual; 32-byte (default)	Stack-frame; 8-byte
Global	Individual; Padded to 64-byte	No redzone

```
// Foo is a global buffer
int i = 0;
while(i < 32) {
  // Access to global Foo
  Foo[i] = <expr>;
  i += 1;
}
```

(a)

```
1 lea 0x40000(%rip), %rbx
2 lea 0x40100(%rip), %r15
3 .loop:
4 # loop body
5 addq 0x10, %rbx
6 cmp %rbx, %r15
7 jle .loop
```

(b)

```
# .Foo allocated @ 0x40000
# Compiler-ASM
lea .Foo(%rip), %rbx
lea .Foo+0x100(%rip), %r15
# Reassembly
lea .LC40000(%rip), %rbx
lea .LC40100(%rip), %r15
```

(c)

Figure 2: Code snippets to illustrate difficulty in modifying global data section. (a) Source code (simplified) provided for clarity, (b) Shows disassembly when the binary is compiled with optimization (`-O2`), (c) Compares a compiler generated assembly file which has the correct semantic connection between the two labels, while the reassembly misses this connection, and treats them as two independent labels. Making this semantic connection is in general undecidable, but a requirement for modifying the layout of global data.

have a potential stack-based buffer overflow, and selectively adds stack-canaries to protect the saved return address. We leverage this observation to our advantage and redzone only stack frames that have such canaries; as other functions are proved to be free from stack-based overflows at compile-time. Additionally, rather than enlarging the stack frame by adding new bytes for the redzone, we reuse the slot occupied by the canary and poison the corresponding byte in shadow memory to disallow access to it. This is equivalent to adding additional bytes in terms of detection capabilities while not incurring the memory overhead.

Lastly, as the stack frame is implicitly freed on function return and may be reused by the next function call, we identify every function exit, and unpoison the redzone before exiting from the function. Mechanisms that unwind the stack frame, such as `longjmp`, require us to unpoison all the stack frames that are unwound. To handle this case, we add additional instrumentation to iteratively unpoison every byte from the current stack top to the saved stack pointer (saved during the corresponding call to `setjmp`).

b) Global: `ASan-retrowrite` does not redzone globals. Symbolizing the disassembly is insufficient to perform arbitrary transformations on data section layouts and requires recovery of semantic information lost during compilation. To illustrate this problem, Figure 2 shows a snippet of disassembly alongside the source code, compiler generated assembly, and the reassembly. The access to the global buffer `Foo` is converted into an access through a pointer in the compiled code, where `%rbx` is the iterator and `%r15` is the bounds for the loop. As these addresses are symbolized to assembler labels independently, i.e., without understanding the semantic connection between labels, the generated reassembly has two independent labels that point to the beginning and end of `Foo` respectively. This is the cause of the problem: if we add bytes below `Foo` for the purposes of a redzone, the above label will no longer point to the semantic end of object `Foo`, and therefore the loop bounds will be incorrect. In general, if there is an object below `Foo`, we cannot know if the instruction references the beginning of the next object or the end of `Foo`. This is a semantic difference that the reassembleable assembly fails to capture. More generally, for every two adjacent globals, there are two labels, separating them. One for the first object's end, the other for the second object's beginning. These two labels are collapsed into a single indistinguishable offset.

Unfortunately, this is a common code pattern. Any binary rewriting tool that tries to modify the data layout must disambiguate the semantic meaning of such references. We could design an analysis to track pointer capabilities (track base pointers for every derived pointer), and propagate this information at every pointer operation involving two reference operands, e.g., subtraction to find length or comparison to check for bounds. This would allow us to semantically disambiguate the meaning of a reference use, i.e., is the address used to refer to the start of an object or is it used to denote the end of the previous object, by checking the pointer base. However, to precisely track pointer capabilities statically on a language that allows arbitrary pointers (such as assembly) we need precise alias information, which is undecidable to compute statically [50]. Alternatively, this information can be recovered through heuristics, but would hurt `ASan-retrowrite`'s soundness and introduce false positives. Therefore, we leave such efforts to future work. While we acknowledge this limitation, note that the number of global objects in an application is fixed and relatively small when compared to the number of allocations on the heap or stack. Therefore, compared to ASan, `ASan-retrowrite` may miss a fixed number of overflows between global objects. Other binary-only tools, such as Valgrind memcheck, also suffer from the same limitations.

C. Binary Address Sanitizer Implementation

`ASan-retrowrite` is implemented on top of `RetroWrite`. We use the disassembly from `RetroWrite` to identify all memory accesses and instrument them with memcheck instructions. The memcheck instructions themselves are written in assembly with actual registers replaced by symbolic register names. This allows us to leverage the register allocation capabilities of `RetroWrite` to reduce the overhead. We identify functions that use stack-canaries, and instrument the stack frames of such functions with redzones as described earlier. For each redzoned function, we identify all exits (including `longjmp`), and unpoison the stack before the exit. ASan initialization and de-initialization functions are registered as new entries in `.init_array` and `.fini_array` respectively. Finally, `RetroWrite` emits the instrumented reassembly file, which is then compiled and

linked against the ASan runtime library, `libasan.so`, to produce the `ASan-retrowrite` instrumented executable. Note that this allows interaction with other code that may already have been instrumented with ASan. Our experience in implementing `ASan-retrowrite` in `RetroWrite` suggests that other source-based sanitizers can be ported to support binaries with minimal engineering effort.

V. AFL Coverage Instrumentation

We now present an instrumentation pass for binary fuzzing. At its core, fuzzing is a form of random software testing where an application is run with random (potentially malformed) inputs while monitoring the runtime for unexpected behaviors, e.g., crashes, memory exhaustion, or infinite loops. However, blackbox fuzzing, i.e., fuzzing with no knowledge about the target application, may not be effective in most cases as a majority of inputs are likely to explore very shallow code paths. This severely limits a fuzzer's ability to uncover bugs in deep parts of code. *Coverage-guided fuzzing* tackles this problem by using program traces generated by the inputs as a feedback mechanism to tailor future inputs to the fuzz target.

a) AFL: AFL is one of the most popular and effective fuzzers in research and practice. To mitigate the cost of collecting and analyzing full program traces, AFL takes a more practical approach by tracking edge-coverage as an approximation of a program trace. AFL maintains a constant-size bitmap in shared memory to keep track of edge hit statistics during a run of the application. At compile time, every basic block start in the CFG is instrumented to collect edge coverage statistics. Statically, each basic block is assigned a key and the bitmap index I_e for an edge e is computed dynamically as: $I_e = cur \oplus (prev >> 1)$, where cur and $prev$ correspond to keys of the current and predecessor basic blocks respectively. AFL instruments applications during compilation, either through `afl-gcc` or through the more optimized llvm-mode, `afl-clang-fast`.

For uninstrumented binaries, AFL resorts to QEMU to instrument binaries dynamically at runtime. However, this has a significant overhead ($\sim$10x) compared to the source-based solution, reducing the fuzzer's throughput. Another significant drawback of using QEMU is its inability to support sanitizers, such as ASan, severely limiting AFL's bug detection capabilities. Recent advances in fuzzing, such as QSYM [51] or Angora [52], extend AFL. If they target binary-only code, these AFL extensions further extend AFL-qemu or angr-afl. Therefore, improving binary AFL-based fuzzing throughput immediately improves the effectiveness of these newer fuzzing techniques.

b) Binary AFL: Implementing binary-level coverage instrumentation on `RetroWrite` requires CFG recovery and instrumenting basic blocks starts with coverage instrumentation: calculating edge index and updating bitmap state. The CFG is implicitly recovered as a part of `RetroWrite`'s symbolization procedure. However, the original AFL implementation instruments the application at the assembly level, i.e., `afl-gcc` (a utility packaged with AFL) parses and instruments assembly files during compilation to generate an instrumented application. Since the assembly files generated by `RetroWrite` closely resemble those generated by a compiler, `afl-gcc` works out-of-the-box to generate AFL instrumented binaries, with no additional effort. This highlights a powerful feature of `RetroWrite`: the reassembly it produces is compatible with existing tools that operate on assembly, readily extending the tools' capabilities to support binary-only applications.

VI. Evaluation

Our evaluation is guided by the following research questions that directly support our earlier claims:

RQ1: Does `RetroWrite` scale to large real binaries?

RQ2: Have we significantly improved state-of-the-art binary-only memory checkers in terms of: (a) *runtime overhead*, and (b) *coverage*, i.e., bug detection rate?

RQ3: Are we competitive with source-based memory corruption detectors, such as Address Sanitizer, in terms of: (a) *runtime overhead*, and (b) *coverage*, i.e., bug detection rate?

RQ4: How does our coverage instrumentation compare to source-based AFL instrumentation? Is our solution a *viable alternative* to using QEMU-based instrumentation of AFL?

To validate our earlier claims and answer our research questions, we perform the following experiments:

1) Rewrite 12 real-world binaries that are *2.7x* larger than those tested for scalability by prior art **(RQ1)**;
2) Performance evaluation of `ASan-retrowrite` on SPEC CPU2006 comparing: baseline benchmarks (no instrumentation), ASan, `ASan-retrowrite`, and Valgrind memcheck (the most popular off-the-shelf binary-only memory checker) **(RQ2.a, RQ3.a)**;
3) Comparative security evaluation of the above targets on the Juliet testsuite on CWEs related to memory corruption **(RQ2.b, RQ3.b)**; and
4) Evaluation of `RetroWrite` for coverage guided fuzzing with AFL, comparing source-based AFL instrumentation, binary-only AFL-instrumentation (our implementation), and QEMU mode for AFL. We compare: (i) Fuzzer throughput, and (ii) their effectiveness in finding bugs in LAVA-M testsuite **(RQ4)**.

Hardware and Environment All our evaluations were performed on a desktop equipped with `Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz` processor and `32 GiB` of memory running `Ubuntu 18.04`. All the results and evaluation presented in this paper can reproduced using the docker image and instructions in the open-source repository[1].

A. Scalability

Table II shows a list of all successfully rewritten binaries and their sizes as part of the evaluation, demonstrating the scalability of `RetroWrite` (**RQ1**). Prior work on reassembleable assembly uses coreutils to evaluate scalability to real world software, which have a median size of 394KB and a

[1]Docker Image: https://github.com/HexHive/retrowrite/tree/master/docker.

Binary	Test Suite	Size
bzip2	SPEC CPU2006	256K
gcc	SPEC CPU2006	12M
gobmk	SPEC CPU2006	7.0M
h264ref	SPEC CPU2006	1.9M
hmmer	SPEC CPU2006	1.2M
lbm	SPEC CPU2006	48K
libquantum	SPEC CPU2006	176K
mcf	SPEC CPU2006	76K
milc	SPEC CPU2006	532K
perlbench	SPEC CPU2006	4.0M
sjeng	SPEC CPU2006	424K
sphinx_livepretend	SPEC CPU2006	804K
base64	LAVA-M	64K
md5sum	LAVA-M	76K
uniq	LAVA-M	64K
who	LAVA-M	576K
Juliet-CWE121	Juliet	12M
Juliet-CWE122	Juliet	7.6M
Juliet-CWE124	Juliet	3.5M
Juliet-CWE126	Juliet	2.7M
Juliet-CWE127	Juliet	3.4M
readelf	Real-world	2.2M
bzip2	Real-world	312K
file	Real-world	668K
bsdtar	Real-world	3.5M
pngfix	Real-world	984K
tiff2rgba	Real-world	1.4M
tcpdump	Real-world	5.0M

Table II: Overview of binaries rewritten by `RetroWrite`.

maximum size of 1.3MB. As Table II shows, we evaluate on 12 binaries that are larger than a megabyte, with a median size of 1.09MB (*2.7x* larger) and maximum size of 12MB (*9.2x* larger). Consequently, we are confident that `RetroWrite` can rewrite arbitrary C binaries.

B. Memory Checker — Performance

We evaluate the performance of `ASan-retrowrite` on the SPEC CPU2006 C benchmarks. Since the original benchmarks exhibit some memory safety violations, we applied patches provided in Google's Address Sanitizer repository [53]. The Google patch blacklists instrumentation of certain functions in perlbench, e.g., `char *move_no_asan`, as they cause violations. Our initial evaluation of `ASan-retrowrite` reported a use-after-free in the above function, after which we manually removed ASan checks from the same function for evaluation. All code was compiled with options: `-O2 -std=gnu89` and `gcc-5.4.0` and flags to produce position-independent executables. Valgrind was configured to track the same set of features as ASan.

Our evaluation indicates that on an average we are about 300% better than Valgrind, and 65% slower than ASan. We present a detailed view of our results in Figure 3. The main reason for ASan's low-overhead when compared to other memory checkers is its highly optimized memory check instrumentation. Therefore, any additional overhead is clearly visible in long-running benchmarks. We identified the following as causes for the overhead of `ASan-retrowrite` when compared to ASan:

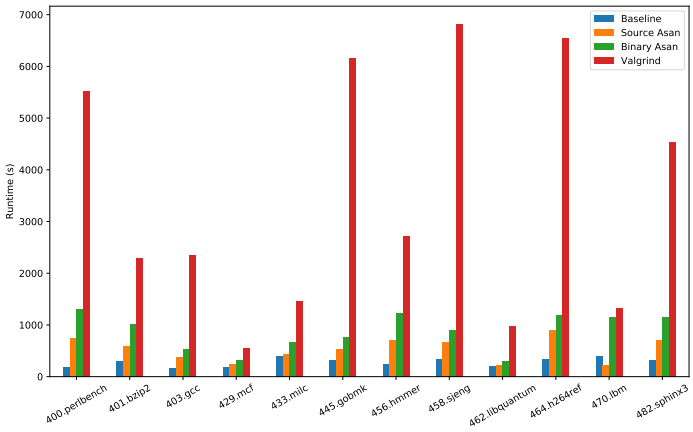

Figure 3: Evaluation on SPEC CPU2006 C Benchmarks. Mean Runtime (in s) for baseline (no instrumentation), ASan, `ASan-retrowrite` (our implementation), and Valgrind memcheck (state-of-the-art binary-only memory checker). Lower numbers indicate better performance.

1) **Instrumentation Locations.** `ASan-retrowrite` instruments more locations than source ASan. The compiler-based instrumentation removes some checks if it can prove accesses are safe.
2) **Register Spills.** As register allocation happens late during code generation, the compiler considers both the original code and the instrumentation, thereby generating better register allocation with fewer spills. `ASan-retrowrite` is limited to a conservative register liveness analysis and opportunistically uses dead registers to reduce register spills, but cannot change the register allocation scheme as a whole.
3) **Optimized Checks Placement.** To reduce register pressure, or flag recomputations, source-based ASan is free to move the memory check instrumentation (according to language semantics). However, our binary ASan cannot do this as hoisting checks is not always safe and needs more principled compiler-like analysis.
4) **Loop Hoisting.** Checking contiguous memory accesses in hot loops can be expensive. As a part of the optimization pipeline, a compiler may choose to hoist such checks out of the loop and perform a single check to reduce the overall overhead. Though possible, implementing such loop-hoisting mechanisms are a lot harder, mainly due to lack of abstractions such as loops in the binary level.

C. Memory Checker — Coverage

We compare our implementation of `ASan-retrowrite` against ASan and Valgrind on the Juliet test suite, a collection of test cases containing common vulnerabilities. Each test case has two variants: a *good* variant that does not contain a vulnerability and a *bad* variant that does. Tools are evaluated based on their capability to report errors on the bad cases while not flagging errors on the good ones. A *false positive* is when a tool reports a vulnerability on a good case. A *false*

Table III: Overview of Bug Detection Rate on Juliet on CWEs related to memory corruption. False postive is when a system reports a bug in a testcase with no bug. False negative is when a system reports no bug in a testcase with a bug. Timeout is when the testcase fails to terminate in 3 seconds.

	ASan	BASan	Valgrind Memcheck
Total	11,828	11,828	11,828
True Positive	4,489	3,257	1,785
True Negative	5,914	5,912	5,914
False Positive	0	0	0
False Negative	1,382	2,614	4,086
Timeout Vuln	43	43	43
Timeout Safe	0	0	0

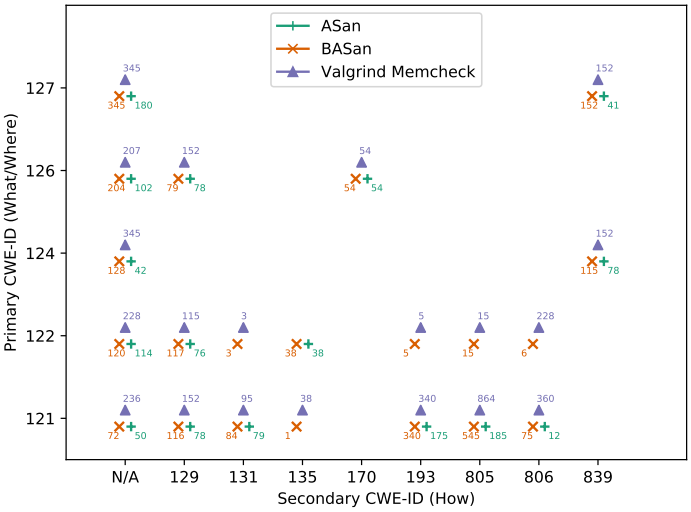

Figure 4: Clusters comparing false negatives on Juliet for the three systems: ASan, `ASan-retrowrite`, and Valgrind memcheck. X and Y axes are categorical and represent CWEs. Numbers next to points denote the number of FN of system that belong the cluster. Explanation of CWEs are provided in Table IV

Table IV: CWE Descriptions

CWE-ID	Description
CWE-121	Stack-based Buffer Overflow
CWE-122	Heap-based Buffer Overflow
CWE-124	Buffer Underwrite ('Buffer Underflow')
CWE-126	Buffer Over-read
CWE-127	Buffer Under-read
CWE-129	Improper Validation of Array Index
CWE-131	Incorrect Calculation of Buffer Size
CWE-135	Incorrect Calculation of Multi-Byte String Length
CWE-170	Improper Null Termination
CWE-193	Off-by-one Error
CWE-805	Buffer Access with Incorrect Length Value
CWE-806	Buffer Access Using Size of Source Buffer
CWE-839	Numeric Range Comparison Without Minimum Check

negative is when a tool misses an error on the bad case. Test cases are organized based on Common Weakness Enumeration (CWE), an ID that indicates the kind of vulnerability that the test case represents. We selected CWEs that represent memory corruption bugs, namely CWE121, CWE122, CWE124, CWE126, and CWE127, and compiled them with source ASan, `ASan-retrowrite`, and without any instrumentation for Valgrind memcheck. We run Valgrind memcheck with the same parameters as we did for the performance evaluation, i.e., disabling leak and uninitialized checks.

An overview of the accuracy of the three systems is shown in Table III. All three systems perform equally well identifying true negatives and have 0 false positives, i.e., they do not report errors on any of the safe variants in the Juliet test suite. In terms of detecting vulnerable cases, ASan has the highest detection rate at 4,489/5,914, followed by `ASan-retrowrite` with 3,257/5,914, and finally Valgrind memcheck with 1,785/5,914. This reflects the trade-off made in adapting source ASan to a binary-only `ASan-retrowrite`. However, `ASan-retrowrite` is far more effective than Valgrind memcheck which is the state-of-the-art binary-only memory checker, making it a viable alternative to Valgrind memcheck for binary-only applications.

Finally, we analyze the false negatives qualitatively to identify common trends and differences in the kind of bugs missed by the three systems. Figure 4 clusters the false negatives based on the types of bugs missed, identified by the CWE assigned to the test cases. The descriptions of relevant CWE, taken from MITRE CWE database [54], is summarized in Table IV. The figure agrees with the general trend in accuracy of the three systems: source ASan, followed by `ASan-retrowrite`, and finally Valgrind memcheck. However, note the differences, e.g., both source ASan and `ASan-retrowrite` miss 38 bugs related to Heap-based Buffer Overflow (CWE122) arising due to Incorrect Calculation of Multi-Byte String Length (CWE135) while Valgrind memcheck misses none. This result is surprising as we do not expect a binary-only tool to detect bugs that a source-based solution fails to. On further inspection, we found that all the testcases in CWE135 trigger an overflow in the destination buffer through a call to `wcscpy` (the `strcpy` equivalent for wide-character strings), which is not intercepted by the ASan runtime library and hence is not instrumented. Adding support for this function allows ASan to detect these violations. Similarly, both the binary-only tools miss an equal number of Off-by-one Errors (CWE193) while source ASan misses none when its on the heap (CWE122) and misses far fewer when on the stack (CWE121). We attribute this difference to more accurate object size information available in source code, empowering compiler-based tools.

On the heap (CWE122), `ASan-retrowrite` has more false negatives than ASan, even though the redzone policy is identical. This is because `ASan-retrowrite` misses some checks on `rep` prefixed instructions, i.e., `rep stos`. This is commonly used to implement operations that loop over buffers, such as `memcpy` and `memset`. To guarantee that an overflow due to such instruction is caught,

`ASan-retrowrite` would need to check if any of the accesses are in the redzone by, e.g., checking if the first and last bytes accessed are within the allowed region. The current implementation only checks the first access and therefore misses some of the overflows that could otherwise be caught. Implementing support for `rep` prefixes requires additional engineering effort in our future work.

D. Fuzzer Evaluation

To evaluate the effectiveness of our binary-only coverage instrumentation, we compare our approach against the current alternatives for instrumenting code to collect coverage. To summarize, we evaluate the following systems:

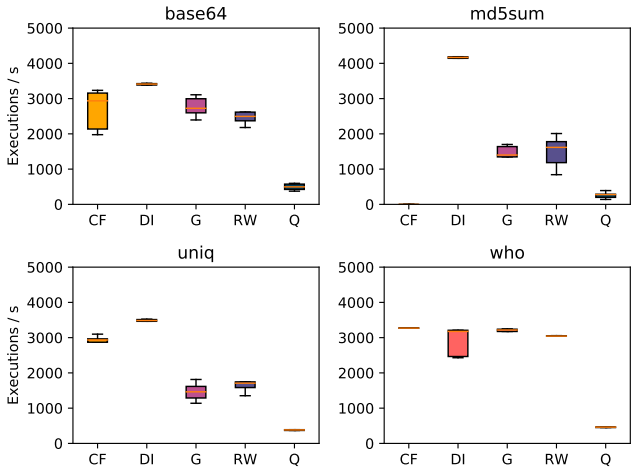

Figure 5: Box plot of fuzzing executions per second on LAVA-M across five, 24 hour trials. Legend: CF=afl-clang-fast, DI=afl-dyninst, G=afl-gcc, RW=afl-retrowrite (our solution), Q=QEMU. Higher numbers indicate better performance.

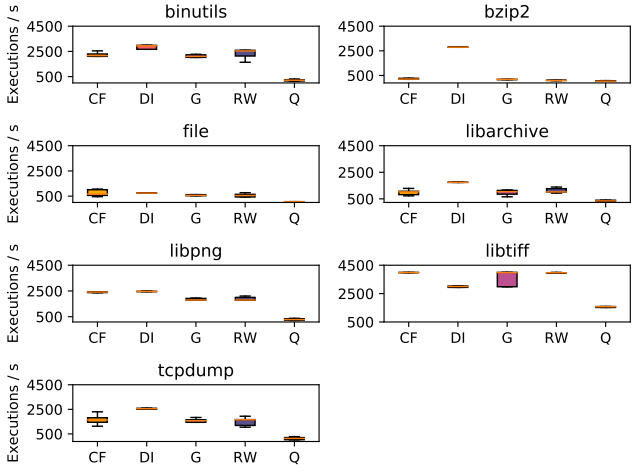

Figure 6: Box plot of fuzzing executions per second on real-world targets across five, 24 hour trials. Legend: CF=afl-clang-fast, DI=afl-dyninst, G=afl-gcc, RW=afl-retrowrite (our solution), Q=QEMU. Higher numbers indicate better performance.

CF: Source code instrumentation at LLVM-IR level, through `afl-clang-fast`,
G: Source code instrumentation at assembly level, through `afl-gcc`,
Q: Runtime instrumentation through `afl-qemu`
DI: Static rewriting through trampolines, through `afl-dyninst`, and
RW: Static rewriting through `afl-retrowrite` (our solution).

As discussed, the number of executions per second (the fuzzer throughput) is important as exploring more inputs directly correlates to larger probability of finding bugs through fuzzing. The fuzzer throughput alone does not give us a complete picture as a high throughput can be achieved by not instrumenting the binary to collect coverage. Such a fuzzer would achieve low coverage and hence discover fewer bugs. Therefore we evaluate along two axes: (i) executions per second, and (ii) number of unique bugs triggered, on the LAVA-M benchmarks [11]. To compare real-world fuzzing performance, we also evaluate the above systems on seven different libraries: (i) readelf (binutils), (ii) bzip2, (iii) file, (iv) bsdtar (libarchive), (v) pngfix (libpng), (vi) tiff2rgba (libtiff), and (vii) tcpdump. All binaries are statically linked against their respective libraries to ensure both the executable and the library are instrumented to collect coverage. We were unable to run the md5sum binary from LAVA-M when compiled with `afl-clang-fast`: the resulting binary crashes (`segfaults`) on any input. Hence, this result for `afl-clang-fast` has been omitted.

Because of its randomness, a single fuzz trial does not give us a complete picture. We follow guidelines for fuzzing as presented in [55] and conduct five trials with a 24-hour timeout per trial. For LAVA-M, the testcase included with the source is used as the initial seed for fuzzing. For fuzzing real-world applications, we used the testcases provided with the library as the initial fuzz seed (tcpdump), and fuzzing seeds included as a part of AFL for the other applications (readelf, bzip2, file, bsdtar, pngfix, tiff2rgba). The box plot for fuzzing performance across these trials is presented in Figure 5 and Figure 6. For LAVA-M, the number of unique bugs found in each of the five trials is presented in the Appendix in Table VI.

Our performance evaluation shows that `afl-qemu` is consistently the slowest mechanism for coverage instrumentation. This is expected as `afl-qemu` instruments the binary at runtime, incurring a higher overhead when compared to approaches that instrument statically. `afl-dyninst` has the highest throughput among the systems that we tested, outperforming compiler-based instrumentation, `afl-clang-fast`, on targets such as bzip2. This is surprising as we do not expect a trampoline-based binary rewriting solution to perform better than compiler-based instrumentation. This is likely due to the coverage instrumentation by `afl-dyninst` being ineffective at guiding the fuzzer through deeper program paths, thereby achieving a shallow coverage and a larger number of executions per second. Our evaluation of discovered bugs indicates that this is in-

Table V: Overview of p-values from Mann-Whitney U Test, comparing afl-retrowrite (**RW**) v/s afl-gcc (**G**), and afl-retrowrite (**RW**) v/s afl-qemu (**Q**). $p < 0.05$ indicates the results are statistically significant. Values rounded to third decimal place.

	RW v/s Q		RW v/s G	
	U-value	Means	U-value	Means
binutils	0.006	2298.81 / 216.07	0.105	2298.81 / 1962.40
bzip2	0.010	104.91 / 48.65	0.018	104.91 / 172.52
file	0.006	568.46 / 58.26	0.417	568.46 / 539.57
libarchive	0.006	1124.81 / 353.22	0.265	1124.81 / 971.91
libpng	0.006	1783.53 / 285.75	0.417	1783.53 / 1759.76
libtiff	0.006	3775.38 / 1503.9	0.500	3775.38 / 3589.85
tcpdump	0.006	1498.07 / 131.35	0.417	1498.07 / 1525.01
base64	0.006	2456.39 / 429.99	0.071	2456.39 / 2764.03
md5sum	0.006	1486.15 / 258.69	0.338	1486.15 / 1379.45
uniq	0.006	1684.75 / 372.38	0.148	1684.75 / 1464.21
who	0.006	2895.17 / 445.28	0.047	2895.17 / 3067.26

deed the case: `afl-dyninst` fails to find bugs despite having a high fuzzing throughput. The other three systems, `afl-clang-fast`, `afl-gcc`, and `afl-retrowrite` achieve similar throughputs across all runs.

Due to fuzzing randomness, we need a statistical test to determine if the deviations in throughput values are statistically significant or just an artifact of the randomness. To do so, we perform Mann-Whitney U test as suggested by Kless et al. [55]. We compare `afl-retrowrite` v/s `afl-qemu` and `afl-gcc` in Table V. All p-values comparing binary AFL and QEMU are < 0.05 indicating that the difference in performance is statistically significant, while p-values comparing binary and source AFL are > 0.05 indicating that the differences in performance are likely due to the fuzzer's randomness. Consequently, we conclude that our binary version improves over QEMU. Performance of `afl-retrowrite` is within measurement noise of source AFL. Note the repeated p-value of 0.006, which corresponds to the smallest achievable value for tests with 5 runs, indicating that each single element of array A is smaller than any element of array B.

The number of bugs discovered by AFL directly shows the effectiveness of coverage instrumentation and fuzzer throughput. Appendix Table VI shows that the QEMU-based instrumentation, `afl-qemu`, only finds two bugs in a single run of uniq. The low throughput of afl-qemu prevents the fuzzer from exploring a large input space and achieving high coverage leading to lower number of bugs found. In contrast, afl-dyninst has high throughput, but fails to find many bugs in our evaluation: only finding a total of 3 bugs across all runs on base64. This is likely due to ineffective (or incomplete) coverage instrumentation, preventing the fuzzer from exploring deeper paths. Finally, `afl-retrowrite` and the two source-based solutions, afl-clang-fast and afl-gcc, discover a similar number of unique bugs across all the runs and benchmarks (barring the failed evaluation of afl-clang-fast on md5sum benchmark). Our evaluation shows that our solution, `afl-retrowrite`, is a viable alternative to afl-qemu for binary-only applications and is identical to source-based solutions, both in terms of performance and bug-finding capabilities.

VII. Discussion

a) Support for C++ Binaries: The current implementation of `RetroWrite` cannot rewrite C++ binaries safely. This is primarily due to missing symbolization for C++ exception handlers. Information required to unwind stack-frames are stored in compressed DWARF format [56] which contain code references that we do not symbolize. This is in line with previous work on reassembleable assembly and a limitation shared by all current approaches. Theoretically, `RetroWrite` does support C++ binaries that do not involve exception handling. However, this has not been tested extensively. We leave the engineering effort of adding support for exception handling as future work.

b) Closing the Performance Gap: Although `ASan-retrowrite` is significantly faster than Valgrind memcheck, there is still a slowdown when compared to ASan. Our current `ASan-retrowrite` instrumentation is directly taken from assembly generated by the compiler for instrumenting several crafted test cases. We did not invest additional effort in hand-optimizing assembly. Therefore, our current `ASan-retrowrite` instrumentation may not be the most optimized version possible. Another opportunity to reduce overhead is to remove unnecessary checks when a memory access is known to be safe, e.g., accessing variables on stack through constant offsets from the stack top. We notice through a preliminary study that `ASan-retrowrite` instruments about 50% - 70% more checks than ASan.

c) Limitations of `ASan-retrowrite`: The limitations of `ASan-retrowrite` on stack and global sections are fundamental to static binary rewriting. To improve precision on stack and data sections, we may need to trade-off soundness or scalability. Though the above holds for the general case, a *soundy* analysis may work for most compiler generated binaries, and it may still be possible to push towards higher precision without introducing false positives. One attractive option is to use local symbolic execution to track base-pointers and disambiguate references. We leave this as an option for future work.

d) Obfuscation: To protect intellectual property, some vendors ship obfuscated binaries. Our framework does not address obfuscation. Binary unpacking is usually specific to the obfuscation scheme used; and an obfuscated binary may be rewritten by `RetroWrite`, after it is pre-processed by a de-obfuscation step [57], [58]. Similar to other efforts in binary analysis, we do not consider adversarial binaries, i.e., binaries crafted to defeat rewriting efforts.

VIII. Related Work

In this section we discuss related work that are both complementary and orthogonal to our efforts in rewriting, fuzzing instrumentation, and memory checker.

a) Fuzzing: `RetroWrite` allows seamless implementation of binary AFL without any changes being made to the original AFL framework itself. This enables any advancements made towards improving AFL to be integrated into our binary

AFL solution without any additional effort. These advancements could range from smarter seed selection [59], [60], [61] to even transformational fuzzing [62].

b) Binary Rewriting: Several approaches to binary rewriting have been proposed, these can be broadly divided into two categories based on time of instrumentation: (a) Dynamic Binary Translation (DBT) based approaches [10], [26], [63], [29], [30], [15] that instrument the binary at runtime, and (b) Static binary rewriting based approaches [14], [16], [31], [12], [13], [15], [32] that instrument binaries on disk. While DBT-based approaches are scalable and widely used for real-world rewriting, they have higher overhead making them unsuitable for highly performant instrumentation. Static binary rewriting approaches have been limited to smaller binaries, have higher memory or runtime overhead, or are limited by the types of transformations that they support.

c) Reassembleable Assembly: `RetroWrite` uses reassembleable assembly at its core to perform rewriting. Reassembleable assembly was first introduced by Uroboros [13] and then improved upon by `ramblr` [12]. As noted by Wang et al. [12] symbolizing a binary is undecidable in general as it requires analysis to distinguish between scalars and references statically, which was shown undecidable [33]. Our work is inspired by these existing approaches, but we *trade-off generality for soundness* and target position-independent code. Consequently, while we are limited to position-independent code, we ensure that `RetroWrite` is correct by construction, not requiring any heuristics. Furthermore, our approach is scalable to rewrite larger, real-world applications. Lastly, both Uroboros and `ramblr` focus on rewriting x86 32-bit binaries (and had symbolization false negatives on 64-bit binaries); whereas `RetroWrite` focuses on x86-64 bit PIC binaries.

d) Disassembly and Control Flow Recovery: Most binary analyses use disassembly as a first step in the tool-chain. For architectures that allow variable length instructions and intermixing of code and data, disassembling an executable is an undecidable problem as it requires analysis to make this distinction between code and data. Two established techniques in disassembly are linear sweep and recursive descent and are discussed in depth by Schwarz et al. [19]. As pointed out by Andriesse et al. [25], achieving high-level of disassembly accuracy for mainstream compilers, such as clang, gcc, and Visual Studio, is possible with techniques such as linear sweep even when the binary is optimized. Surprisingly, their evaluation shows that linear sweep as implemented in `objdump` outperforms tools that use more sophisticated techniques. Though our current implementation uses linear sweep and has been sufficient for all our evaluations, we can reuse existing tools [20], [21] to handle other edge-cases.

Control-flow recovery [64], [65], [66], [67], [68] improves disassembly coverage and vice-versa, and therefore is implemented as tightly coupled passes in `angr` [23]. `RetroWrite` does not require precise recovery of the CFG as indirect calls and jumps are symbolized at the program point where the address is taken, rather than at the point of control-flow transfer. However, the rewriter API can support a richer set of transformations at basic block granularity with a more precise CFG if the desired instrumentation requires it.

e) Function Identification: Modern approaches [39], [40], [41], [42] use machine learning to detect function start and sizes. Tools such as IDA [20] and radare [21] implement architecture and compiler specific heuristics to detect function boundaries. These techniques have different trade-offs in terms of precision and accuracy. Any of these existing techniques may be reused in our framework, as a pre-processing step, to support instrumentation of stripped binaries. Note that the accuracy of function identification does not affect our rewriting correctness, but has implications on the instrumentation API, e.g., a function level instrumentation pass may miss instrumenting some functions if they are not identified.

f) Variable and Type Information Recovery: Recovering types and variable information is an important step in decompilation as it leads to more natural looking code, and hence much effort has been spent on this subject [46], [47], [48], [49]. These techniques are inherently geared towards readability and can tolerate some degree of unsoundness. Any of these techniques can be used to identify stack variables and instrument `ASan-retrowrite` checks at a finer-granularity. However, this is a trade-off between precision and soundness of error reporting. Any errors in variable recovery may lead to false positives, which may or may not be desirable based on an individual use-case.

IX. Conclusion

`RetroWrite` is a principled, zero-cost rewriter for PIC binaries. To aid fuzzing blackbox binaries, we develop two instrumentation passes in `RetroWrite`: (i) `ASan-retrowrite`, a binary-only memory checker, based on and compatible with ASan, and (ii) binary-AFL to collect coverage for greybox fuzzing. We show that `ASan-retrowrite` is significantly better than the current state-of-the-art binary-only memory checker, Valgrind memcheck, both in terms of performance and coverage. `ASan-retrowrite` is compatible with ASan, thereby allowing users to selectively rewrite closed-source parts of code with `ASan-retrowrite` while compiling the rest of the code with ASan. For coverage-guided fuzzing, our binary-only AFL instrumentation is at least as good as the source-based AFL instrumentation, and far better than the current coverage collection for blackbox binaries using QEMU. `RetroWrite` is open source: `https://github.com/HexHive/retrowrite/`.

X. Acknowledgements

We thank our shepherd Brendan Dolan-Gavitt and the anonymous reviewers for their insightful comments. This research was supported by ONR award N00014-17-1-2513 and by NSF CNS-1801601. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[2] M. Corporation, "A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003," https://support.microsoft.com/en-us/kb/875352, 2013.

[3] C. Cowan, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, A. D. Rubin, Ed. USENIX Association, 1998.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, V. Atluri, C. A. Meadows, and A. Juels, Eds. ACM, 2005, pp. 340–353.

[5] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *CSUR*, 2017.

[6] M. Zalewski, "american fuzzy lop," 2017, [Online; accessed 1-December-2018]. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[7] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318.

[8] "Radamsa," https://gitlab.com/akihe/radamsa, accessed: 2018-11-24.

[9] "Afl blackbox," [Online; accessed 1-December-2018]. [Online]. Available: https://github.com/mirrorer/afl/tree/master/qemu_mode

[10] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 89–100.

[11] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. K. Robertson, F. Ulrich, and R. Whelan, "LAVA: large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 110–121.

[12] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[13] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 627–642.

[14] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in COTS software with binary rewriting," in *Future Challenges in Security and Privacy for Academia and Industry - 26th IFIP TC 11 International Information Security Conference, SEC 2011, Lucerne, Switzerland, June 7-9, 2011. Proceedings*, ser. IFIP Advances in Information and Communication Technology, J. Camenisch, S. Fischer-Hübner, Y. Murayama, A. Portmann, and C. Rieder, Eds., vol. 354. Springer, 2011, pp. 154–172.

[15] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE'11, Szeged, Hungary, September 5-9, 2011*, J. Foster and L. L. Pollock, Eds. ACM, 2011, pp. 9–16.

[16] Z. Deng, X. Zhang, and D. Xu, "BISTRO: binary component extraction and embedding for software security applications," in *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, ser. Lecture Notes in Computer Science, J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134. Springer, 2013, pp. 200–218.

[17] "Afl dyninst," [Online; accessed 1-December-2018]. [Online]. Available: https://github.com/vrtadmin/moflow/tree/master/afl-dyninst

[18] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 213–223.

[19] B. Schwarz, S. K. Debray, and G. R. Andrews, "Disassembly of executable code revisited," in *9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*, A. van Deursen and E. Burd, Eds. IEEE Computer Society, 2002, pp. 45–54.

[20] "Ida pro," https://www.hex-rays.com/products/ida/, accessed: 2018-11-24.

[21] "Radare," http://rada.re/r/, accessed: 2018-11-24.

[22] V. 35, "binary.ninja : a reversing engineering platform," https://binary.ninja/, accessed: 2018-11-24.

[23] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 138–157.

[24] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. M. Thuraisingham, "Differentiating code from data in x86 binaries," in *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part III*, ser. Lecture Notes in Computer Science, D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds., vol. 6913. Springer, 2011, pp. 522–536.

[25] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 583–600.

[26] H. Patil, R. S. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA*. IEEE Computer Society, 2004, pp. 81–92.

[27] D. Bruening, T. Garnett, and S. P. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA*, R. Johnson, T. Conte, and W. W. Hwu, Eds. IEEE Computer Society, 2003, pp. 265–275.

[28] B. Derek, G. AI, A.-J. Chris, G. E. Edmund, and Z. Kevin, "Building dynammic tools with dynamorio on x86 and armv8," https://github.com/DynamoRIO/dynamorio/releases/download/release_7_0_0_rc1/DynamoRIO-tutorial-feb2017.pdf, 2018, [Online; accessed 28-Feb-2018].

[29] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 41–46.

[30] M. Payer and T. R. Gross, "Fine-grained user-space security through virtualization," in *ACM International Conference on Virtual Execution Environments*, 2011.

[31] M. Smithson, K. Elwazeer, K. Anand, A. Kotha, and R. Barua, "Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, R. Lämmel, R. Oliveto, and R. Robbes, Eds. IEEE Computer Society, 2013, pp. 52–61.

[32] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[33] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *Comput. J.*, vol. 23, no. 3, pp. 223–229, 1980.

[34] "Ubuntu Newsletter," https://lists.ubuntu.com/archives/ubuntu-devel/2017-June/039816.html, accessed: 2018-11-24.

[35] "Fedora Harden All Packages," https://fedoraproject.org/wiki/Changes/Harden_All_Packages, accessed: 2018-11-24.

[36] "Gentoo Profiles 17.0," https://www.gentoo.org/support/news-items/2017-11-30-new-17-profiles.html, accessed: 2018-11-24.

[37] "Android Lollipop Security Enhancements," https://source.android.com/security/enhancements/enhancements50.html, accessed: 2018-11-24.

[38] "iOS Building PIE," https://developer.apple.com/library/archive/qa/qa1788/_index.html, accessed: 2018-11-24.
[39] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, D. Fox and C. P. Gomes, Eds. AAAI Press, 2008, pp. 798–804.
[40] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 611–626.
[41] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 845–860.
[42] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 177–189.
[43] "Msrc - trends, challenges, and strategic shifts in the software vulnerability mitgation landscape," https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01-BlueHatIL-Trends,challenge, andshiftsinsoftwarevulnerabilitymitigation.pdf, accessed: 2019-05-01.
[44] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 245–258.
[45] ——, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, J. Vitek and D. Lea, Eds. ACM, 2010, pp. 31–40.
[46] G. Balakrishnan and T. W. Reps, "DIVINE: discovering variables IN executables," in *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, 2007, pp. 1–28.
[47] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*, 2010.
[48] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_3.pdf
[49] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 27–41.
[50] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, 1994.
[51] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX Security Symposium*. USENIX Association, 2018, pp. 745–761.
[52] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 711–725.
[53] "Asan patch for spec cpu2006," [Online; accessed 1-December-2018]. [Online]. Available: https://github.com/google/sanitizers/blob/master/address-sanitizer/spec/spec2006-asan.patch
[54] "MITRE CWE Database," https://cwe.mitre.org/, accessed: 2018-11-24.
[55] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2123–2138.
[56] "Dwarf standard specification," [Online; accessed 1-December-2018]. [Online]. Available: http://dwarfstd.org/Dwarf5Std.php
[57] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.
[58] M. A. Ben Khadra, D. Stoffel, and W. Kunz, "Speculative disassembly of binary code," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '16. New York, NY, USA: ACM, 2016, pp. 16:1–16:10.
[59] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2329–2344.
[60] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1032–1043.
[61] C. Lemieux and K. Sen, "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage," *ASE 2018- Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 475–485, 2018.
[62] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 697–710.
[63] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.
[64] C. Cifuentes and M. V. Emmerik, "Recovery of jump table case statements from binary code," in *7th International Workshop on Program Comprehension (IWPC '99), May 5-7, 1999 - Pittsburgh, PA, USA*. IEEE Computer Society, 1999, pp. 192–199.
[65] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.
[66] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 423–427.
[67] C. Krügel, W. K. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, 2004, pp. 255–270.
[68] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 829–844.

APPENDIX

Table VI: Number of unique bugs found in five fuzzing trials. All trials used input provided with the LAVA-M dataset as initial seeds. Each trial was run for 24 hours. **Legend.** CF: afl-clang-fast, G: afl-gcc, Q: afl-qemu, DI: afl-dyninst, RW: afl-retrowrite.

	RW	CF	DI	G	Q
base64	[5, 2, 0, 6, 1]	[4, 2, 2, 1, 2]	[1, 2, 0, 0, 0]	[2, 1, 2, 2, 3]	[0, 0, 0, 0, 0]
md5sum	[1, 0, 0, 1, 0]	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0]
uniq	[1, 1, 3, 2, 2]	[1, 1, 3, 1, 2]	[0, 0, 0, 0, 0]	[1, 3, 1, 0, 0]	[0, 0, 0, 0, 2]
who	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0]