RETROWRITE: STATICALLY INSTRUMENTING COTS BINARIES FOR FUZZING AND SANITIZATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Sushant Dinesh

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

April 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

Professor Mathias Payer, Chair
Department of Computer Science
Professor Dongyan Xu
Department of Computer Science
Professor Benjamin Delaware
Department of Computer Science

Approved by:
Professor Voicu Popescu
Head of the Computer Science Graduate Program

ACKNOWLEDGMENTS

I want to thank my advisor, Prof. Mathias Payer, for his continued support and guidance. His feedback has been instrumental in both my personal and professional development. Thank you for being a constant source of inspiration. I will always cherish the time I spent in your research group and hope to continue collaboration in the future. I would also like to thank the members of my committee, Prof. Dongyan Xu and Prof. Benjamin Delaware, for their insightful feedback and questions.

I want to thank the members of HexHive group, both past and current, for their insightful feedback. Your reviews on writing and presentation is much appreciated and helped me grow as a researcher. I want to specially thank Prashast for all the discussions and putting up with my crazy ideas. I enjoyed our discussions and I learned a lot about formulating and communicating ideas through them. I wish you all the best with your Ph.D. and beyond.

I want to thank my family for their constant support and encouraging me to pursue graduate studies. Without their support, I would not be able to pursue my dreams fearlessly. I want to thank my father for being an inspiration in my life and motivating me to pursue a Ph.D. and my mother for her unconditional emotional support. I want to thank my sister, Dyuthi, for all the fun discussions and reminding me that there is a life outside of computer science.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Dinesh, Sushant M.S., Purdue University, April 2019. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. Major Professor: Mathias Payer.

End users of closed-source software currently cannot easily analyze the security of programs or patch them if flaws are found. Notably, end users can include developers who use third party libraries. The current state of the art for coverage-guided binary fuzzing or binary sanitization is dynamic binary translation, which results in prohibitive overhead. Existing static rewriting techniques cannot fully recover symbolization information, and so have difficulty modifying binaries to track code coverage for fuzzing or add security checks for sanitizers.

The ideal solution for adding instrumentation is a static rewriter that can intelligently add in the required instrumentation as if it were inserted at compile time. This requires 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 our observation, we design a binary-rewriting instrumentation to support American Fuzzy Lop (AFL) and Address Sanitizer (ASan), and show that we achieve compiler levels of performance, while retaining precision. Binaries rewritten for coverage-guided fuzzing using `RetroWrite` are identical in performance to compiler-instrumented binaries and outperforms the default QEMU-based instrumentation by *7.5x* while triggering more bugs. Our implementation of binary-only Address Sanitizer is *3x* faster than Valgrind memcheck, the state-of-the-art binary-only memory checker, and detects 80% more bugs in our security evaluation.

1. INTRODUCTION

Software is often large and complex and therefore prone to exploitable vulnerabilities. While mitigations such as ASLR [1], DEP [2], Stack Canaries [3], or CFI [4,5] protect the software against the active exploitation of some vulnerabilities they cannot protect software against the exploitation of all vulnerabilities and they are of limited use in discovering the locations of vulnerabilities. Best practices in software testing leverage coverage-guided fuzzing and sanitizers to discover bugs while producing concrete test cases to reproduce the bugs in a testing environment. Both coverage-guided fuzzing as well as sanitization requires additional instrumentation of the code. Current standard tools such as AFL [6], a coverage-guided fuzzer or Address Sanitizer (ASan) [7] require a recompilation of the source code to (i) get high level information about where to place the instrumentation and (ii) actually weave the instrumentation into the compiled code.

Unfortunately, code is frequently only available in binary form. For example, many libraries or binaries such as the Google Hangouts plugin, Skype, or Zoom are not available as source code to protect the intellectual property of the vendors. Without access to source code, testing the security of such programs is challenging. Existing approaches either give up and rely on, e.g., blackbox fuzzing [8] which often results in shallow coverage or rely on dynamic binary translation to add coverage tracking [9] or memory checks [10] to the executed binary at prohibitively high runtime cost of 100x and more (running AFL in QEMU mode on the LAVA-M [11] test suite results in 10x to 100x slowdown).

A static binary translation technique that enables code and data changes would allow a static binary analysis to rewrite the binaries and achieve performance similar to compiler-based techniques. Current static binary rewriting techniques [12,13] have low overhead, but their reliance on heuristics makes them unsound. The fundamental difficulty is statically disambiguating between reference and scalar constants, so that a program can be “reflowed”, i.e., have its code and data pointers rewritten according to any inserted instrumentation or changed data sections. During assembly, labels are translated into relative offsets or relocation entries. A static binary rewriter must recover all these offsets correctly. Existing work on static rewriting fits into three buckets based on how this problem is approached: (i) recompilation [14], which attempts to lift the code to an intermediate representation; (ii) trampolines [15,16], which rely on indirection to insert new code segments without changing the size of basic blocks; and (iii) reassembleable assembly, which creates an assembly file equivalent to what a compiler emits, 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 adds performance overhead. Therefore, we believe that resymbolizing binaries for reassembleable assembly is the most promising technique for static binary rewriting.

We show that static binary rewriting, leveraging reassembleable assembly, can produce sound and efficient code for an important subset 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. We achieve this by leveraging the relocation information that is required for position independent code, and produce assembly files that an assembler cannot distinguish from compiler generated code. Whereas source based techniques rely on register allocation later in the compilation process to assign registers for their instrumentation, we provide a register liveness analysis that prevents unnecessary spilling of registers to the stack for instrumentation. To show the usefulness of our rewriting framework, we present case studies on rewriting for AFL and ASan.

Fuzzing is one of the most effective forms of software testing and is widely adopted as a part of the software development cycle. Blackbox fuzzing [8], i.e., fuzzing without any insight about the application under test, requires well-curated test cases as it otherwise achieves only shallow coverage. Coverage-guided fuzzing overcomes this limitation by using program traces as a feedback mechanism to choose future input sequences. AFL [6] is a popular coverage-guided greybox fuzzer and has been instrumental in discovering massive amounts of bugs [17]. AFL instruments applications at compile time to collect edge-coverage. To collect coverage for fuzzing binary-only applications, AFL ships with a QEMU-based instrumentation tool. Even with several optimizations, the QEMU based tool is about 3x - 30x slower than the corresponding source instrumented counterpart which severely restricts the fuzzing throughput and further, the ability to find bugs. Additionally, instrumentation such as ASan is not supported in QEMU mode [9] thereby limiting AFL's ability to find bugs in blackbox binaries.

To effectively fuzz test binaries, we identify two key requirements: (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 (not break binaries) and scalable to support real-world use cases. Attempts to statically instrument binaries using DynInst [18] are not widely adopted as they do not satisfy the second criteria. We leverage `RetroWrite` to develop a pass to statically instrument binaries such that the instrumented binaries are *just as performant as their compiler instrumented counterparts.* As `RetroWrite` is fundamentally sound in rewriting the binaries it supports, our solution can be widely deployed and used as a replacement to the current QEMU-based instrumentation when fuzzing position-independent code.

Fuzzers depend on program crashes to detect and report bugs. Consequently, bugs that do not trigger crashes are not caught through fuzzing. Address Sanitizer (ASan) is the most widely used, tripwire-based memory checker that greatly increases the probability of a program crash when a memory corruption bug is triggered. In addition to increasing the probability of detecting a bug, ASan provides a detailed backtrace to help developers understand and patch the underlying bug. ASan is implemented as a compiler pass and instruments code at compile-time. The availability of source information allows ASan to be far superior in terms of performance and bug detection rate when compared to binary-only solutions [10, 19]. 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 Binary Address Sanitizer (BASan) as an instrumentation pass in `RetroWrite` to retrofit binaries with memory checks to aid in fuzzing binaries. Our approach is similar to and inspired by ASan as implemented in the compiler. BASan is both lightweight and finds more bugs when compared to Valgrind memcheck. Additionally, BASan is compatible with ASan, thereby enabling blackbox components of a software, e.g., closed-source or legacy libraries, to be rewritten by BASan while compiling the rest of the codebase with ASan. In short, BASan is better than existing binary-only memory checkers while being compatible with source-based solutions.

We evaluate `RetroWrite`'s AFL coverage instrumentation, `afl-retrowrite`, for fuzzing throughput and effectiveness in finding bugs. We compare `afl-retrowrite` against compile-time instrumentation mechanisms, afl-clang-fast and afl-gcc, and binary-only instrumentation mechanisms, afl-dyninst and afl-qemu. On LAVA-M bechmarks, `afl-retrowrite` finds more bugs than the other binary-based solutions and is equivalent to the compiler-based counterparts — finding a total of 25 bugs across runs of the benchmarks, 22 more than the other binary-based solutions. `afl-retrowrite` is a viable replacement to afl-qemu for binary only applications, achieving about *4.5x higher fuzzing throughput* compared on QEMU instrumentation on real-world applications and comparable in performance to compiler-based instrumentation.

We evaluate BASan for performance and bug detection rate. Our performance evaluations show that BASan has a massive *3x speedup* against Valgrind memcheck and is only *0.65x slower* than ASan (due to the lack of compiler optimization and register pressure). We evaluate BASan's capability to find bugs on the Juliet test suite, a curated set of test cases representative of real-world bugs. On the Juliet test suite, BASan has a higher bug detection rate[1] of 0.55 when compared to 0.30 Valgrind memcheck, while it still falls short of ASan (0.75). In short, BASan is better in terms of performance and bug detection rate compared to the state-of-the-art binary-only memory checker, Valgrind memcheck. To summarize, our BASan and AFL instrumentation implemented in `RetroWrite` are significantly better than current state-of-the-art tools for fuzzing blackbox binaries, integrate with source-based tools for compatibility, and are viable drop-in replacements. In short, our contributions are:

- A static binary rewriting framework[2] that allows sound, efficient rewriting of 64 bit PIC binaries (chapter 2),
- An instrumentation pass[2] that allows binaries to be run with AFL with the same performance as compiler-based AFL instrumentation (chapter 4),
- An instrumentation pass[2] that retrofits binaries with ASAN checks, increasing by orders of magnitude the efficiency of memory safety analysis for binaries (chapter 3),
- A comprehensive evaluation of BASan and AFL instrumentation (chapter 5) on benchmarks and real-world applications, followed by a discussion of limitations (chapter 6).

[1]Number of bugs detected / Total number of bugs
[2]Implementation will be open-sourced on acceptance.

2. STATIC BINARY REWRITING

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 through, e.g., coverage-guide greybox fuzzing on *binaries*, and binary-only memory checkers with near source-based performance. However, rewriting binaries is not as straightforward as editing source code, mainly due to the fact that binaries lack source-like abstractions. Binaries lack type information, and data-structure abstractions are flattened to raw memory accesses. 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:

- **Performance:** Have low runtime and memory overhead when the binary is recompiled with instrumentation.
- **Correctness:** Preserve as much of the original program characteristics (barring changes made by the instrumentation) as possible. This ensures that any bugs found directly translate to the original binary.
- **Scalability:** Scale to real-world software.

Existing DBT-based techniques [10, 15, 20–22] do not satisfy our performance criteria while existing work on static binary rewriting [12–16, 23, 24] do not satisfy at least two of the three requirements.

2.1 Background

This section introduces the reader to building blocks of binary analysis and binary rewriting.

Disassembly Disassembly is the first step on binary rewriting, and is used to recover the existing instructions for analysis / modification. Disassembling a binary compiled for a variable length instruction set architecture is hard as the disassembler has to identify the instruction length. 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 in [25]. 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 [26] has been the industry standard for disassembling and reverse engineering, but there are other viable contenders such as radare2 [27], Binary Ninja [28], and static binary analysis frameworks such as angr [29]. 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 [30]. However, as pointed out by Andriesse et al. [31], the undecidablilty 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.

Binary Rewriting Binary rewriting techniques can be broadly classified into two categories based on when they instrument the binary:

- **Dynamic Binary Translation (DBT).** DBT translates the binary as it is executing. Consequently, they leverage runtime information and do not depend on complex static analysis that may not scale. This makes them an attractive solution for rewriting large software. Several off-the-shelf solutions for DBT exist, including Intel PIN [20], DynamoRIO [32, 33], QEMU [22], DynInst [15] and Valgrind [10]. The lightest weight DBT techniques, i.e., Intel PIN and DynamoRIO, have anywhere between ∼10% to ∼20% rewriting overhead, i.e., with no instrumentation.

- **Static Binary Rewriting.** Static rewriting translates binaries *before* they are executed. Since the instrumentation is performed ahead of time, 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 any generic piece of software. However, static rewriting is an active area of research with several research prototypes [12–14, 16, 23, 34]. Existing prototypes vary by runtime overhead, memory overhead, and the characteristics of rewritten binary.

No existing rewriter meets our design criteria for a security oriented rewriter. DBT suffers from prohibitive runtime overhead. 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 with the scalability of DBT and the overhead of static rewriting that remains precise enough to add security instrumentation is required. We design reassembleable assembly to fit this niche.

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 hardcoded 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 hardcoded addresses as these are assigned at link time.

Reassembleable assembly creates assembly files that appear to be compiler generated, i.e., they do not contain hardcoded values but instead assembly labels. The core of generating reassembleable assembly is thus the process of *symbolization*, i.e., converting reference constants into assembler labels. Symbolizing the assembly allows security 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 by Uroboros [13]. Wang et al. designed 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 the symbolization accuracy. Additionally, ramblr acknowledges that their rewriting can never be complete and develop heuristics to abort the reassembly safely when they cannot guarantee rewriting correctness.

Reassembly is the most 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 in the framework that transform the assembly files before they are finally reassembled to produce the instrumented binary.

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 disambiguating whether a constant represents a scalar or a reference is infeasible [35]. Therefore, existing techniques are empirical and use heuristics to approximate a sound static analysis. While these work in most cases they are generally insufficient to rewrite real-world binaries, e.g., ramblr (the current 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 missclassifications, which prevent the binary from being rewritten correctly. However, while this restriction holds for the general case, there is hope for position independent binaries.

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 being fixed. However, recent architectural advancements, e.g, being able to reference 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 [36], Fedora [37], and Gentoo [38], have switched over to compiling and shipping binaries as PIC by default. In the smartphone ecosystem, Android has removed support for non-PIE linker and compiled binaries have to be PIE since Lollipop [39]. Though iOS does not forbid non-PIC binaries as Android does, it strongly encourages PIC and emits warnings for non-PIC binaries [40]. 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 position independent code by leveraging the relocation information it contains for symbolization, and rely on existing approaches to support non-PIC binaries.

2.2 RetroWrite

`RetroWrite` implements static rewriting through reassembleable assembly. The core of generating reassembleable assembly is the process of *symbolization*, i.e., statically disambiguating between reference and scalar type for constants and replacing references with appropriate assembler labels. In the general case, this requires heuristics. However, for PIC we present a principled symbolization strategy without any reliance on heuristics. `RetroWrite` leverages the relocation information in PIC binaries that enables the dynamic linker/loader to load the binary at arbitrary addresses for symbolization. This information is sufficiently detailed to allow us to reconstruct all labels that the compiler originally emitted before the binary was assembled.

`RetroWrite` is designed as a framework, with the reassembleable assembly based rewriter at its core. The rewriter serves as a central framework, on top of which other modules can be added to transform the assembly files to, e.g., track coverage for greybox fuzzers or add redzones for ASAN. The rewriting framework exposes an API to the modules for these transformation, and includes useful information such as register liveness for transformations to use. At a high-level, our rewriting framework has three major steps:

1. **Binary loading and Disassembly.** The first step is to load the sections required for reassembly. These include both the text and the data sections. Additionally, `RetroWrite` also loads auxiliary information, such as symbols and relocations from the binary. After loading, `RetroWrite` performs a linear-sweep to disassemble the text section.

2. **Control Flow Recovery.** After disassembling the binary, `RetroWrite` analyzes the disassembly to generate a best-effort CFG — identifying and adding

edges for direct control-flow transfers. However, `RetroWrite` does not resort to heavyweight analyses to infer indirect control-flow targets.

3. **Symbolization.** Finally, `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. At the end of the symbolization phase, `RetroWrite` dumps reassembleable assembly which may further be instrumented by other modules within the framework, much like editing assembly files.

The reassembleable assembly produced by the symbolization step is then available for the security modules to perform their transformations on. Once the final assembly file has been generated, it can be assembled using any off-the-shelf assembler to generate a working binary.

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. Our technique is lightweight and directly works on disassembly generated from any off-the-shelf disassembler. Consequently, our design makes it straightforward to extend the `RetroWrite` 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 all identified control-flow instructions, i.e., calls and jumps, are converted into assembler labels as these have to be code-to-code references.

2. **PC-relative Addressing**. As position-independent code cannot reference fixed addresses, references are calculated relative to the program counter (in

x86-64, this is `rip`). We identify instructions that compute such pc-relative addresses and convert the operand 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. Note that this implicitly covers cases of indirect jumps and calls as the reference to functions are symbolized at the point where address is taken, thereby making up for the 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 position-independent code.

2.3 Implementation

Our current implementation supports Linux x86-64 position independent executables. `RetroWrite` is implemented in about 2,000 lines of Python code and uses Capstone (a disassembly framework with support for multiple architectures) to disassemble raw bytes into x86-64 instructions. `RetroWrite` uses pyelftools, an ELF parsing library to load ELF files and parse relocation information.

Though our current implementation is restricted to the x86-64 architecture, other architectures supported by Capstone may easily be added with minimal engineering effort. We believe our prototype is robust and suitable for real-world binary rewriting. We will open-source our framework on acceptance.

Generating reassembleable assembly is the first step towards rewriting binaries. In practice, writing instrumentation passes to safely instrument binaries at a 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 based on the use-case, (ii) working around the ABI to ensure the instrumentation does not break the binary, and (iii) automatic register allocation to achieve compiler-like overhead. The following paragraphs discuss the implementation details about our instrumentation API built in `RetroWrite`.

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 radare, as a part of a pre-processing step. An alternative is to use other existing research in function identification [41–44]. These options are discussed further in related work section and may be reimplemented in our framework with additional engineering effort.

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 is not possible in such functions as it would overwrite the stack-local variables and lead to incorrect execution. To work around this, `RetroWrite` uses a static analyses to find such leaf functions. 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.

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 and infeasible in practice.

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 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.

3. BINARY ADDRESS SANITIZER

Software written in low-level languages, i.e., languages without memory-safety guarantees, such as C and C++ are susceptible to memory corruption bugs. These memory corruption bugs are the root cause of several vulnerabilities that an attacker exploits to gain arbitrary code execution capabilities. Efforts to enforce full runtime memory safety (spatial and temporal) for these applications have been prohibitively expensive [45, 46] 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, before the code makes it production. However, these bugs may be subtle and may not always be detected through a crash. Even if the software does crash, isolating the root cause is usually non-trivial for large software. Memory checkers are a class of software testing tools that detect these memory corruption bugs and terminate the application. They also provide a detailed backtrace that led to the crash, making bug isolation and patching easier. The usual practice is to use memory checkers in conjunction with unit tests and fuzz testing to catch memory corruption vulnerabilities before release.

Valgrind is a popular dynamic binary instrumentation (DBI) framework that allows one to develop dynamic analysis tools, such as memory checkers, for binary-only applications. While most other dynamic binary instrumentation frameworks, such as PIN [20] and DynamoRIO [21, 33], focus on performance, Valgrind is a framework designed for heavyweight binary analysis. Valgrind provides an efficient mechanism for dynamic analyses to associate and track metadata to every register and memory value, i.e., shadow values. Valgrind memcheck is a state-of-the-art memory checker implemented on top of Valgrind. 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 and incurs anywhere between 2x to 300x overhead. This makes memcheck unattractive for use with fuzzing where higher throughput directly correlates to deeper coverage and consequently greater probability of triggering bugs.

3.1 Address Sanitizer Semantics

Address Sanitizer (ASan) [7] is a trip-wire based approach to detect memory corruption. In short, ASan modifies the memory allocation of an application to pad every memory object with a redzone, forbidden regions 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:

ASan Allocation. All memory objects are padded with redzone — regions around allocated memory objects accessing which is illegal and triggers a fault. The specific redzone policy depends on the allocation region:

- *Heap.* ASan implements an instrumented version of `malloc` and `free` as a part of ASan runtime library. The specialized allocator allocates additional bytes for the redzones and poisons them before returning memory to the application. To detect use-after-free the free implementation poisons the freed memory region and tries to delay reuse of freed memory for as long as possible by maintaining a free-list.

- *Stack.* For stack objects, redzones for individual stack objects are created and poisoned at runtime, when a function enters. By default, the size of the stack redzone is 32 bytes. Allocations are padded to 32-byte alignment.

- *Global.* A redzone is inserted below every global statically. The size of the redzone is so chosen that the total size of the object (original size + redzone) is 64-byte aligned. Redzones for globals are poisoned at runtime as a part of an initialization routine.

ASan Memory Check. To enforce the ASan policy, every memory access needs to be checked to ensure they point to valid, allocated memory. Since the number of memory accesses in a program is typically much larger than the number of allocations, the memory check instrumentation needs to be highly optimized to reduce overhead. As ASan keeps track of legal bytes using shadow memory, its memory check compiles to a couple of bitwise operations, a single memory access, and two comparisons. Essentially, the instrumentation checks if the shadow byte corresponding to the memory accessed is poisoned. Failing the check implies the memory access is illegal, thereby terminating the application with a backtrace to help debug the memory error.

3.2 Design

Our goal in `RetroWrite` is to implement a version of the memory checker that closely resembles and integrates seamlessly with the source-based sanitizer. We want to work 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 re-using existing compiler based instrumentation where source code is available. In short, we do not want to implement an all or nothing solution. The main difficulty in porting source ASan directly to binaries, even with a rewriting framework, is the

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

(a)

```
lea 0x40000(%rip), %rbx
lea 0x40100(%rip), %r15
.loop:
# loop body
addq 0x10, %rbx
cmp %rbx, %r15
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 3.1.: 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.

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 impacts 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, rather than replicating ASan, we trade some precision to scale to real-world software. Qualitatively, this is how our implementation of BASan compares to ASan across memory regions:

Heap. On the heap, BASan is equivalent to ASan. Linking against the ASan runtime library ensures that calls to the memory allocator such as `malloc` are intercepted by ASan's implementation of malloc, which inserts a redzone around allocated memory objects before returning memory to the application. This means that our implementation utilizes the same mechanism as ASan to redzone objects on the heap.

Stack. BASan treats the entire stack-frame of a function as one giant struct, i.e., we redzone at a stack-frame granularity. This may miss rare bugs where the overflow is contained within the stack-frame that a ASan would catch. However, such bugs are rare in real-world applications, and therefore we do not focus on improving precision beyond stack-frame granularity. This limitation is fundamental to binary-only tools: recovering accurate stack variable information, i.e., location and size, is well studied in decompilation literature [47–50]. Popular reverse-engineering tools,such as IDA Pro, identify stack variables using architecture specific heuristics.

Our use-case for variable information is different from existing work — decompilation can tolerate misidentified stack objects as they are geared towards human readability, while BASan cannot as this results in false positives. We note that it is infeasible for any static analysis recover stack objects with high precision while being sound as this translates to solving the halting problem, e.g., accessing a stack buffer sequentially within a loop whose invariant is not known. As determining the number of loop iterations statically is undecidable, we cannot infer the size of this buffer. In short, we can tolerate loss of precision but not soundness. To support real-world use cases, such as fuzzing, where we cannot tolerate false positives, we trade-off fine-grained precision on the stack for soundness. In theory, we could eliminate this limitation by relying on auxiliary methods, such as using debug information or information reverse engineered by an analyst, to identify and redzone stack objects at a finer granularity. This can be implemented in our current framework as a pre-instrumentation analysis pass to recover stack variable information.

Global. BASan does not redzone globals for a couple of reasons. In general, 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 3.1 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 reassembly generated 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 be sure 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 that separate them. One for the end of the first object, the other for the beginning of the second object. These two labels are collapsed into a single indistinguishable offset.

Unfortunately, this is a common compiler optimization. Any binary rewriting tool that needs to modify the data layout needs to 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 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 [51]. Alternatively, this information can be recovered through heuristics, but would hurt BASan's soundness and introduce false positives. Therefore, we leave such efforts to future work. We acknowledge this limitation, but 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, BASan may miss a fixed number of overflows between global objects. Despite these limitations BASan outperforms Valgrind.

To reduce overhead and amount of instrumentation, BASan does not redzone stack frames for every function. During compilation, the compiler performs a conservative analysis to identify functions that 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`).

3.3 Binary Address Sanitizer Implementation

BASan 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 BASan instrumented executable. Note that this allows interaction with other code that may already have been instrumented with ASan. Our experience in implementing BASan in `RetroWrite` suggests that other source-based sanitizers can be ported to support binaries with minimal engineering effort.

4. BINARY FUZZING INSTRUMENTATION

Fuzzing is one of the most effective forms of software testing. 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. Due to its simplicity, generality, and ease of parallelization, fuzzing is widely adopted as a part of software testing pipelines, and has been instrumental in uncovering several important bugs. However, blackbox fuzzing, i.e., fuzzing without any knowledge about the application under test, 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 decide future sequence of inputs to the fuzz target. As program tracing can be expensive, most fuzzers trade accuracy for a more coarse-grained coverage, either at basic block or edge granularity.

AFL AFL is one of the most popular fuzzers in both academia and industry due to its ease of use and effectiveness in finding real-world bugs. As collecting and analyzing full program traces can be expensive, AFL takes a more practical approach by tracking edge-coverage as an approximation of a program trace. AFL maintains a bitmap (64KB by default, configurable) 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 blackbox binaries, AFL resorts to QEMU to instrument binaries dynamically at runtime. However, this has a significant overhead ($\tilde{1}$0x) when compared 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 finding capabilities.

4.1 Binary AFL

Implementing coverage instrumentation in our rewriting framework requires CFG recovery and instrumenting basic block starts with coverage instrumentation — calculating of edge index and updating the bitmap state. This CFG is implicitly recovered as a part of the 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 that we generate closely resembles compiler generated assembly files, `afl-gcc` works out-of-the-box to generate AFL instrumented binaries, with no additional effort. This showcases a useful feature of our framework: the reassembly we produce is compatible with existing tools that operate on assembly, and extends the tools' capabilities to support binary-only applications.

5. EVALUATION

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

RQ1: 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.

RQ2: Are we competitive to source based memory corruption detectors, such as Address Sanitizer, in terms of: (a) *runtime overhead*, and (b) *coverage*, i.e., bug detection rate.

RQ3: 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 evaluations:

1. Performance evaluation on SPEC CPU2006 comparing: baseline benchmarks (no instrumentation), ASan, BASan, and Valgrind (the most popular off-the-shelf binary-only memory checker);

2. Comparative security evaluation of the above targets on the Juliet testsuite on CWEs related to memory corruption; and

3. 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.

Hardware and Environment All our evaluations were performed on a desktop equipped with `Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz` processor and `32 GiB` of memory running `Ubuntu 16.04`.

Rewriter Overview To show scalability of `RetroWrite`, a list of all the binaries successfully rewritten as a part of the evaluation and their sizes is shown in Table 5.1. These binaries are substantially larger that binaries evaluated by previous work on reassembleable assembly.

5.1 Memory Checker — Performance

We evaluate the performance of our BASan on the SPEC CPU2006 C benchmarks. Since the original benchmarks have memory safety violations in some of the benchmarks, we applied patches provided in Google's Address Sanitizer repository [52] to enable execution with ASAN. All code was compiled with default options: `-O2 -std=gnu89` and `gcc-5.4.0`. Additionally, we added flags to produce position-independent executables. Valgrind was configured to track the same set of features as ASan.

The source code patch for ASan blacklists certain functions in perlbench, e.g., `char *move_no_asan`, from being instrumented by the sanitizer as they cause violations. Our initial evaluation of binary-only ASan reported a use-after-free in the above function, after which we manually removed ASan checks from the same function for evaluation.

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 5.1. Though our BASan is significantly better than Valgrind memcheck, it is still 0.65x slower than ASan. One of the main reasons 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 bench-

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

Table 5.1.: Overview of binaries rewritten by `RetroWrite`.

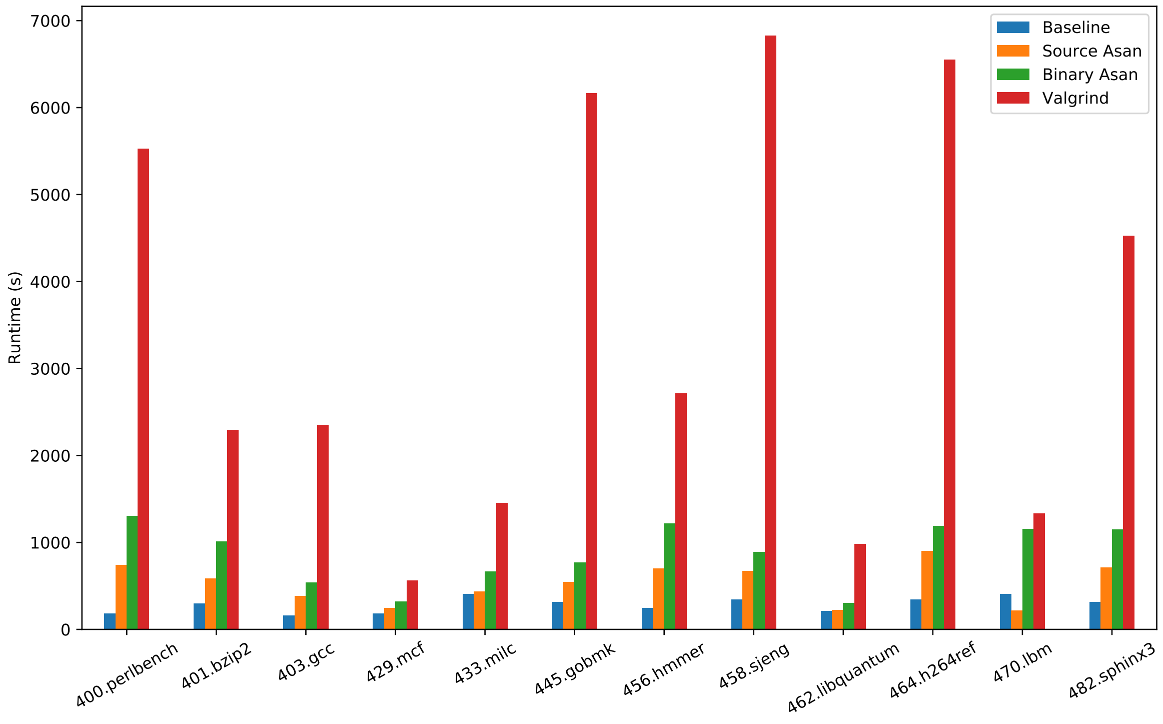

Figure 5.1.: Evaluation on SPEC CPU2006 C Benchmarks. Mean Runtime (in s) v/s Benchmark, comparing: baseline (no instrumentation), ASan, BASan (our implementation), and Valgrind memcheck (state-of-the-art binary-only memory checker). Lower numbers indicate better performance.

marks. We identified the following as causes for BASan overhead when compared to ASan:

1. **Instrumentation Locations.** Our binary ASan instruments more locations than source based ASan. The compiler-based instrumentation removes some checks if it can prove accesses are safe.

2. **Register Spills.** As register allocation happens in later phases of code generation, the compiler can take the instrumentation into account when allocating registers, thereby generating better register allocation schemes with lesser spills. Our BASan is limited to a conservative register liveness analysis and opportunistically use dead registers to reduce register spills, but cannot change the register allocation scheme as a whole.

3. **Optimal Placement of Checks.** To reduce register pressure, or flag recomputations, source based ASan is free to move the memory check instrumentation to any program point before the memory access (flexible). 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 a 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.

5.2 Memory Checker — Coverage

We compare our implementation of BASan against ASan and Valgrind on 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

Table 5.2.: Overview of Bug Detection Rate on Juliet on CWEs related to memory corruption: CWE121, CWE122, CWE124, CWE126, and CWE127. 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.

	Source ASAN	Binary ASAN	Valgrind Memcheck
Total	11828	11828	11828
True Positive	4489	3257	1785
True Negative	5914	5912	5914
False Positive	0	2	0
False Negative	1382	2614	4086
Timeout Vuln	43	43	43
Timeout Safe	0	0	0

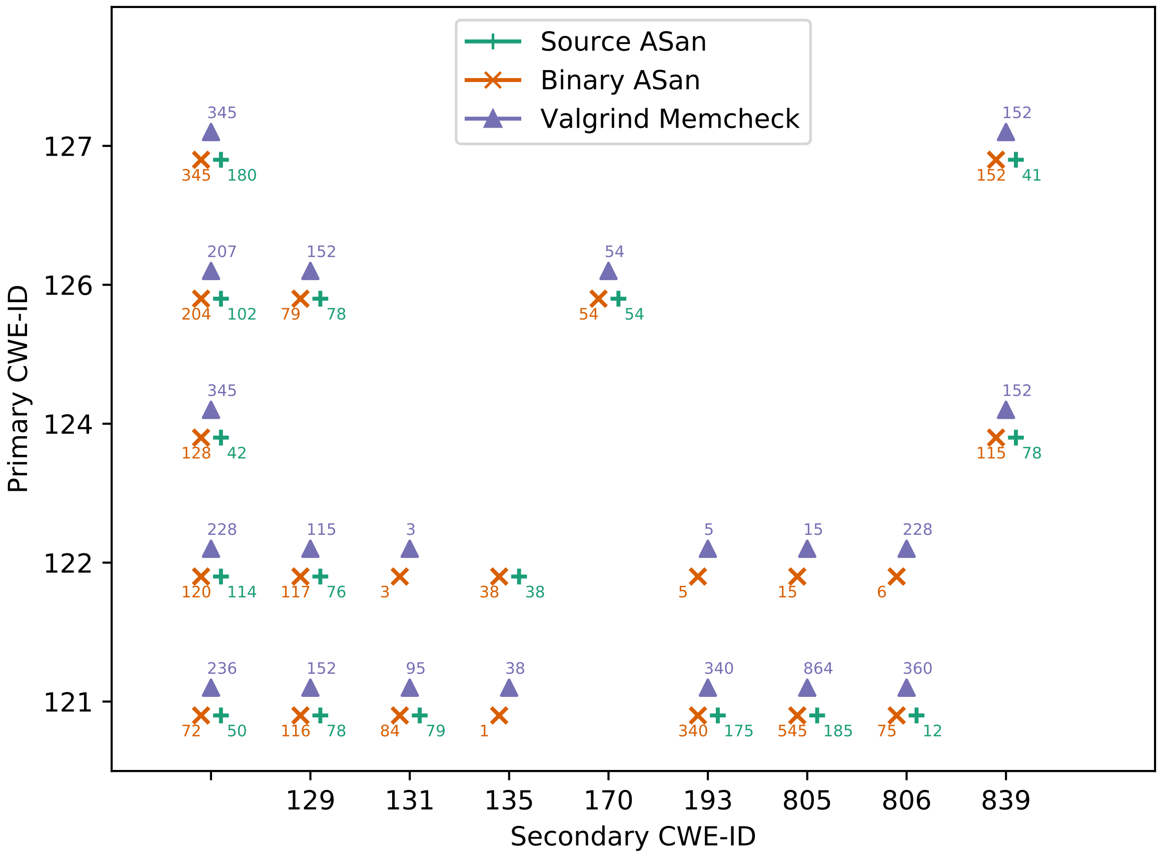

Figure 5.2.: Clusters comparing false negatives on Juliet for the three systems: Source ASan, Binary ASan, and Valgrind memcheck. X and Y axes are categorical and represent CWEs. Numbers next to points denote the number of FN of system that belong the cluster. Explanation of CWEs are provided in Table 5.3

Table 5.3.: 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

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. Similarly, a *false 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, binary-ASan (our solution), 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 5.2. 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 (looking through the two false positive cases for binary ASan manually, both the cases are due to a segmentation fault when calling `strcpy` as a part of the test case and not an actual violation report). In terms of detecting vulnerable cases, ASan has the highest detection rate of 4,489/5,914, followed by BASan 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. However, binary ASan 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 5.2 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 [53], is summarized in Table 5.3. The figure agrees with the general trend in accuracy of the three systems: source ASan, followed by binary ASan, and finally Valgrind memcheck. However, it is interesting to notice the differences, e.g., both source and binary ASan 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` (`strcpy` equivalent for wide-character strings), which is not intercepted by the ASan runtime library and hence is not instrumented. The crux of the failing test is shown in Figure 5.3. Similarly, both the binary-only tools miss 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), this difference can probably be attributed to due to more accurate object size information available in source allowing source ASan to perform more accurate checks.

On the heap (CWE122), BASan has more false negatives than ASan, even though the redzone policy is identical. This is because BASan 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 BASan 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 that we leave for future work.

5.3 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:

CF: Source code instrumentation at LLVM-IR level, through `afl-clang-fast`,

G: Source code instrumentation at assembly level, through `afl-gcc`,

```
// Allocate source and destination buffers
wchar_t* src = (wchar_t*) malloc(32 * sizeof(wchar_t));
wchar_t* dst = (wchar_t*) malloc(16 * sizeof(wchar_t));

wmemset(src, L'A', 31);
src[31] = L'\0';

// Overflow!
(void) wcscpy(dst, src)

// Not reported by ASan, caught by Valgrind memcheck!
```

Figure 5.3.: Code snippet showing failing bad-case for ASan. ASan fails to report the above overflow as wcscpy is not intercepted while Valgrind memcheck successfully detects this overflow.

Table 5.4.: 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]

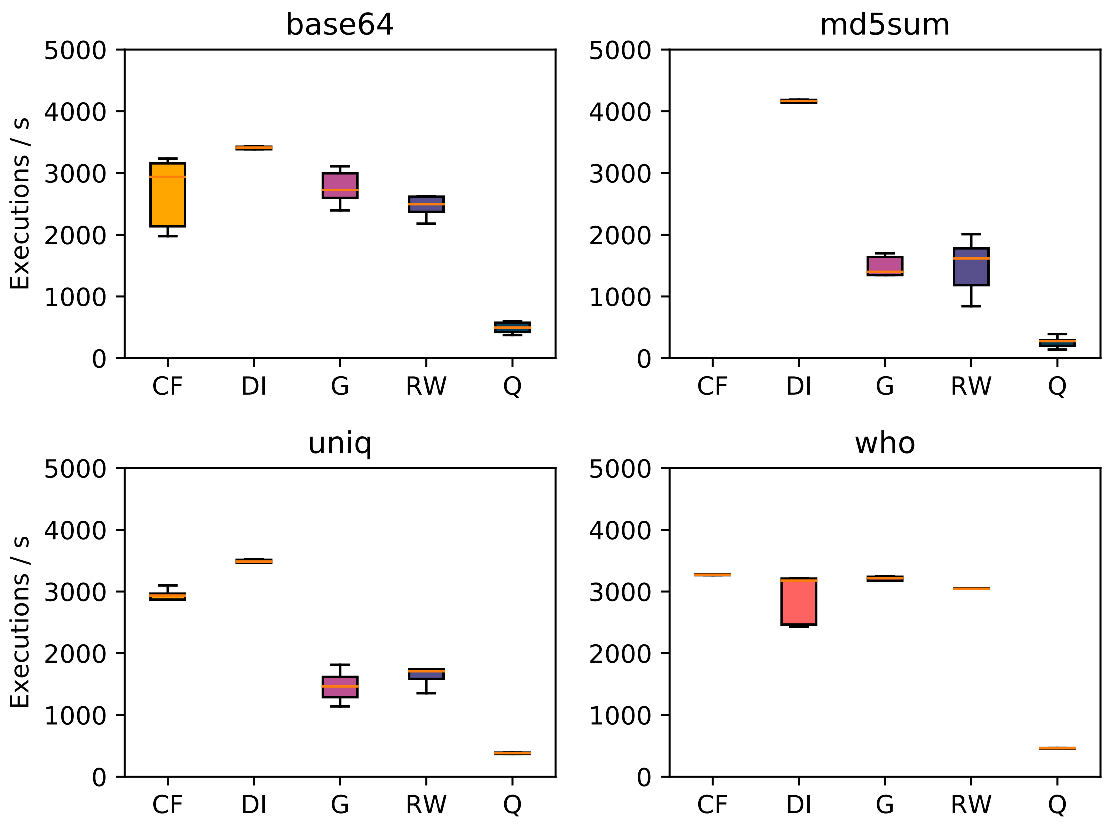

Figure 5.4.: Box plot of fuzzing executions per second on LAVA-M across five, 24 hour trials. Legend: CF=afl-clang-fast, DI=afl-dyninst, G=afl-gcc, RW=afl-retrowrite (our solution), Q=QEMU. Higher numbers indicate better performance.

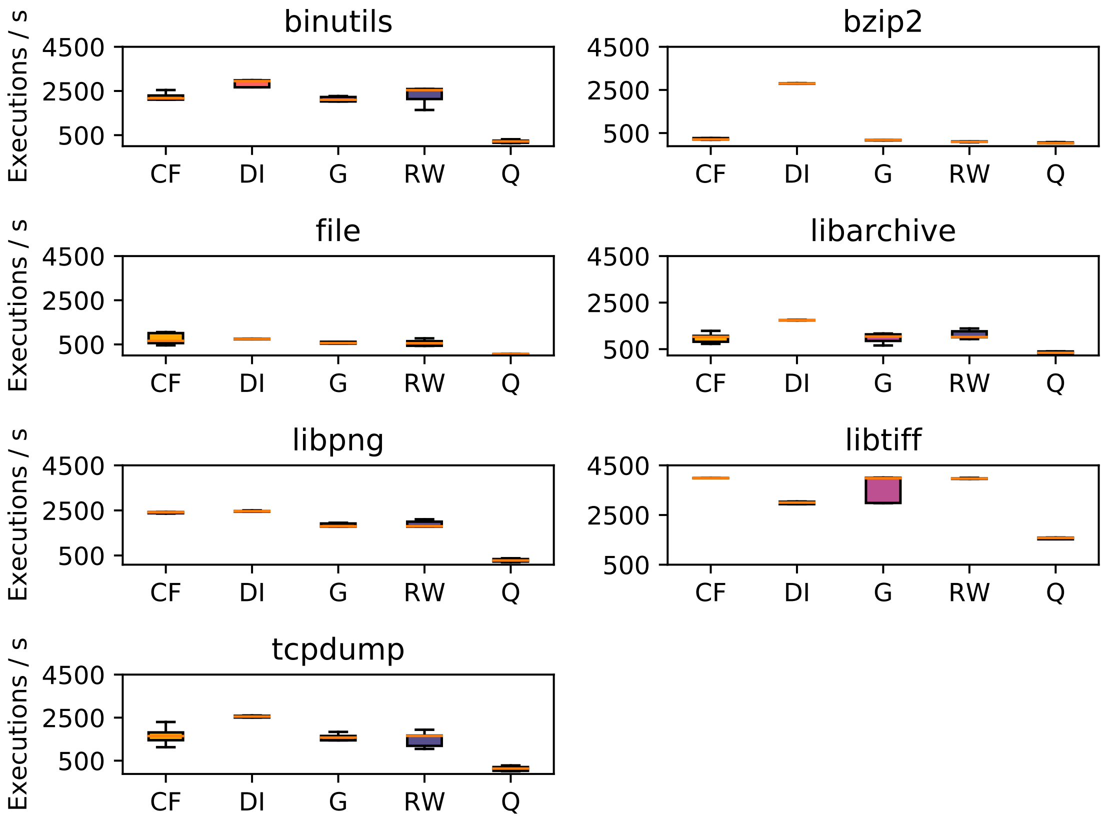

Figure 5.5.: Box plot of fuzzing executions per second on real-world targets across five, 24 hour trials. Legend: CF=afl-clang-fast, DI=afl-dyninst, G=afl-gcc, RW=afl-retrowrite (our solution), Q=QEMU. Higher numbers indicate better performance.

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. Just the fuzzer throughput 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 agaisnt 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 (`segfault`) on any input. Hence, this result for `afl-clang-fast` has been skipped.

As fuzzing is random, a single fuzz trial fails to give us a complete picture. We follow guidelines for fuzzing as presented in [54] 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 5.4 and Figure 5.5. For LAVA-M, the number of unique bugs found in each of the five trials is presented in Table 5.4.

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 thereby incurring a higher overhead when compared to the other approaches that instrument statically. `afl-dyninst` has the highest throughput among the systems that we tested, out-performing 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 bugs found indicates that this is indeed 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 roughly similar throughputs across all runs of benchmarks.

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. [54]. We compare `afl-retrowrite` v/s `afl-qemu` and `afl-gcc` in Table 5.5. All p-values comparing binary AFL and QEMU are < 0.05 indicating that the difference in performance may be statistically significant, while p-values comparing binary and source AFL are > 0.05 indicating that the differences in performance are likely due to fuzzer's randomness.

The number of bugs discovered by AFL directly shows the effectiveness of coverage instrumentation and fuzzer throughput. From Table 5.4, it is evident that the qemu based instrumentation, `afl-qemu`, only finds two bugs in a single run of uniq. The low throughput of afl-qemu prevents fuzzer from exploring a large input space and achieving high coverage leading to lower number of bugs found. In contrast, afl-dyninst has a 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 coverage instrumentation preventing afl fuzzer from exploring deeper paths and uncovering bugs. Lastly, we see that afl-retrowrite and the two source based

Table 5.5.: 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 sixth decimal place.

	RW v/s Q	RW v/s G
binutils	0.006093	0.105038
bzip2	0.010786	0.018357
file	0.006093	0.417266
libarchive	0.006093	0.265435
libpng	0.006093	0.417266
libtiff	0.006093	0.500000
tcpdump	0.006093	0.417266
base64	0.006093	0.071836
md5sum	0.006093	0.338052
uniq	0.006093	0.148135
who	0.006093	0.047346

solutions, afl-clang-fast and afl-gcc, are roughly similar in the number of unique bugs found 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.

6. CONCLUSION

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 [55] 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 the three approaches. Theoretically, we do support C++ binaries that do not make use of exception handling. However, this but has not been tested extensively. We leave the engineering work of adding support for exception handling as future work.

Closing the Performance Gap Although BASan is significantly faster than Valgrind memcheck, there is still a slowdown when compared to ASan. Our current memcheck 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 memcheck instrumentation may not be the most optimized version possible. Another area of focus 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 BASan instruments about 50% - 70% more sites than ASan.

Limitations of BASan The limitations of BASan 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.

Hybrid Approach Current DBT based approaches translate all code during execution, i.e., the main executable and all its dependencies. Such approaches can be combined with `RetroWrite` to improve both generality and performance — the main binary that is not position-independent may be instrumented using DBT while the shared libraries could be instrumented using `RetroWrite` thereby reducing the overall overhead.

Obfuscation To protect their intellectual property, many vendors ship obfuscated binaries. Our framework does not address obfuscation, and assumes that no such techniques have been employed on binaries presented for analysis. Unpacking binaries is usually dependent of the obfuscation scheme used. Such obfuscated binaries may be rewritten by `RetroWrite` after pre-processing by a deobfuscation step. Similar to previous work in binary analysis we do not consider adversarial binaries, i.e., binaries crafted to defeat our rewriting efforts, to be in-scope of this work.

6.1 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.

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 [56–58] to even transformational fuzzing [59].

Binary Rewriting Several approaches to binary rewriting have been proposed, these can be broadly divided in to two categories based on time of instrumentation: (a) Dynamic Binary Translation (DBT) based approaches [10,15,20–22] that instrument binary at runtime, and (b) Static Binary Rewriting based approaches [12–16,23,24] 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.

Reassembable 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. symbolizing a binary is undecidable in general as it requires analysis to distinguish between scalars are references statically, which has been shown to be undecidable [35]. Our work is inspired by these existing approaches, but we *trade-off generality for soundness* and target position-independent code. Consequently, though we are limited to position-independent code, we ensure that `RetroWrite` is correct by construction, not requiring any heuristics. Furthermore, our approaches are scalable allowing us to rewrite larger, real-world applications. Lastly, both Uroboros and `ramblr` were focussed on rewriting x86 32-bit binaries (and had symbolization false negatives on 64-bit binaries) while `RetroWrite` focusses on x86-64 bit position-independent binaries.

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. [25]. As pointed out by Andriesse et al. [31], 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 [26, 27] to handle other edge-cases. Control flow recovery has been the topic of discussion in [60–64]. Control flow recovery improves disassembly coverage and vice-versa, and therefore is implemented as tightly coupled passes in `angr` [29]. `RetroWrite` does not require precise recovery of the CFG as indirect calls and jumps are symbolized at 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 and therefore these works are of interest to us.

Function Identification Modern approaches [41–44] use machine learning to detect function start and sizes. Tools such as IDA [26] and radare [27] 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.

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 [47–50]. 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 BASan 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.

6.2 Conclusion

To summarize, we develop `RetroWrite`, a principled, zero-cost rewriter for position-independent code. To aid in fuzzing blackbox binaries, we develop two instrumentation passes in `RetroWrite`: (i) BASan, a binary-only memory checker, based on and compatible with ASan, and (ii) binary-AFL to collect coverage for greybox fuzzing. We showed that BASan is significantly better than the current state-of-the-art binary-only memory checker, Valgrind memcheck, both in terms of performance and coverage. Additionally, BASan is compatible with ASan, thereby allowing users to selectively rewrite closed-source parts of code with BASan while still compiling the rest of the code with ASan For coverage-guided fuzzing, we showed that 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.

REFERENCES

REFERENCES

[1] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits."

[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, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *SEC '98*, 1998.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS '05*, 2005.

[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, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 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 *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[11] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *Security and Privacy (SP), 2016 IEEE Symposium on.* IEEE, 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," 2017.

[13] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling." in *USENIX Security Symposium*, 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 *IFIP International Information Security Conference.* 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.* ACM, 2011, pp. 9–16.

[16] Z. Deng, X. Zhang, and D. Xu, "Bistro: Binary component extraction and embedding for software security applications," in *European Symposium on Research in Computer Security.* Springer, 2013, pp. 200–218.

[17] M. Zalewski, "The bug-o-rama trophy case," 2017, [Online; accessed 1-December-2018]. [Online]. Available: http://lcamtuf.coredump.cx/afl/#bugs

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

[19] 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.* IEEE Computer Society, 2011, pp. 213–223.

[20] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 81–92.

[21] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

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

[23] 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 *Reverse Engineering (WCRE), 2013 20th Working Conference on.* IEEE, 2013, pp. 52–61.

[24] E. Bauman, Z. Lin, K. W. Hamlen, A. M. Mustafa, G. Ayoade, K. Al-Naami, L. Khan, K. W. Hamlen, B. M. Thuraisingham, F. Araujo *et al.*, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proceedings of the 25th Network and Distributed Systems Security Symposium (NDSS)*, vol. 12. Springer, pp. 40–47.

[25] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Reverse engineering, 2002. Proceedings. Ninth working conference on.* IEEE, 2002, pp. 45–54.

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

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

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

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[30] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases.* Springer, 2011, pp. 522–536.

[31] 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: USENIX Association, 2016, pp. 583–600. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

[32] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275.

[33] B. Derek, G. AI, A.-J. Chris, G. E. Edmund, and Z. Kevin, "Building dynammic tools with dynamorio on x86 and armv8," 2018, [Online; accessed 28-Feb-2018].

[34] E. Bauman, Z. Lin, and K. Hamlen, "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, 2018.

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

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

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

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

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

[40] "iOS Building PIE," https://developer.apple.com/library/archive/qa/qa1788/_index.html, accessed: 2018-11-24.

[41] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code." in *AAAI*, 2008, pp. 798–804.

[42] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks." in *USENIX Security Symposium*, 2015, pp. 611–626.

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

[44] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*. IEEE, 2017, pp. 177–189.

[45] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[46] ——, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

[47] G. Balakrishnan and T. Reps, "Divine: Discovering variables in executables," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2007, pp. 1–28.

[48] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University, 2010, p. 5.

[49] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[50] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 27–41.

[51] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[52] "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

[53] "MITRE CWE Database," https://cwe.mitre.org/, accessed: 2018-11-24.

[54] 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*. ACM, 2018, pp. 2123–2138.

[55] "Dwarf standard specification," [Online; accessed 1-December-2018]. [Online]. Available: http://dwarfstd.org/Dwarf5Std.php

[56] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, ser. CCS, 2017, pp. 1–16.

[57] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," 2018, pp. 1–18.

[58] 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.

[59] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2018-May, 2018, pp. 697–710.

[60] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," in *Program Comprehension, 1999. Proceedings. Seventh International Workshop on.* IEEE, 1999, pp. 192–199.

[61] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.

[62] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *International Conference on Computer Aided Verification.* Springer, 2008, pp. 423–427.

[63] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.

[64] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications." in *USENIX Security Symposium*, 2014, pp. 829–844.

[65] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*, 2012, p. 299.

[66] E. Athanasopoulos, "Control Flow Integrity for COTS Binaries - draft2," in *Usenix Security*, no. 2013, 2013, pp. 1–5.

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

[68] "Mcsema," https://www.trailofbits.com/research-and-development/mcsema, accessed: 2018-11-24.

[69] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," 2009.

[70] A. Sepp, B. Mihaila, and A. Simon, "Precise static analysis of binaries by extracting relational information," in *Reverse Engineering (WCRE), 2011 18th Working Conference on.* IEEE, 2011, pp. 357–366.

[71] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[72] "Cve-2012-0809," http://seclists.org/fulldisclosure/2012/Jan/att-590/advisory_sudo.txt, accessed: 2018-02-26.

[73] "Cve-2014-2299," https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=9843, accessed: 2018-02-26.

[74] "Cve-2014-0195," https://www.rapid7.com/db/modules/auxiliary/dos/ssl/dtls_fragment_overflow, accessed: 2018-02-26.

[75] "Cve-2013-2028," http://mailman.nginx.org/pipermail/nginx-announce/2013/000112.html, accessed: 2018-02-27.

[76] "Nginx test suite," https://github.com/nginx/nginx-tests, accessed: 2018-02-37.

[77] "Qemu tcg," https://wiki.qemu.org/Documentation/TCG, accessed: 2018-02-26.

[78] A. Di Federico, M. Payer, and G. Agosta, "Rev.ng: A unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. New York, NY, USA: ACM, 2017, pp. 131–141. [Online]. Available: http://doi.acm.org/10.1145/3033019.3033028

[79] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari, "Analyzing dynamic binary instrumentation overhead," *Workshop on Binary Instrumentation and Application, October 2006, San Jose, CA.*, 2006.

[80] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "Marx: Uncovering class hierarchies in c++ programs," in *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*, 2017.

[81] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict virtual call integrity checking for c++ binaries," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.* ACM, 2017, pp. 140–154.

[82] G. Ramalingam, J. Field, and F. Tip, "Aggregate structure identification and its application to program analysis," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1999, pp. 119–132.

[83] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.

[84] 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 *USENIX Security Symposium*, 2016, pp. 583–600.

[85] J. Kinder, "Static analysis of x86 executables," Ph.D. dissertation, Technische Universitat Darmstadt, 2010.

[86] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of win32/intel executables using etch," in *Proceedings of the USENIX Windows NT Workshop*, vol. 1997, 1997, pp. 1–8.