ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ArmWrestling: efficient binary rewriting for ARM

Master Thesis

Luca Di Bartolomeo

January 15, 2021

Advisor: Prof. Dr. Mathias Payer
Supervisor: Prof. Dr. Kenny Paterson
Institute of Information Security
Department of Computer Science, ETH Zürich

This thesis was developed with the guidance of prof.Mathias Payer and the HexHive research group at EPFL, and under the remote supervision of prof. Kenneth Paterson at ETH.

No matter where you go, everyone is connected.
— Serial Experiments Lain

Dedicated to my parents, my brothers Sara and Leo, my dear Giulia, to my friends back in Rome and to my roommates Matteo and Filippo who all inspired me and kept up with my constant complaining. Thanks!

Acknowledgments

I would like to thank my advisor, Prof. Mathias Payer, for his support, guidance, and for assigning me this inspiring project. I wish all the best for him, and I look forward to continue working with him in the future.

I would like to thank as well the HexHive research group, as I always found myself very welcome there, even if I could visit them only once a week.

The last six months have been inspiring and pleasant: I had very good conversations with everyone at EPFL and I always felt like I was learning a lot.

Special thanks go to my family and my friends in Rome. Their support was always available even when remote and it has been a huge pleasure to visit them once in a while in Italy. I am also need to thank my S.O. Giulia, I felt she was always behind my back, keeping a good check on my mental sanity during the worst times of the outbreak. My roommates Matteo and Filippo deserve a mention here as well, since their patience and their rubber duck debugging skills proved to be fundamental during some nasty bug-hunting sessions.

Finally, I would also like to mention my CTF team, Flagbot, and the teams I had played with occasionally, namely Polyglots and TRX; they made me spend so many weekends staying at home but ultimately let me meet many new interesting people. Thanks!

Abstract

Closed source programs are particularly hard to audit for vulnerabilities. Moreover, it is often the case that modern security measures and mitigations are available only as compiler passes that require possession of the source code. Even if there were good recent attempts [24] [30] [25] at completely avoiding usage or closed source libraries of modules that run at privileges higher than we might want (e.g., manufacturer specific kernel modules [16]) in practice it is almost impossible to restrict our computing to exclusively open source and audited software. It is then of paramount importance that we find new ways of securing software without source code.

Many existing tools were developed to improve the auditability of closed source programs, especially aimed at helping the fuzzing process, with approaches such as implementing AddressSanitizer (a compiler pass only applicable when the source code is available) through dynamic instrumentation. However, even state-of-the-art dynamic instrumentation engines incur in prohibitive runtime overhead (between 3x and 10x and more). Static rewriters introduce less overhead, but they are mostly targeted towards the `x86` architecture.

We would like to show that symbolization for `ARM` binaries is a viable alternative to existing static approaches, that has less flexibility (only works on C, position independent binaries) but has negligible overhead. We present RetroWrite-ARM, a *zero-overhead* static binary rewriter for `aarch64` executables that solves key challenges such as pointer construction symbolization and jump table instrumentation, based on the (`x86_64`) RetroWrite project.

Our proof of work implementation of a memory sanitizer instrumentation pass has the same core functionality of AddressSanitizer, and our benchmarks show that it is an order of magnitude faster than Valgrind's `memcheck`.

Contents

Chapter 1

Introduction

Mobile environments are ubiquitous but often lack security updates. For example, on Android over 60% of devices are outdated by at least two years [1]. Security testing in mobile environments is extremely challenging as vendors often release components as binary-only, i.e., without accompanying source code. Binaries cannot be readily evaluated by security analysts, resulting in potentially undiscovered vulnerabilities in these binary-only modules, which often run with high privileges and are exposed to the network. Binary rewriting is a process that allows insertion of arbitrary code (instrumentation) in an executable without the need to recover its source code, allowing the insertion of hardening measures that can substantially reduce the exploitability in the case of a vulnerability. Binary rewriting is also vital to analysts looking for new vulnerabilities, as it's an important step in the fuzzing of closed source software.

Binary rewriters can be split into two main categories: *dynamic* rewriters , that inject code into the target executable at run-time, and *static* ones , that transform the executable into a new one with the instrumentation already inserted. The first kind allow more flexibility, and support a broader range of binaries, but they insert noticeable overhead during its execution; the static ones instead have a much smaller footprint in execution time but are

only applicable on binaries on which static analysis (the analysis without runtime information) is successful.

While there is no shortage of *static* rewriters targeted towards the x86 architecture, only a few support ARM. In fact, ARM binaries require a very different kind of static analysis than x86, and present many challenges that are not found on x86. Those challenges derive from the fact that the ARM ISA (Instruction Set Architecture) is fixed-size, and all instructions occupy exactly 4 bytes. Since addresses in the 64 bit space are 8-byte long, they do not fit into a single instruction, and executables must "craft" addresses in various ways (storing them in a data section at compile time, or using arithmetic expressions to construct them). Many static analysis techniques commonly used on x86 are not usable anymore on ARM for this reason. There are a number of *dynamic* rewriters that support ARM, and they work by running together with the binary to instrument and modifying it in real time: this allows them to use powerful dynamic analysis techniques to recover more information, but as a side effect they introduce large overhead even with very light instrumentation.

Introducing only a small overhead onto an instrumented binary is a very desirable feature to have, as nowadays the largest use case of binary instrumentation is the injection of fuzzing hooks and memory sanitization checks in a closed source binary to drastically improve the efficiency of modern fuzzing engines. Using a static binary rewriter versus a dynamic one when fuzzing can lead to an order of magnitude more executions per second.

Fuzzing on the ARM architecture has had little growth compared to x86, and one of the main causes is that the instrumentation passes required for fuzzing are supported only by heavy dynamic binary rewriters, which lead to low fuzzing performances due to their overhead. The fact that ARM became popular only very recently, combined with the difficulty of perform-

ing static analysis on it, are the reasons behind the lack of advanced static binary rewriters for ARM. In fact, most of the existing ones rely on simpler techniques (like *trampolines*) that require only superficial analysis, but are not flexible nor efficient.

Static analysis is the core foundation on many state-of-the-art static binary rewriters: one of the most recent and successful ones for x86_64 is RetroWrite [11]. It uses a novel technique called *symbolization*, which is the process of transforming an executable back to an assembly listing that is *re-assemblable*, where absolute addresses are substituted with assembly labels (or *symbols*). Having all addresses substituted with labels makes adding custom code (instrumentation) or arbitrary transformation on existing code very easy, as the resulting assembly can be parsed by a generic off-the-shelf assembler and assembled into a new instrumented executable.

In this thesis we would like to introduce RetroWrite-ARM, a new, advanced static rewriter for ARM64 binaries that builds on the original RetroWrite (from now on referred as RetroWrite-x64) and uses the same *symbolization* technique to allow the development of complex and efficient instrumentation for closed-source binaries. Our work focuses on the development of novel approaches to tackle the static analysis challenges presented by ARM executables such as pointer construction symbolization and jump table enlargement. We also implement an example instrumentation pass (implementing the same algorithm of Google's *AddressSanitizer*, an LLVM compiler pass to sanitize memory accesses) to show the ease and efficiency of writing instrumentation using the symbolization technique, we then measure its performance to demonstrate that the execution time is comparable to the original LLVM pass (that requires the source code of the binary to instrument, so impossible to apply on binary-only modules.)

The main challenges we had to tackle were the detection and recovery of

pointers, that are either stored in a read-only region or more commonly are built through a series of arithmetic operations because a single pointer cannot fit into a single `ARM` instruction. This required reverse-engineering the patterns of code the compiler generates and clever ways to transform said arithmetic sequences of instructions into symbolized assembly snippets that can support the insertion of arbitrary code in between. Another big part of our work focused on the detection, recovery and symbolization of jump tables: switch statements on `ARM` binaries are very different from `x86` (they are stored as a set of offsets from a given base address, not as a sequential list of absolute addresses). All the standard heuristics methods that are commonly used on `x86` cannot be used on `ARM` as the jump tables are indistinguishable from random data, so this required the development of a bare-bones symbolic emulator to detect them. Furthermore, since they are stored *compressed* in memory (if cases are close to each other, every offset may occupy only 1 or 2 bytes instead of the normal 8 bytes), they do not support the insertion of large instrumentation. We solved this problem with a new technique called *jump table enlargement* which lets us introduced arbitrary-sized instrumentation inside a jump table.

We will show that our binary rewriter incurs in very low overhead through standard CPU benchmarks commonly used to evaluate binary rewriters. We will also show that our implementation scales well to COTS binaries, as the benchmarks include large programs such as `gcc` and `perl`. Finally, we also evaluate the performance of our implementation of AddressSanitizer, showing that it is competitive with its source-based counterpart, adding only 19% additional overhead on top of it (depending on the binary and hardware configuration of the machine).

Our core contribution consists in the development of the first zero-overhead `aarch64` static binary symbolizer that scales to large COTS binaries, with key insights in detecting and symbolizing pointer construction mechanisms and

compressed jump tables, and an efficient instrumentation pass that retrofits `aarch64` binaries with AddressSanitizer.

Chapter 2

Background

Here we will provide a summary of the basic notions required to understand the underlying concepts behind RetroWrite-ARM and the problems we faced during its development.

2.1 Binary Rewriting

Binary rewriting describes the alteration of a compiled program in such a way that the binary under investigation remains executable. It is useful in the case some additional functionality needs to be added to a closed-source binary, or to perform live changes on an already running executable. At first, binary rewriting was considered a "hack" to change parts of a program during execution (run-time patching, with the first uses dating back to the 60s on the PDP-11 [22]) but today it evolved into a stable and researched field of computer science, with a number of applications ranging from security to software optimization.

In the next subsection we will briefly explain how it works, and then we will present the most common uses of binary rewriting.

2.1.1 Binary rewriting in short

To better explain how does a rewriting algorithm work, we can summarize its process into four main steps [33]:

- *Parsing*: Executables are complex formats and are often optimized for speed instead of readability. Separating code, data and metadata in an executable format is not easy: often all three are scattered through the file in different ways. Furthermore, data is usually *untyped* (there is no type information on global variables), and the boundary between code and data is not always clearly defined as in many architectures instructions have variable width. Finally, recovering and distinguishing references (pointers) from scalars (constants) is a problem which was only recently solved on position-independent executables [11]. The purpose of this step is to parse the input file and separate information between code and data into clearly defined structures ready to be analyzed in the next step.

- *Analysis*: This step focuses on the code of the executable, and on the recovery of as much information as possible from the raw instructions present in the code sections of the executable. Usually this involves the disassembly of the raw instructions to be able to analyze them individually, the study of the structure of the code to split it into functions, and the analysis of the branches of each function to build the control flow graph (CFG). Often times this step also involves the computation of all cross-references (list of pointers to a specific address). At the end of this step the rewriter should have a very precise representation of the executable in a convenient data structure that allows fast answers to queries such as listing all the call sites of a specific function.

- *Transformation*: With all the information gathered from the analysis, now the rewriter must identify all the instrumentation locations in

which code has to be altered/injected. For example, in the case of a memory sanitization instrumentation pass, the code must locate all memory stores and all memory loads and mark them for insertion of snippets of code that check if the target memory region lies inside an uncorrupted heap chunk, and abort execution if the check fails.

- *Code generation*: The rewriter now has to make sure that the inserted instrumentation does not break the intended functionality of the original executable, and then patch it or generate a new executable from scratch. There are multiple ways to achieve this step, the most common ones being using trampolines (creating a section with the new code in the target binary, and then inserting branches at instrumentation locations to make the new code reachable), in-place instrumentation (generation of a new executable with the new instructions already inserted at each instrumentation locations, with all the code and data references adjusted to make place for the new code), or run-time patching.

The last two steps, "transformation" and "code generation" are where most of the research in binary rewriting is done. We will shortly list the most prominent techniques that rewriters use to perform those latest two steps:

- Trampolines: All the instrumentation code is put into a new section in the binary. At every instrumentation point, the rewriter changes an instruction with a branch that redirects execution to the right location inside the new section. At the end of the instrumentation, there is a branch that leads back to the old location in the executable. This is one of the simplest methods as the layout of the code section stays the same and no references or branches are broken inside the executable.
- Direct: At each instrumentation point, code is either overwritten or shifted to make space for the instrumentation to insert. In the case of a shift, all references and branches must be carefully readjusted to

match the displacement of the code.

- Lifting: This approach involves lifting the binary code to an Intermediate Representation (IR) language similar to the one used in compilers such as LLVM. The logic behind it is that IR usually is a simpler language on which it is easier to apply instrumentation to. At the end of the process the IR is compiled back to binary code and a new instrumented executable is generated.
- Symbolization: The process of transforming a binary into a *reassemblable* assembly listing and applying instrumentation on it. Since this is the method used by RetroWrite-ARM, we will go more detail of how it works in the next section.

A more comprehensive report of all the different techniques can be found in a recent survey about binary rewriting by Wenzl et al. [33].

2.1.2 Applications of binary rewriting

The applications of binary rewriting are multiple and can be summarized as follows:

- **Cross ISA binary translation**: A binary translator is a special software that mimics the behaviour of a device while executing on a different device. Emulators use binary rewriting to translate system calls, instructions, memory access and all the other execution primitives from one processor architecture to another. An example of this would be QEMU [3].
- **Optimization**: In the domain of high performance computing, having a way to patch subtle things like cache misses or timing anomalies in very long running tasks without the need to restart the whole program is of special interest. In such situations, binary rewriting is a solution for run-time patching, as shown by DynInst [4] or Frida [13].

- **Profiling**: Having an in-depth look during the execution of a binary by inserting profiling or tracing instructions in the middle of its code can prove to be particularly useful in many applications, like catching memory leaks (e.g., Valgrind [23]), coverage information for fuzzing (e.g., AFL-QEMU [35]) and more.
- **Hardening**: This is by far the most popular use case of binary rewriting, as many times we are forced to use software with the absence of source code, with no vendor support, or with deprecated build tools that make recompilation impossible. Binary rewriting can be used to apply security measures such as adding stack canaries, implementing address layout randomization schemes and memory sanitization to make exploitation substantially harder. Closed source software and lack of vendor support is so widespread that there are already many binary rewriting tools on `x86` that are aimed at hardening executables. Examples of such software are Stackguard [8] (that supports the insertion of stack canaries) or RetroWrite-x64 [11] (that implements a memory sanitization pass to prevent heap corruptions).

2.2 Dynamic and Static Instrumentation

In this section we will analyze the difference between the two different binary rewriting approaches, namely *dynamic* and *static* instrumentation.

2.2.1 Dynamic instrumentation

Dynamic rewriters modify and instrument the code of the target binary during runtime. Usually, the target binary is executed in a controlled environment side by side with the rewriter engine, which patches instructions and fixes references on the go. Sometimes the rewriter engine leverages the operating system's primitives to control the execution of the target, like us-

ing the `ptrace` system call on Linux, but there are notable cases in which the rewriter engine comes with its own instrumentation runtime (e.g., Dynamo [2]) or implement a full featured virtual machine (e.g., STRATA [26]).

The big advantage of dynamic rewriters is the additional information that is available at run time, like the path that the execution has taken, or the contents of the registers. Furthermore, dynamic rewriters can avoid analyzing the whole binary at once, as they can just focus on the part that is being currently executed, making them scalable to arbitrarily large programs.

However, the additional runtime information comes at a high performance cost: running the rewriter engine alongside the target binary is expensive, and the frequent context switches the CPU must perform to execute both processes make the performance even worse. The total overhead for an instrumentation pass like memory sanitization for a dynamic rewriter like Valgrind is an order of magnitude higher [11] than the overhead introduced by source-level memory sanitization.

2.2.2 Static instrumentation

Static rewriters process the target binary *before* execution, and produce as output a new binary with all the required instrumentation included. The overhead introduced is usually very low, and execution speeds are comparable to compile-time instrumentation. Furthermore, static rewriters are able to add complex instrumentation that is computationally expensive to introduce, as since it is done statically before execution, it won't introduce unnecessary delays at runtime.

Without runtime information, to correctly perform instrumentation static rewriters need to rely on complex static analysis, which is inherently imprecise and often needs to rely on heuristics. The common disadvantage of static rewriters is that they do not scale well on large binaries or binaries

that do not follow standard patterns. In fact, virtually no static rewriter supports packed binaries or self-modifying code, as they are too complex to statically analyze. Moreover, many static rewriters struggle even with binaries produced by deprecated compilers or with aggressive optimization flags.

More recent static rewriters such as Ramblr [31], Uroboros [32], and RetroWrite-x64 rely on *symbolization*, which works around the rigid structure of binaries by substituting hard coded references with assembly labels. RetroWrite-x64's approach is particularly interesting in the fact that it avoid heuristics to differentiate between scalars and references by focusing on position-independent executables (PIE).

2.3 Symbolization

The *symbolization* technique is a special form of code generation that focuses on the output of *reassemblable* assembly that can be directly fed to a generic assembler to produce the instrumented executable. Listing 2.1 highlights an example of this process .

Symbolization works by transforming all reference constants in the executable (both in the code and data sections, including relative branches) with assembly labels, in such a way that pointers and control flow will resolve correctly even after new instructions are inserted in the middle of the code. The usefulness of symbolization relies in the fact that many existing tools can be applied to insert instrumentation or analyze the symbolized assembly.

The symbolization approach is usually defined as being *zero-overhead*, as the generated executable does not incur in more overhead other than the time it takes to execute the inserted instrumentation (unlike other methods,

like trampolines, where for each instrumentation location two additional branches need to be executed).

Originally, symbolization was introduced by Uroboros [32], and was later used by ramblr [31] and RetroWrite-x64 [11].

```
0x400: adr x0, 0xab0000
0x404: cmp x1, 20
0x408: b.eq 4
0x40c: ret
0x410: ret
```

```
.LC400: adr x0, .LCab0000
.LC404: cmp x1, 20
.LC408: b.eq .LC410
.LC40c: ret
.LC410: ret
```

Listing 2.1: Assembly in the original binary (left), and after symbolization (right)

2.4 Examples of code instrumentation

In this section we will go into more detail on a very common use-case of instrumentation, *fuzzing*, and explain AdressSanitizer, the instrumentation pass we implemented in the ARM port of RetroWrite-ARM.

2.4.1 Fuzzing

Automatic vulnerability discovery techniques are getting a lot of traction lately, mostly because software is getting ever more complex and large, and manual analysis and auditing do not scale. Fuzzing is certainly one of the most interesting automatic vulnerability discovery techniques. It relies on the semi-random generation of test cases to give as input to a target binary, trying to find a specific input that makes the binary get into an invalid or undefined state, as it is a good indicator of a possibly exploitable vulnerability. This technique got even more popular after the release of `AFL` [35], a fuzzer that relies on coverage information to generate new test cases to maximize the amount of instructions tested by each new input, and `Honggfuzz` [29], a modern fuzzer used to efficiently test APIs thanks to its innovative *persistent fuzzing* feature.

Most state-of-the-art fuzzers rely on instrumentation to improve vulnerability discovery, as it makes the fuzzing process much more efficient. In particular, some of the most popular instrumentation passes used to speed up fuzzing are the following:

- *Coverage information*: Coverage information helps the fuzzing engine by monitoring the execution path taken for each input test case. In this way, the fuzzing engine can generate inputs with the aim of maximizing the amount of code executed by each test case, increasing the probability of finding bugs. Coverage information is commonly obtained by inserting monitoring instrumentation at the start of each function or before each branch instruction.
- *Memory Sanitization*: This instrumentation pass adds a check before each instruction that reads or writes to memory to verify that the memory access does not result in a stack or heap overflow. Originally developed to debug memory corruption errors, memory sanitization has seen widespread use in fuzzing engines as it can halt execution and report an error as soon as a memory corruption is detected.

2.4.2 ASan

AddressSanitizer, or ASan in short, is one of the most common static memory sanitization checks that can be added to a binary through a compiler pass, which can be found in both the `clang` and `gcc` family of compilers. This compiler pass helps finding bugs by actively checking for memory corruptions, hooking calls to libc's `free` and `malloc` functions. ASan is not only used by developers to help debug their code, but it is also extensively used by fuzzers, as ASan will detect a memory violation as soon as it happens, letting the fuzzer know earlier and with more reliability when a bug was found (otherwise, the fuzzer has to wait that the memory corruption causes a crash).

ASan works by introducing a new region of memory called *shadow memory*, with a size of exactly 1/8 of the whole virtual memory available to a process. By keeping track of each call to `malloc` and `free`, ASan stores in the shadow memory a compressed layout of the valid memory in the original virtual space, and sets up *red zones* to highlight invalid or freed memory. Those red zones trigger a security violation and abort the process as soon as they are accessed. Despite its non-negligible overhead (Around 73% on average [27]), ASan is widely used thanks to the absence of false-positives, and for its usefulness in detecting memory corruption vulnerabilities which are still commonly found in C/C++ codebases.

2.5 The ARM architecture

We will provide a short summary of what are the main differences between `x86` and `ARM`, with particular focus on the ones that proved to be source of non trivial problems during the development of RetroWrite-ARM.

- *Fixed-size instruction set*: Contrary to `x86`, the `ARM` instructions are all of the same size, fixed to the value of 4 bytes. A consequence of this is that a pointer cannot fit into a single instruction. To store a pointer in a register in `ARM`, there are two main options: the first is using a region of data where the pointer is hard-coded at compile time, called a *literal pool*; the second one is building the pointer in a multi-stage fashion by using arithmetic operations. While the first one is easier, it is also less performant, and compilers will always resort to the second when possible. This makes recovering information about global variable accesses very hard.

- *Jump table compression*: On `x86`, jump tables are stored as list of pointers in a data section (usually `.rodata`), with one pointer for each case of the jump table. Instead, on `ARM`, jump tables are stored as offsets

from the base case. This is because the compiler compresses the jump table, and in most cases a single byte is enough to store the offset from the base case to for each case of the jump table. This is the source of many problems for static rewriting: first of all jump tables are harder to detect, as on x86 scanning the data sections for arrays of valid instruction pointers was a quite reliable way of detecting jump tables, while on ARM they are indistinguishable from random bytes; secondly inserting too much instrumentation between cases of the same jump table could lead to the offset not fitting into a single byte anymore, and breaking the whole jump table structure in memory. Finally, extracting the number of cases of a jump table is quite harder in ARM, since it is impossible to scan cases until an invalid pointer is found, as like stated before, jump table entries in ARM are indistinguishable from random bytes.

- *Discontinuities in immediates*: Some ARM instructions, like "add", support having immediates as one of the operands. However, they do not accept a standard range of immediates like in x86, but instead a specific set of values that may not be continuous. For example the "add" instruction can use only immediates that can be expressed with a value of 8 bits scaled by a "ror" with a 4 bits operand.

- *Alignment issues*: The stack pointer register "sp" must always be aligned to 16 bytes. Failing to do so will trigger a SIGBUS error and crash the application.

- *No push/pop*: There are no instructions in aarch64 equivalent to the x86 push/pop. Instead, a push is performed by storing a register on the stack and manually decreasing the stack pointer, like "`str x0, [sp, #-16]!`". Similarly, a pop can be performed like this "`ldr x0, [sp], #16`".

- *Multiple register stores/loads*: The aarch64 architecture supports saving

and loading two registers at once from memory with instructions such as "`stp`" and "`ldp`". They are very often used by programmers and compilers thanks to the performance gain.

- *Peak performance vs energy efficiency*: While `x86` is aimed towards maximising performance and speed, one of the main objectives of the design of the `ARM` architecture is maximising energy efficiency. This is the reason behind the simplicity of the instruction set, as the CPU can be smaller and less complex compared to `x86` — and, ultimately, less transistors translate to less power consumed.
- *Not enough mature tools*: The popularity of `ARM` CPUs is still relatively new and the tooling is not mature enough, as in fact we found bugs in both the disassembler we chose to use (Capstone [5]) and the debugger (GDB)

Chapter 3

Design

We will now go over the design goals of RetroWrite-ARM, we will explain which were the key issues that we had to face due to the quirks of the `ARM` architecture, and the solutions we adopted to overcome these problems both in the *symbolization* and in the *instrumentation* parts of RetroWrite-ARM.

3.1 Goals

Our goal is to develop a zero-overhead binary translator for `aarch64` executables that enables powerful translation and overcomes challenges spawned by the `ARM` ISA. It should also support COTS software and scale well to large binaries. Finally, its implementation should be modular in order to avoid limiting any kind of instrumentation.

3.2 System architecture

RetroWrite-ARM follows the same structure as RetroWrite-x64 and is divided into two main components: the symbolizer and the instrumentation passes. The symbolizer takes care of parsing and analyzing a binary, substituting every reference in the target with assembly labels, plus some minor tasks to keep the original functionality of the binary intact. Listing 2.1 shows

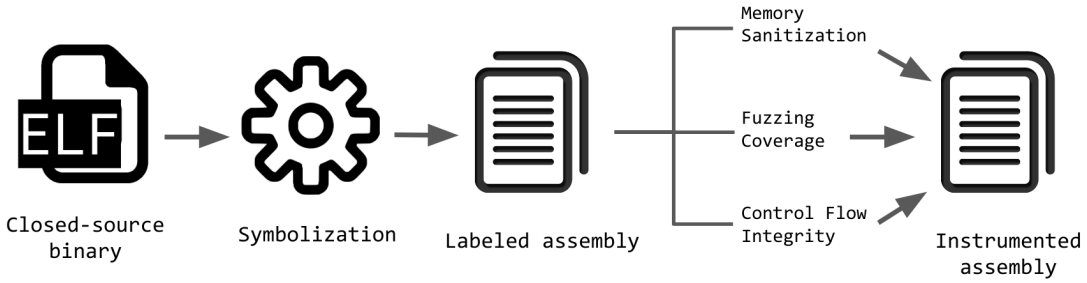

Figure 3.1: Overview of the structure of RetroWrite-ARM

the output of the symbolization process on a small example assembly snippet.

One or more instrumentation passes can be enabled to apply transformations to the resulting binary. For now, only a single pass is implemented (BASan), but many more can be easily added.

3.2.1 Differences with RetroWrite-x64

RetroWrite-ARM uses the same approach as RetroWrite-x64 for parsing the executables, although with different implementation details to support the `ARM` architecture (e.g., different handling for the relocations). The technique to distinguish between references and scalars introduced in RetroWrite-x64 is also the same.

Since this document focuses on the new challenges that the `ARM` architecture introduced, we will not go into details about the above algorithms, and point the reader to the original RetroWrite-x64 paper [11] for further reference.

The novelty in RetroWrite-ARM relies in the additional static analysis methods that we had to develop to support detection of pointer constructions and jump tables. To the best of our knowledge, our work is the first attempt to generate symbolized enlarged jump tables and symbolized pointer

constructions. In the next section we will go into detail about the above challenges and how did we solve them.

3.3 Key Issues

The outstanding challenges of statically analyzing and instrumenting ARM binaries can be summarized as follows:

- Detecting and fixing pointer constructions.
- Detecting and symbolizing jump tables.
- Supporting extensive instrumentation by enlarging jump tables.

In the following pages we will get into detail for each one of those issues, and explain the reasoning behind our solution.

3.3.1 Pointer construction

The aarch64 instruction set is defined as *fixed size*, because every instruction is large exactly 4 bytes. This makes the CPU design simpler, helps keeping memory always aligned, and permits the CPU to fetch multiple instructions at once, since decoding is not necessary to determine instruction boundaries. However, despite the many advantages of this characteristic, there are some drawbacks too, including not being able to store a pointer in a single instruction (as pointers have a size of 8 bytes). The aarch64 ISA provides two main solutions to this problem.

The first one consists in storing the pointers in a special read-only region of memory, called a *literal pool*, and then load those pointers into a registers using the special construct "ldr <reg>, =pointer", a pseudo-instruction that the assembler will translate with the correct memory address once pointer has been stored in a literal pool. Since all "ldr" instructions are PC-relative, and since the "ldr" instruction keeps 21 bits available to store the offset from

the PC, the assembler will store *pointer* in a literal pool which is in the ± 1MB range of the "`ldr`" instruction. While this is a simple and straightforward approach, very useful in the case of hand-written assembly, this requires an additional memory access that may impact performance in the long run. Furthermore, the assembly will fail if it is not possible to store a literal pool in the given range, such as in the case of a function larger than 2MB. In that case, it is up to the programmer to find a suitable spot for the literal pool, by manually specifying its location with the `.ltorg` assembly directive. It is often recommended to store literal pools directly after non-conditional jumps to avoid stepping over them during execution [20].

The second solution is to build pointers using multiple instructions and basic arithmetic primitives. `aarch64` provides instructions such as "`adrp <reg>, pointer`", which loads the base page of a pointer into a register. It is a PC-relative instruction, and targets pointers in the ± 4GB range. In other words, the "`adrp`" instruction can point only to memory locations that are 4KB aligned. Usually the instruction can be followed by an "`add`", a "`sub`" or an offset-based load such as "`ldr <register>, [<base_page>, offset]`". This second way, while more contrived and harder to read, is faster than the first one as it does not require a memory access, and also often benefits from custom hardware optimizations (such as in the Cortex A72, one of the most common `ARM` CPUs [7]).

The global variable problem

For the reasons stated above, compilers generally use pointer constructions instead of literal pools when the code needs to access a global variable, preferring performance over assembly readability. Having each pointer value separated in two different instructions makes the static analysis of a binary substantially harder. Furthermore, compiler optimizations frequently exacerbate the difficulty of analysis by reusing parts of some pointer values

to build new ones, or reordering instructions around in such a way that a pointer can be built on two instructions which are kilobytes away from each other. In some extreme cases, by enabling the `-O3` optimizations, we found instances of pointers built on two instructions that were on different functions, due to the compiler optimizing a macro in the C source code.

In the symbolization process (that will be explained in detail later), we need to know the value of every pointer used in the program, in order to correct it when we will add instrumentation later on. Thus we are required to develop an analysis technique that lets us recover the value of every single global pointer used in the binary. We will now shortly describe the ideas behind the solution we implemented.

Our solution for the global variable problem

At first, some basic static analysis is performed on the binary, in order to recover functions, control flow, basic blocks and disassembly of the `.text` section. After this, we scan the disassembly for each possible instance of pointer building in the binary. After analyzing common compiler patterns, we found out that the "`adrp`" instruction (which loads the base page address of a pointer) is an indicator of a possible start of a pointer building process.

After collecting all the possible instances of pointer building, the next step is to find out the final pointer value of each one. This turned out to be an extremely difficult task, as we soon found out that there are too many ways of how a pointer can be built. We implemented a pattern-matching solution at first, trying to detect common compiler patterns for pointer building; while we correctly found out the value of the vast majority of pointers, a single mistake could make the binary crash, and our solution was not working on binaries of large sizes, as we inevitably failed to parse at least one or two edge-cases.

We later shifted to a different approach: instead of trying to find exact value of a pointer by pattern matching, we take the set of all possible sections a pointer could have and exclude wrong values until possible. Over 99% of the times, this approach leaves only a single section. We then need to symbolize only the initial "`adrp`" and keep the offsets from the relevant section exactly the same as they were in the original binary to make sure that any pointer built with that "`adrp`" will resolve correctly. The rest of the times, we fall back to the old pattern matching solution.

This final solution scales really well, as proved by the fact that we rewrote very large binaries and successfully ran them through long benchmarks. For more details on how the exclusion algorithm works, see the next chapter, "Implementation".

3.3.2 Jump table target recovery

There is a big difference in how jump tables in `ARM` are implemented compared to `x86`. In fact, in `x86`, a jump table is represented through a list of code pointers in the `.rodata` section. The assembly generated by the compiler will simply load the pointer from the list indexed by the number of the case that is going to be executed, and jump to it.

On `ARM`, things are different: jump tables are stored as a list of *offsets* from the base case (case number 0) in memory. The compiler generates assembly that fetches the correct offset based on the case number from the list in memory, adds the offset to the base case, and jumps to the resulting value. Listing 3.1 shows an example of jump table access in `aarch64`. The first two instructions build a global pointer to where the jump table is stored in memory. In line 3 the offset to the corresponding case is loaded into register "`w1`", and then later added to the base case "`x0`" on line 5.

Furthermore, jump tables in `aarch64` are complicated by the fact that they

```
1 adrp x0, <jump_table_page_address>
2 add x0, x0, <jump_table_page_offset>
3 ldrb w1, [x0, w1, uxtw]
4 adr x0, <base case address>
5 add x0, x0, w1, sxtb 2
6 br x0
```

Listing 3.1: Example of a jump table in `aarch64`

are often *compressed* in memory. Since they store offsets, not pointers, and commonly jump table cases are very close to the base case, compilers usually avoid using the full 8 bytes of memory for each case (which would be normal in `x86`), but will use less if possible. For instance, if all offsets are less than 256, the compiler will use a single byte in memory to store each case.

Detection of jump tables

The first problem we had to face was the discovery of jump tables. While they have a very distinct pattern (a load from memory, followed by some arithmetic, and then an indirect jump), many other constructs share similar patterns (e.g., using a callback in a struct). We found out that a reliable way of detecting them is by backward-slicing every time the disassembler encountered an indirect jump, and then verifying if the value of the register used for the indirect jump could be represented with an expression which could be resolved statically and matched a very defined pattern (load 1/2/4 bytes from memory, load a base address, add the offset and then jump to the result).

To represent the value of a register as an expression, we developed a simple pseudo-emulation engine that steps backwards from a given instruction, following control flow and building step by step the resulting expression, similar to what a dumbed-down symbolic executor would output. The pseudo-emulation engine is limited, supports circa 20 instructions, as emulating `ARM` was out of the scope of the project and we only needed it for jump table

detection. A detailed explanation of how it works is in the next chapter.

Detection of jump tables size

Another problem that quickly arose from the peculiarities of `ARM` jump tables is that it is much harder to estimate the number of cases that a jump table supports, compared to `x86`. In fact, in `x86`, simple heuristics such as scanning memory for continuous sections of valid instruction pointers until an invalid one is found can be a valid heuristic. However, as stated before, in `ARM` jump tables are indistinguishable from random bytes, so it is impossible to use heuristics to understand the bounds of a particular jump table in memory.

We found that backward slicing is again a robust solution here too. After detecting a jump table, we can identify the instruction that takes care of loading the offset from memory, and from there we mark the register that holds the value of the number of the case that is going to be executed. Backward-slicing until a comparison operation is performed on the marked register, bounding the number of cases to an absolute number, turned out to be a very reliable solution.

3.3.3 Enlarging jump tables

Another problem spawned from how jump tables are represented in `ARM` comes up when instrumenting a function that contains a jump table. In fact, it is very likely that adding too much instrumentation inside a single case could *overflow* one of the offsets that stored its distance from the base case. Especially when maximum compression is used and offsets are stored in a single byte, it is very common to overflow multiple of them even with light instrumentation.

This was one of the hardest problems to fix, and we considered the following

solutions:

- Expand the jump table in memory. Enlarge the `.rodata` section and move everything to make space in memory for the expanded jump table. While possible, this would have been be a drastic change that was not scalable or easily implemented.
- Create a new jump table in a new section, and patch the pointer building code at the start of the jump table access code. While this was the easiest solution to implement, we discarded it because of the additional space required and its poor scalability.
- Divide all the offsets by the same constant value. Normally, all offsets of a jump table represent the distance between a case and the base case expressed in bytes divided by 4. This is because each instruction is 4 bytes long, and it would not make sense to point inside an instruction. In fact, in Listing 3.1, line 5, we can see how the offset is shifted by 2 to the left (so multiplied by 4). However, we can use the same technique the compiler uses and store offsets divided by 8, 16 or more, and changing how much the offset is shifted to the left before being used, thus enabling us to store larger differences in a single byte.

  The trade-off with this approach is that offsets can no longer point to a single instruction, but to a block of 2, 4 or more instructions, depending on how much enlargement was needed. To make sure that each offset points to the right instruction, some light *nop-padding* is applied between cases to make sure that alignment is correct every time.

We ended up using the last solution, as even if it was slightly more complex to implement, it would help us keep the original memory layout of the binary, which is a very desirable property in binary rewriting.

3.3.4 Control Flow broken by instrumentation

When adding substantial amount of instrumentation to a binary, some pc-relative branches can break, like the instruction "`cbz`", which cannot jump to addresses farther than 1MB.

In this cases we fix the relevant instruction by making them point to a some additional instrumentation containing a trampoline to the original target of the branch.

3.3.5 Instrumentation register saving

We designed RetroWrite-ARM to support any kind of instrumentation, without sacrificing performance and functionality. We realized though that many different kinds of instrumentation require some intermediate calculations to be saved in registers. This was causing noticeable overhead in the instrumentation, as registers needed to be saved on the stack and later restored at every instrumented location.

To avoid this additional overhead, we implemented a static analysis of register usage for every function, with instruction-level granularity (i.e., the result of the analysis is the set of registers that can be freely used without saving them for every instruction inside a given function). The instrumentation can then use the set of free registers without worrying about hindering the original functionality of the binary.

Chapter 4

Implementation

In this chapter we cover the implementation of the rewriter and of BASan, the memory sanitization instrumentation pass. We will also share details on the optimizations we implemented to minimize instrumentation overhead.

4.1 Symbolizer

Symbolization requires detection of every single pointer and control flow mechanism in the binary. In `aarch64`, this may prove to be harder than it looks, as pointer construction patterns are difficult to recover and jump tables are not as heuristics friendly as they are in x86. In the following subsections we will go over each problem and explain our approach to tackle it.

4.1.1 Detecting pointer constructions

Standard compilers (clang, gcc) that target `aarch64` use a common pattern for building pointers: an "`adrp`" instruction, loading the page of the destination address, and then either an "`add`" or similar arithmetic instruction to fix the offset inside the page, or a memory operation like "`ldr`" or "`str`" that include the offset inside the page (e.g., "`ldr x0, [x1, 256]`").

```
adrp x0, 0xab0000
add x1, x0, 256 ; pointer 0xab100
ldr x2, [x0, 512] ; pointer 0xab200
add x0, x0, 128 ; pointer 0xab080
```

Listing 4.1: Example of multiple pointers built from the same `adrp` instruction

```
adrp x0, 0xab0000
mov x1, x0
add x1, x1, 256 ; pointer 0xab100
```

Listing 4.2: Example of changing register during pointer construction

We implemented two different approaches and combined them together to successfully recover pointers in `aarch64` binaries. The first one is based on pattern matching. We first build a list of possible pointer building locations, enumerating all instances of the "`adrp`" instruction. Next, we find all "`add`", "`ldr`", or "`str`" instructions (or their variants) that use the register that was partially built with the "`adrp`", and try to recover the original pointer by emulating the arithmetics involved in those instructions. This approach alone was not enough because of the following difficulties:

- Multiple pointers built with the same "`adrp`": Listing 4.1 shows how sometimes the same "`adrp`" page loading instruction is used for multiple pointer constructions, sometimes very far away from each other
- Moving base page register: another difficulty was that sometimes the register used to store the base page changed in the middle of the pointer construction, like in Listing 4.2
- Base register stack saving: in very large functions, sometimes the base registers were loaded at the start and saved on the stack, to be later restored and used for pointer building. An example is present in Listing 4.3

We implemented a light data flow recovery algorithm that statically ana-

```
adrp x0, 0xab0000
str x0, [sp, -16!]
...
ldr x3, [sp, 16!]
add x3, x3, 512 ; pointer 0xab200
```

Listing 4.3: Example of base page register stack saving

lyzed the control flow and the stack usage of a given function, following around the register used by the "adrp" and checking for its usage, to address all the difficulties stated above. However, it is particularly hard to support every single edge case, and missing a single pointer symbolization is fatal and will cause a crash when the pointer is dereferenced (which sometimes happens a while after the pointer is built, and can be very time consuming to detect and debug). While this pattern matching approach alone worked with the vast majority of instances of pointer construction, it was insufficient to completely symbolize all pointers and often failed on large binaries.

Our second approach took advantage of the fact that RetroWrite-ARM does not instrument data sections, and the vast majority of global variables point to a data section. Instead of trying to parse the pointer building patterns, we try to guess which section of the original binary an "adrp" could be pointing to. Since the "adrp" loads a base page, and a page offset is added, the sections that can be addressed by a single pointer construction are those that overlap the "adrp" address with a $\pm$ 1 KB range. Since all sections except `.text` are not instrumented, if we are able to narrow down the possible target of a pointer construction to a single section, we can symbolize the pointer by just adjusting the starting "adrp" to correctly address the same symbolized section as in the original binary, since offsets used by "add" or "ldr" will stay the same. For example, if we encounter the instruction "adrp x0, 0xab000" and the only section close enough is the `.bss` that starts at address `0xab256`, we can symbolize every pointer construction on "x0" by

changing the above "`adrp`" to "`ldr x0, =(.bss - 256)`".

This second approach is more stable, as it does not incur in any of the problems stated above. However, it was not always applicable, as especially in smaller binaries multiple sections could be in the $\pm$ 1 KB range from the "`adrp`" destination address; in that case, we used some simple data flow analysis to exclude as many sections as possible. We found out that in around 99% of cases we are able to use this "`adrp`"-adjusting approach without needing to do any heuristics at all. In the case we are not able to determine which single section the "`adrp`" is pointing to, we fall back to the first pattern matching based approach.

4.1.2 Symbolization of pointers

After detection of a pointer construction in the target binary, it is still not trivial how to symbolize a pointer. There are two solutions to this problem: using *literal pools* and using pointer construction.

Literal pools

A literal pool is a special region of memory in a binary that stores absolute addresses. It is widely used in `ARM` to overcome the challenge of not being able to store a pointer in a single instruction.

The `ARM` assembly specification [18] states that the assembler will store a pointer in a literal pool when using the following construct: `ldr x0, =<pointer>`. This pseudo-instruction will be assembled with a pc-relative load to the nearest literal pool that contains the full pointer address. The location of the literal pool must be manually specified in assembly through the `.ltorg` directive. Usually, literal pools are stored between functions in the `.text` sections. Since the "`ldr`" pc-relative load can only target addresses in the $\pm$ 1 MB range, literal pools must be stored inside functions if they are larger

than 2 MB.

In our first implementation we used literal pools to symbolize pointers, but we detected noticeable overhead introduced even without instrumentation added. The runtime of symbolized binaries without instrumentation was around 5% higher when compared to the original binaries. The reason behind the additional overhead is twofold: first, each pointer retrieved through literal pools requires a memory access each time; secondly, literal pools occupy precious space in the `.text` section causing more cache misses than necessary.

Pointer construction symbolization

To avoid the overhead introduced by the usage of literal pools, we decided to use pointer construction ourselves. The symbolization of a pointer construction is composed of two parts: the symbolized "`adrp`" base page loading and the symbolized "`add`" for the page offset. An example of such process can be found in Listing **??**. The "`adrp`" is symbolized by just substituting its argument address with the symbolized label. The page offset part, instead, is symbolized through a special assembler directive that evaluates to the last 12 bits of the specified assembly label (and 12 bits are exactly enough to specify the offset inside a page).

```
0x400: adrp x0, 0xab000
0x404: add x0, x0, 256 ; pointer to 0xab100
```

```
.LC400: adrp x0, .LCab100 ; base page
.LC404: add x0, x0, :lo12:.LCab100 ; page offset
```

Listing 4.4: Example pointer construction in the original binary (above) and symbolized pointer construction (below)

4.2 Jump Tables

Switch statements in ARM binaries are stored as a list of offsets, instead of absolute addresses like in x86. This makes symbolizing them particularly tricky. First of all, detecting them is not easy: listing 4.7 shows how they do not have a particular pattern in memory. In this section we will go over what was our approach to finding them and how did we symbolize them without breaking their functionality.

```
0x400: adrp x0, 0x8000
0x404: add x0, x0, 3
0x408: ldrb w1, [x0, w1, uxtw]
0x40c: adr x0, 0x418
0x410: add x0, x0, w1, sxtb 2
0x414: br x0
0x418: movz x0, 1 ; case 0,1,2
0x41c: ret
0x420: movz x0, 10 ; case 3
0x424: ret
0x428: movz x0, 100 ; case 4
0x42c: ret
```

```
0x8003: .byte 0 ; case 0
0x8004: .byte 0 ; case 1
0x8005: .byte 0 ; case 2
0x8006: .byte 8 ; case 3
0x8007: .byte 16 ; case 4
```

Listing 4.5: Left: code for a performing a switch. Register w1 holds the case number that is going to be executed. The offset is loaded at 0x408, which is added to the base case address (loaded at 0x40c) and then jumped into (0x414).
Right: corresponding jump table in memory, with 5 cases each occupying a single byte in memory. Cases can be repeated, and are impossible to distinguish from other data in memory.

4.2.1 Detection of jump tables

Our algorithm to detect a jump table pattern works as follows:

- Recover the complete control flow of each function, using the linear sweep technique.
- Mark all indirect jump locations, identified by "br" instructions.
- For each "br" indirect jump, backwards-slice the code to find all paths that may lead to the "br", with a maximum path length of 50 instructions. This upper bound is generous (in all the cases we analyzed

15 instructions were enough) and prevents computationally expensive path explosions.

- Reverse-emulate every path, and store every possible (symbolized) value that the register of the indirect call can have.
- If the value that the register can have is the same for every path, and corresponds to a jump table symbolic expression, then mark the `br` instruction as part of a jump table construct.

To reverse-emulate every path that leads to the indirect call, we implemented a very limited `aarch64` symbolic instruction emulator. It supports around 20 `ARM` instructions (all those that are common in jump table constructs, plus arithmetic instructions and a few memory-related ones). Figure 4.6 shows a very simple example of the output of this emulator (jump table constructs are often more nuanced and interleaved with other instructions).

After we get the symbolic value of the indirect jump register, we compare it to the standard jump table expression, which is the following:

```
base_case_addr + *(jump_table_base_addr + register_case_number * ?) << ?
```

The symbol `?` is a wildcard for any (positive) integer value. `base_case_addr` is the address of case 0 in the jump table. `jump_table_base_addr` is instead the address in memory of the jump table offsets. Lastly, `register_case_number` is a register with as value the number of the case that is going to be executed.

Finally, if the indirect call is recognized as a jump table, the last step is determining how many cases the jump table is made of. We solved this by backward-slicing from the instruction loading the case number (the "`ldr`" at `0x408` in Figure 4.6) and looking for an upper-bound comparison with a constant value on the register that holds the current case number to be executed (`w1`). If the comparison is directly followed by a jump, than we mark the jump target location as the "default" case and set the number of

cases of the jump table based on the constant of the comparison. Figure 4.6 shows an example of the output of our emulator when analyzing a "br" indirect jump.

```
0x3f8: cmp w1, 128
0x3fc: b.hi .default_case
0x400: adrp x0, 0x8000
0x404: add x0, x0, 3
0x408: ldrb w1, [x0, w1, uxtw]
0x40c: adr x0, 0x418
0x410: add x0, x0, w1, sxtb 2
0x414: br x0
```

```
Analyzing 0x414: br x0
x0 = x0
x0 = x0 + (w1 << 2)
x0 = 0x418 + (w1 << 2)
x0 = 0x418 + (*(x0 + w1*1) << 2)
x0 = 0x418 + (*(0x8003 + w1*1) << 2)
Result:
Base case: 0x418
Jump table addr: 0x8003
Case number reg: w1
Number of cases: 128
Shift: 2
```

Listing 4.6: Above: example of a jump table pattern. Below: output of our symbolic emulator.

4.2.2 Jump Table symbolization

The symbolization of a jump table in memory is done using the assembler's support for simple arithmetic on assembly labels. Since on ARM a jump table is a list of offsets from a base instruction, we symbolize that with differences between labels. An example of this can be seen in Figure 4.7.

Since the offsets in the symbolized version are calculated with assembly labels, any amount of code can be added between cases, and the assembler will make sure that the jump table will work correctly. This is one of the cases where the benefits of using symbolization as a rewriting technique really shines, as it gives us the freedom of inserting arbitrary instrumentation

(even by hand) without having to worry about correcting references.

However, there is a catch: adding *too much* instrumentation could overflow the value used to store the offset from the base case. In the example in Figure 4.7, if there are more than 256 instructions between a case and the base case, the offset will overflow as it is stored in a single byte. In the next subsection we will cover how we actually support adding arbitrary amount of instrumentation.

```
0x8003: .byte 0 ; case 0
0x8004: .byte 0 ; case 1
0x8005: .byte 0 ; case 2
0x8006: .byte 8 ; case 3
0x8007: .byte 16 ; case 4
```

```
0x8003: .byte (.LC418 - .LC418) / 4
0x8004: .byte (.LC418 - .LC418) / 4
0x8005: .byte (.LC418 - .LC418) / 4
0x8006: .byte (.LC420 - .LC418) / 4
0x8007: .byte (.LC428 - .LC418) / 4
```

Listing 4.7: Left: jump table as stored in memory as found in the original binary. Right: symbolized jump table in the output of RetroWrite-ARM.

4.2.3 Jump Table enlargement

When too much instrumentation between jump table cases is added, the value used to store the offset from the base case can overflow. To address this, we implemented support for using a larger divisor when storing assembly label differences. As an example, instead of storing `(LC418 - .LC410) / 4` like in Figure 4.7, we can store `(.LC418 - .LC410) / 8` to fit up to 512 instructions between `.LC418` and `.LC410`. The same reasoning can be reapplied with higher powers of two.

However, using a divisor higher than 4 means losing precision in the addresses of the cases we want to represent. Since 4 bytes is the size of each instruction, dividing by 4 means that every instruction can be targeted; dividing by 8 means that only one every two instructions can be targeted. To avoid having jump table cases not targetable due to the loss of precision, we insert `nop` padding before each case in order to make every case aligned and

targetable, with the amount of nops depending on how high is the divisor (e.g., dividing by 8 means that each case must by 8-byte aligned, so using up to 1 nop before each case; dividing by 16 means using up to 3 nops, and so on). Since the number of nops inserted depends on alignment, we leave this task to the assembler using the "`.align`" directives.

After changing the divisor, we also need to change the indirect jump calculations in the binary's code to match the new shift value. Usually, the offset to be added is multiplied by 4 using an instruction like "`add x0, x0, w1, sxtb 2`" (which shifts left by 2), as can be seen in Figure 4.6. We change the shift value according to how high we set the dividend in the symbolized jump table (the `add` instruction support shifting left up to 4, but we insert a standard "`lsl`" instruction before if it's higher than 4).

Listing 4.8 highlights an example of this.

4.3 Instrumentation (BASan)

The ASan instrumentation was designed to be compatible with the AddressSanitizer library provided by Google, `libasan`. We carefully selected shadow memory offsets and sizes to match those included in `libasan`. The library will hook on each call to `malloc` and `free`, writing in the shadow memory the available bytes that can be used. RetroWrite-ARM takes care of finding each access in memory and inserting instrumentation just before each access to check the relevant bytes of shadow memory and error out in case an overflow or other memory corruption was found.

Listing 4.9 shows the ASan checking algorithm in high level. To implement it as an instrumentation pass, we manually wrote assembly code that matched its functionality and could be adapted to both reads and stores. Different versions of ASan snippets were developed depending on the size of the

```
.LC400: adrp x0, .LC8003
.LC404: add x0, x0, :lo12:.LC8003
.LC408: ldrb w1, [x0, w1, uxtw]
.LC40c: adr x0, .LC418
.LC410: add x0, x0, w1, sxtb 4
.LC414: br x0
.align 4
.LC418: movz x0, 1 ; case 0,1,2
.LC41c: ret
.align 4
.LC420: movz x0, 10 ; case 3
.LC424: ret
.align 4
.LC428: movz x0, 100 ; case 4
.LC42c: ret
...
...
0x8003: .byte (.LC418 - .LC418) / 16
0x8004: .byte (.LC418 - .LC418) / 16
0x8005: .byte (.LC418 - .LC418) / 16
0x8006: .byte (.LC420 - .LC418) / 16
0x8007: .byte (.LC428 - .LC418) / 16
```

Listing 4.8: Example of enlarged symbolized jump table. The shift value at `0x410` is increased from 2 to 4, and the divisor at `0x8003` is increased from 4 to 16. Before each case, an `.align 4` assembly directive is inserted, as each of them need to be aligned to 16 bytes.

```
byte shadow_value = *(MemToShadow(address));
if (shadow_value) {
  if ((address & 7) + AccessSize - 1 >= shadow_value) {
    ReportError(address, AccessSize);
  }
}
```

Listing 4.9: AddressSanitizer checking algorithm example implementation

store/load, to ensure maximum efficiency and avoid wasting calculations at runtime. Listing 4.10 shows an example of an instrumented instruction.

Unfortunately, the BASan instrumentation pass is not completely equivalent to its source based counterpart, in fact we can highlight the following differences:

- Missing global variable bounds checking: without the source code, it is impossible to distinguish boundaries between global variables in data

```
.ASAN_ENTER:
; saving registers on the stack
stp x17, x16, [sp, -16]!
stp x15, x14, [sp, -16]!
str x13, [sp, -16]!
; saving condition flags register
mrs x13, nzcv
; loading from shadow memory
mov x14, 0x1000000000
lsr x16, x1, 3
ldrsh w15, [x14, x16]
; if shadow memory is empty (w15 == 0), continue execution
cbz w15, .LC_ASAN_EXIT
; otherwise, report error and quit
mov x0, x1
bl __asan_report_load16_noabort

.LC_ASAN_EXIT:
; restoring condition flags register
msr nzcv, x13
; restoring registers from the stack
ldr x13, [sp], 16
ldp x15, x14, [sp], 16
ldp x17, x16, [sp], 16
; original instruction
ldp x2, x3, [x1]
```

Listing 4.10: Example of BASan memory sanitization on a 16-byte memory load

section. For this reason, checks on global variables are missing.

- Missing stack variables bounds checking: similar to above, without source code it is extremely hard to find boundaries between variables on the stack. Stackframe bounds checking could be easily implemented (thus providing the same functionality as a stack canary), but for now BASan has checks only for the heap.
- Number of instrumented locations: ASan used as a compiler pass is able to prune the checking on a lot of memory accesses, since on many instances the safeness of a memory access can be determined at compile-time with the source code at hand. However, RetroWrite-ARM works only on binaries and thus is forced to instrument all mem-

ory accesses.

Despite the following limitations, the BASan instrumentation pass provides the same core functionality of the original ASan compiler pass (memory sanitization on the heap), and with very little additional overhead.

Optimization: register savings

The snippets of assembly used to implement ASan memory checking require some temporary register usage to store intermediate calculations, and thus require special precautions before inserting them in the middle of a binary. Our first approach was to save the values of each register we were planning to use on the stack, and then restore the old values as the last step of the instrumentation. However, we quickly realized that those frequent memory accesses were introducing noticeable overhead.

We then switched to performing a static analysis of register usage on every function of the binary, in such a way that at any given instruction we know which registers can be freely used and which cannot be modified without hindering the correct execution of the binary. Then, when inserting the instrumentation, we modify the snippets in such a way that they will try to use as many 'free' registers as possible, while saving the others on the stack.

Of particular note is the saving of the value of register "`nzcv`", which holds the condition flags in `aarch64`. This register is always saved and restored at each instrumented location, as all ASan snippets contain a conditional branch for the error checking.

Chapter 5

Evaluation

In this section we validate the claims that we made earlier by performing experiments. We show that RetroWrite-ARM can symbolize and instrument `ARM` binaries and that presents important features such as the following:

- Scalability: support for large binaries.
- Performance: low instrumentation overhead.
- Correctness: instrumentation does not affect original functionality of the binary.

5.1 Setup and Hardware

We decided to run experiments on two different machines: a low end setup and a high end setup. Evaluating on machines at opposite end of the performance spectrum allows us to highlight differences between the two device classes. The machines we had available are the following:

- A raspberry Pi 4B, with a Cortex A-72 (1.5GHz) CPU and 4GB of RAM
- An Atlas/A57 (2.4GHz) CPU with 64 GB of RAM

To benchmark the performance of rewritten binaries, we used the C language benchmarks of the SPEC CPU 2017 benchmark suite [19]. We focus

on the C language benchmarks as RetroWrite-ARM does not (yet) support the symbolization of C++ exceptions or class hierarchy recovery. Table 5.1 shows the list of evaluated benchmarks. All of the benchmarks were run on both machines, but on the Raspberry some benchmarks were excluded from the results as 4GB of RAM were not enough to avoid using swap memory, compromising the experiments. The Atlas machine was generously hosted by the CloudLab project [6] in their Utah data center.

The benchmarks were compiled with the `gcc` compiler version 7.5.0 on Ubuntu 18.04. The following command line flags were used to compile the baseline benchmark binaries: "`-O3 -fgnu89-inline -fno-strict-aliasing`", in addition to the flags to produce position independent executables. The "`-fgnu89-inline`" flag uses GNU semantics for inline functions, and resolves issues with duplicate symbols errors during the compilation of the `gcc` benchmark. The "`-fno-strict-aliasing`" flag disables GCC's aggressive aliasing compilation, and is recommended to be used by the SPEC CPU manual [28] to avoid problems with the `perlbench` benchmark.

To demonstrate the scalability of our approach, we rewrite binaries as large as the `gcc` benchmark (>10MB), which results in over 125MB of symbolized assembly. We evaluate the performance of the symbolization engine by running the benchmarks rewritten by RetroWrite-ARM without adding any kind of instrumentation. We then evaluate the performance of our memory sanitization instrumentation pass by comparing it against its source based equivalent ("`-fsanitize=address`" in the compiler flags), and also against memory sanitization implemented using a dynamic instrumentation approach, by using Valgrind with the memory sanitization option ("`valgrind --tool=memcheck`").

Name	Size	Stripped size	.text size	Functions
cpugcc_r	63 MB	10 MB	8912092	11799
perlbench_r	11 MB	2.5 MB	2070252	2408
imagick_r	2.4 MB	2.2 MB	1935340	2187
x264_r	687 KB	645 KB	563836	547
nab_r	249 KB	229 KB	194756	235
xz_r	244 KB	215 KB	160540	360
mcf_r	45 KB	38 KB	30964	46
lbm_r	23 KB	18 KB	10508	28

Table 5.1: Name and size of the SPEC CPU2017 benchmarks written in C.

5.2 Performance

5.2.1 Symbolization performance

Figure 5.1 and Figure 5.2 demonstrate that the overhead of reassembled symbolized binaries is negligible. Table 5.3 shows more detailed results for the Atlas machine. The average overhead is 0.76%, in line with other state of the art static rewriters [34] for other platforms.

While the overhead is low to negligible, we want to list the two main causes: alignment and pointer construction. First, RetroWrite-ARM currently does not conduct any cache line optimization for code and section layouts. Compilers layout code blocks to maximize code cache hits. Second, each pointer building instruction in the original code "`adrp`" is symbolized with two instructions (an "`adrp`" followed by an "`add`"): so for each global variable access the binary makes the CPU will execute an extra clock cycle. Note that, in practice, this will not cost a full clock cycle as many `ARM` CPUs have hardware optimizations for `adrp`+`add` sequences of instructions [7].

Interestingly, at first we used to have substantial overhead from symbolization alone. This was at an early stage of the project, where we used literal pools to symbolize global variable pointer constructions, instead of generating symbolized pointer constructions ourselves like compilers do, mostly

Name	Static pointer constructions	Dynamic pointer constructions	Literal pools	Symbolized pointer building
perlbench_r	34 285	168 797 437 831	19.48%	2.91%
gcc_r	9095	1 232 266 305	4.10%	0.95%
imagick_r	19 127	16 275 621 901	1.75%	0.30%
nab_r	2003	28 193 001 045	0.88%	0.34%
xz_r	1087	1 471 854 761	0.33%	0.44%
mcf_r	108	7 909 505	1.07%	−0.07%
lbm_r	90	36 160	0.08%	0.59%
Average	-	-	**4.12%**	**0.76%**

Table 5.2: Overhead of RetroWrite-ARM on the Atlas machine without instrumentation comparing the recovery of pointers by using literal pools and by using symbolized pointer building.

because using literal pools was simpler to debug and faster to implement. However, literal pools introduce an additional instruction (a memory load), plus they require space in the `.text` section for the storage of the pointers, reducing cache performance. Table 5.2 demonstrates how much the overhead decreased when we switched to symbolized pointer constructions, from an average of 4.12% to 0.76%.

The most affected benchmark by the symbolization overhead is `perlbench`, and this fact is consistent with our measurements (as `perlbench` has an order of magnitude more global variable accesses executed compared to the other benchmarks,) and is also the benchmark which benefited the most from the removal of literal pools. A negative overhead is present on the `mcf` benchmark, but we consider it small enough to lie inside measurement error.

5.2.2 Memory sanitization performance

In this section we compare the overhead introduced by RetroWrite-ARM's memory sanitization instrumentation and the overhead introduced by adding memory sanitization during compilation ("`-fsanitize=address`").

An overview of the results can be seen in Figure 5.1 for the Atlas machine,

and in Figure 5.2 for the Raspberry Pi. Both figures show the same pattern: memory sanitization introduces some overhead but the execution time remains similar to source-based instrumentation. However, we can see how the Raspberry suffered from a slightly higher overhead with instrumentation enabled. We think that one of the possible causes of this is that a low-end system like the Raspberry suffers more from cache misses than a high-end system (and memory sanitization is a very heavy instrumentation that increases code size by up to three-fold, substantially degrading cache performance).

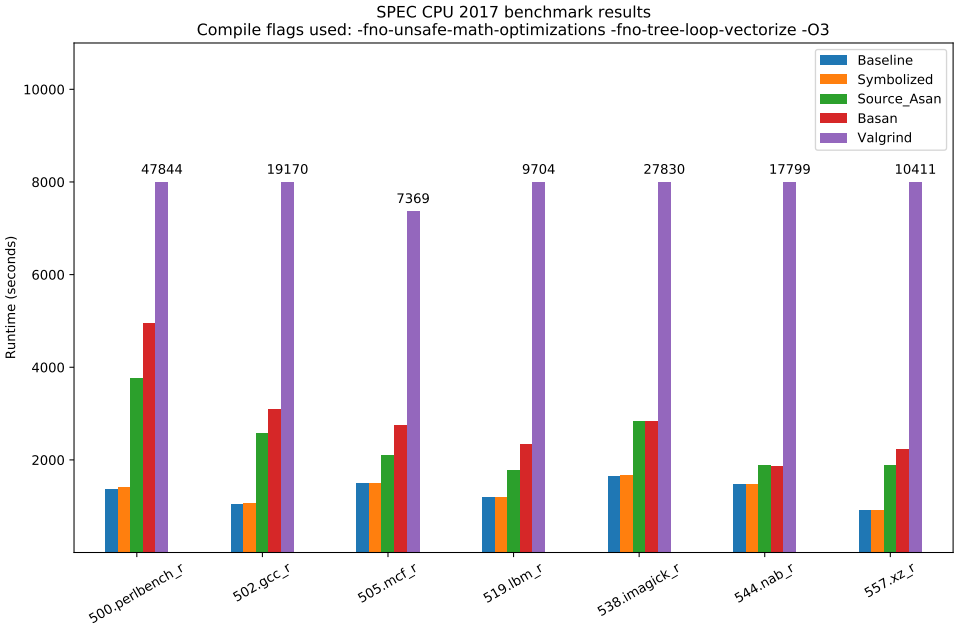

Figure 5.1: Benchmark runtime on the Atlas machine. The yaxis has been limited at 8000 for clear depiction of smaller values.

It is interesting to look at detailed results for the Atlas machine in Table 5.3. We can see that source-based ASAN has a 84.04% overhead on average, but the overhead varies a lot between individual benchmarks, depending on the

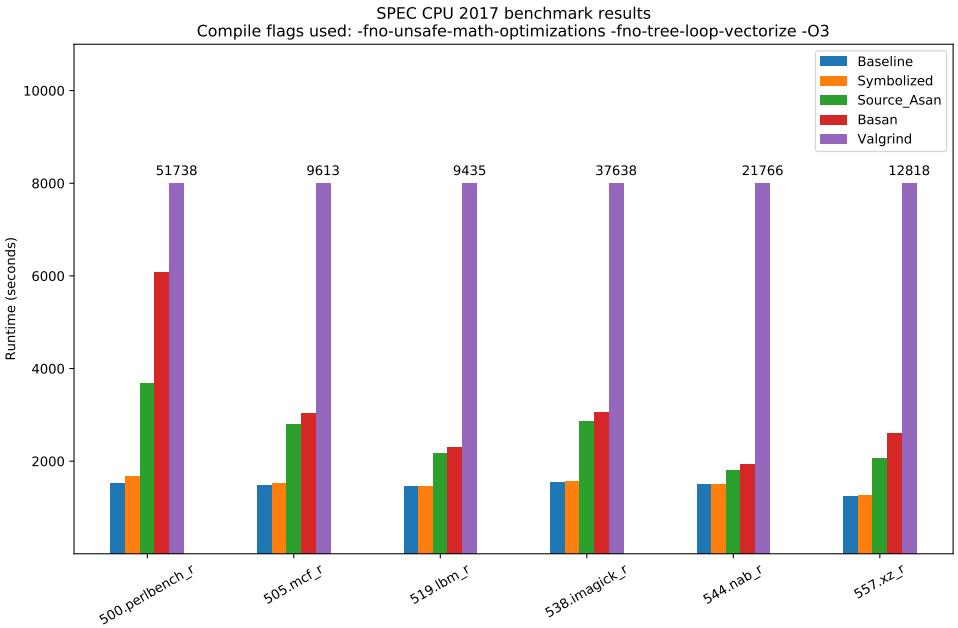

Figure 5.2: Benchmark runtime on the Raspberry machine. The yaxis has been limited at 8000 for clear depiction of smaller values.

amount of memory accesses. For example, `perlbench`, running an interpreter, executes a high amount of memory loads and stores that need to be individually checked by the address sanitizer each time; in fact, the overhead is noticeably higher (173.84%). The same holds true for RetroWrite-ARM's memory sanitization instrumentation ("Binary ASAN" in Table 5.3): the overhead is larger in programs that frequently use memory.

On average, binary ASAN is 19% slower than source ASAN (119.61% instead of 84.04% average overhead). The reasons behind the slowdown are the following:

- Number of instrumented locations: Without source code available, RetroWrite-ARM cannot use certain types of analysis that the compiler uses to determine the safety of certain memory accesses. For this reason, RetroWrite-ARM instruments each memory access, while source ASAN skips some of them.

Name	Symbolization only	Source ASAN	Binary ASAN	Valgrind
cpugcc_r	0.95%	145.80%	196.09%	1729.20%
perlbench_r	2.91%	173.84%	260.10%	3377.03%
imagick_r	0.30%	71.53%	71.17%	1578.53%
nab_r	0.34%	28.57%	27.21%	1110.82%
xz_r	0.44%	105.79%	144.76%	1036.57%
mcf_r	-0.07%	40.77%	83.74%	390.94%
lbm_r	0.59%	49.58%	97.31%	715.46%
Average	**0.76%**	**84.04%**	**119.61%**	**1429.94%**

Table 5.3: Overhead of RetroWrite-ARM without instrumentation and of RetroWrite-ARM with BASAN instrumentation on SPEC CPU2017 on the Atlas machine compared to the original benchmark and the original benchmarks compiled with source based ASAN.

- Assembly generation optimizations: RetroWrite-ARM is a straightforward rewriter and instruments memory accesses by inserting hand-written assembly snippets. While some optimizations were implemented (register savings), RetroWrite-ARM so far has no peephole optimizer to reflow the emitted instrumentation.

Still, the slowdown with respect to source ASAN is tiny compared to state of the art tools. In fact, without source code, the only way of adding memory sanitization to a binary is through dynamic instrumentation. We ran the benchmarks using the current state of the art dynamic instrumentation tool that implements memory sanitization (Valgrind's `memcheck` tool [23]), and plotted the runtime in Figure 5.1 and Figure 5.2. The overhead introduced by dynamic instrumentation is an order of magnitude higher than the overhead of binary ASAN, depending on the amount of memory accesses of the benchmark.

Note that, so far our binary ASAN instrumentation does not detect out of bounds accesses for global variables and stack accesses. While loads/stores are instrumented and checked, the underlying memory redzones are not instrumented due to lack of precise variable information. The length and scope of globals and stack variables is not determinable in the general case

and without this information, we cannot check them precisely.

5.2.3 Optimization: register savings

One of the biggest performance improvements in our memory sanitization instrumentation came from the re-usage of "free" registers without the need to save them on the stack at each instrumentation location. The impact of this optimization depends on the amount of registers that our static analysis determines to be "free" at every given instrumentation location. In the best case scenario (if no registers need to be pushed to the stack and then restored) 6 less instructions are used for the BASan instrumentation. Figure 5.1 shows an example of this optimization on an instrumented 8-byte memory load.

```
stp x17, x16, [sp, -16]!
stp x15, x14, [sp, -16]!
str x13, [sp, -16]!
mrs x13, nzcv
add x17, x1, 2488
mov x14, 0x1000000000
lsr  x16, x17, 3
ldrsb  w15, [x14, x16]
cbz  w15, .LC_ASAN_EXIT
mov x0, x17
bl __asan_report_load8_noabort
.LC_ASAN_EXIT:
msr nzcv, x13
ldr x13, [sp], 16
ldp x15, x14, [sp], 16
ldp x17, x16, [sp], 16
; original instruction
ldr x1, [x1, #0x9b8]
```

```
mrs x13, nzcv
add x17, x1, 2488
mov x14, 0x1000000000
lsr  x16, x17, 3
ldrsb  w15, [x14, x16]
cbz  w15, .LC_ASAN_EXIT
mov x0, x17
bl __asan_report_load8_noabort
.LC_ASAN_EXIT:
msr nzcv, x13
; original instruction
ldr x1, [x1, #0x9b8]
```

Listing 5.1: Left: Instrumented 8-byte memory load. Right: Instrumented 8-byte memory load with register savings turned on (best case scenario).

Table 5.4 shows in detail the overhead difference in the memory sanitization instrumentation when register savings are disabled. We can see how register savings are critical to the performance of the instrumentation, as on average

Name	Register savings	No registers
gcc_r	196.09%	259.64%
perlbench_r	260.10%	453.71%
imagick_r	71.17%	173.40%
nab_r	27.21%	74.22%
xz_r	144.76%	212.01%
mcf_r	83.74%	124.92%
lbm_r	97.31%	99.58%
Average	**119.61%**	**195.79%**

Table 5.4: Overhead of RetroWrite-ARM's memory sanitization with register savings turned on or off. On average, the register savings optimization produces code with 48.1% less overhead.

the overhead is reduced by 48.1%.

5.2.4 Comparison to trampolines

Trampolines are one of the most straight-forward way to instrument a binary, and they are are free from the burden of static analysis required to perform symbolization. However, we would like to show that the tradeoff in performance is very large compared to the in-place instrumentation that symbolization permits to do.

We implemented the same memory sanitization instrumentation pass by using trampolines, and ran the same benchmarks to compare the timings. Results can be seen in Table 5.5.

This was a fairly simple experiment that used only a single type of very heavy instrumentation (memory sanitization adds a lot of branches in the code and a very large number of instructions need to be instrumented), but the difference in overhead is very large, with trampolines being 56% slower on average compared to symbolized in-place instrumentation. We expect lower differences in overhead when using lighter forms of instrumentation, but not lower enough to avoid being noticeable.

Name	Binary ASAN	Trampoline ASAN
perlbench_r	260.10%	636.41%
imagick_r	71.17%	188.96%
nab_r	27.21%	77.82%
xz_r	144.76%	232.64%
mcf_r	83.74%	135.31%
lbm_r	97.31%	104.79%
Average	**109.73%**	**227.38%**

Table 5.5: Overhead of RetroWrite-ARM's in-place instrumentation and trampoline-based instrumentation using memory sanitization as the instrumentation pass. On average, trampolines are 56% slower than in-place instrumentation.

5.3 Correctness

While the SPEC CPU benchmark checks the validity of the output of each benchmark that is run, we decided that we wanted further proof of the correctness of our approach with more widely used and common binaries. For this reason, we selected the coreutils suite of tools, and ran their tests on the binaries rewritten with RetroWrite-ARM.

We used the release 8.32 package downloaded from https://ftp.gnu.org/gnu/coreutils/. We first built the binaries using `make`, then rewrote every single one of them two times: first, without instrumentation, and then with memory sanitization enabled. We then ran the extensive version of their testsuite with `make check-very-expensive`.

The results of the tests can be seen in Table 5.6. Of the 4 failed tests with memory sanitization turned on, two of them are caused by the testsuite trying to `LD_PRELOAD` libraries (which is not compatible with AddressSanitizer) in the `stdbuf` tests, and the other two are because of the overhead of the instrumentation that triggers a timeout in the tests.

	Symbolization only	Binary ASAN
Total	621	621
Passed	540	536
Skipped	81	81
Failed	0	4

Table 5.6: Results of the coreutils "very expensive" test suite on binaries rewritten through RetroWrite-ARM, comparing memory sanitization enabled or disabled. The skipped tests are due to uninstalled third-party dependencies, or architecture specific features missing on the Raspberry.

5.4 Comparison to Egalito

The only state-of-the-art static binary rewriter that had similar features compared to RetroWrite-ARM is Egalito [34]. Egalito is a binary "recompiler" targeting compiled C code as position-independent ARM64 Linux executables that lifts binary to a custom Intermediate Representation (IR) and supports a large variety of instrumentation and transformation passes, and has a (reported) similar baseline overhead to RetroWrite-ARM (0.46% on SPEC CPU2006). The comparison with Egalito is interesting because the authors had to tackle very similar problems as ours (pointer construction and jump table recovery); however, they do not specify how they solved the problem of jump table enlargement in the case of heavyweight instrumentation. Unfortunately, we could not perform any kind of comparison as the version available in the public repository crashes on any `aarch64` binary we tried to rewrite. We opened an issue [1] on their public Github repository on November 28, 2020, but it still left unanswered (as of January 15, 2021).

[1] https://github.com/columbia/egalito/issues/32

Chapter 6

Related Work

6.1 Dynamic binary rewriting

While dynamic rewriters are scalable and do not suffer from the limitations of static analysis, they incur large overheads for translation and execution and are not suited for high performance instrumentation. They can either make use of the operating systems' debugging primitives (ptrace API in Linux, or the application debugging API in Windows), or of a custom loader, like `PIN` [21] or `DynamorIO`, to modify and instrument the target binary at runtime. While they have to solve similar problems compared to static rewriting (such as disassembly and CFG recovery), they can leverage run-time information that is unavailable to static rewriters. For example, a dynamic rewriter knows the target of an indirect branch before executing it, since the register values are known at any given point during the execution. Dynamic rewriters can also be built on top of emulation frameworks, which is the case for `QASAN` [14] which uses `QEMU` to implement AddressSanitizer memory sanitization on any binary supported by `QEMU`.

6.2 Static binary rewriting

We will divide static rewriters by the technique they use to insert instrumentation in the target binary, namely trampolines, lifting to an intermediate representation, and symbolization.

6.2.1 Static rewriters that use trampolines

`E9patch` [12] is a recent static rewriter for `x86_64` Linux ELF binaries, that inserts instrumentation with trampolines. This techniques works by substituting the instruction that need to be instrumented by a non-conditional jump that points to the instrumentation. At the end of the instrumentation, another non-conditional jump brings the execution back to next instruction of the original code. `E9Patch` introduces novel techniques to solve the problem of substituting an instruction in a varying-length ISA (such as `x86_64`).

Using trampolines has the advantage of requiring very low static analysis beforehand; however, a large overhead is introduced at each instrumentation location, as two jumps need to be executed each time.

`BISTRO` [9] is aimed at the rewriting of individual components in an executable, and works by *stretching* the binary to make space for the new instrumentation; however it still makes use of trampolines ("anchors") to avoid breaking indirect call targets.

6.2.2 Static rewriters that lift to IR / full translation

Lifting the target binary to an intermediate representation has multiple advantages. First, if the chosen IR is a standardized one (such as the LLVM IR), a wide variety of tools can be used to apply instrumentation and modify the lifted IR. `McSema` [10] is a great example of an LLVM IR lifting approach that supports `x86_64` ELF and PE binaries, with support for C++ exceptions and `aarch64` support in development. The IR produced by `McSema` can be

readily fuzzed with `libFuzzer`, an LLVM-based instrumented fuzzer that would normally require the source code. The disadvantages in lifting to an IR are in the fact that the lifting requires very heavy static analysis (in fact, `McSema` uses `IDAPro` as backend) and some overhead can be introduced by unoptimal lifting of the original binary.

6.2.3 Static rewriters that use symbolization

In 2015, `Uroboros` [32] was the first approach in generating *reassemblable assembly* from `x86` binaries. Unfortunately, it still suffered from classic challenges in static analysis such as relying on heuristics to differentiate between scalars and references. Two years later `Ramblr` [31] significantly improved on `Uroboros`' approach with better heuristics and less overhead. In 2020, RetroWrite-x64 restricted the class of target binaries to position-independent ones, finally solving the problem of distinction between scalars and references.

6.3 Static rewriters aimed at ARM binaries

In this section we will focus on static rewriters on the `ARM` architecture, which have to tackle the same challenges caused by fixed-size instructions as we did.

`Repica` [15] is one of the most recent static rewriters for both 32-bit and 64-bit `ARM` binaries, and uses an instrumentation technique similar to reassembly: instrumentation can be inserted both before and after an instruction, and relative jumps are carefully adjusted to keep control flow and references exact. The authors use a backwards-slicing technique to detect jump tables similar as ours, and recover pointer construction mechanisms with data-flow propagation. Their instrumentation however does not utilize optimizations such as register savings, incurring in higher overhead than necessary.

`Egalito` [34] is a binary "recompiler" targeted at C, position-independent ELF binaries with support for both 32-bit and 64-bit x86 and `ARM`. `Egalito` lifts binaries to a custom IR to insert instrumentation, supports a large variety of instrumentation and transformation passes, and has a reported similar baseline overhead to RetroWrite-ARM (0.46% on SPEC CPU2006). Detection of jump tables is done through backwards-slicing; the authors do not specify how they solved pointer construction recovery.

Nor `egalito` neither `repica` address the problem of jump table enlargement, fundamental in the case of heavy-weight instrumentation like memory sanitization. Even though it is older, `RevARM` [17] is the only one that supports resizing jump tables to insert large instrumentation; however `RevARM` *stretches* the jump table to make space for larger offsets, instead of multiplying the already existings offsets and keeping the jump table of the same size as we do. Also, `RevARM` only supports 32 bit `ARM` binaries.

To the best of our knowledge, no other static rewriter for `aarch64` that efficiently supports heavy-weight instrumentation such as AddressSanitizer has been proposed, other than the one proposed in this study.

6.4 Summary of related work

We present a short summary of binary rewriters in Table 6.1 for quick comparison. Please note that this table is vastly incomplete and that the rewriters listed in this table were hand-picked by us because were relevant to our work or because were directly referenced previously. For a more detailed summary, we redirect the reader to the survey by Wenzl et al. [33].

Name	Year	Architectures	Type	Technique	Features/Limitations
Bistro	2013	x86(32)	static	stretching	ASLR support Closed-source dependency (IDAPro)
McSema	2014	x86(32,64)	static	LLVM IR	C++ exceptions support. Closed-source dependency (IDAPro)
Uroboros	2015	x86(32,64)	static	symbolization	First symbolization approach
Ramblr	2017	x86(32,64)	static	symbolization	Like Uroboros, but more reliable.
RevARM	2017	ARM(32)	static	stretching	First static rewriter for ARM Enlarged jmptbl support Closed-source dependency (IDAPro)
Repica	2018	ARM(32, 64)	static	stretching	Safe scalar/reference distinction
Egalito	2020	x86(64), ARM(64)	static	custom IR	Safe scalar/reference distinction Many instr passes supported Kernel support (?)
E9Patch	2020	x86(64)	static	trampolines	No CFG recovery needed High overhead due to trampolines
RetroWrite	2021	x86(64), ARM(64)	static	symbolization	Safe scalar/reference distinction Enlarged jmptbl support Memory sanitization support
Valgrind	2002	x86(32,64), ARM(32,64)	dynamic	custom IR	Many instr passes supported
AFL-QEMU	2013	x86(32,64), ARM(32,64)	dynamic	emulation	Big range of supported archs
QASAN	2020	x86(32,64), ARM(32,64)	dynamic	emulation	Big range of supported archs Memory sanitization support
DynamorIO	2002	x86(32,64), ARM(32)	dynamic	virtualization	Many instr passes supported
PIN	2005	x86(32,64)	dynamic	JIT-recompilation	Low dynamic overhead (30%) Closed-source, Intel proprietary

Table 6.1: Summary of some of the state-of-the-art binary rewriting projects.

Chapter 7

Future Work

Support for more source languages: For now, RetroWrite-ARM supports only binaries compiled from the C language, both for the `x86` and the `ARM` implementation. The easiest addition would be to add support for C++ by expanding the analysis capabilities of RetroWrite-ARM to support exception tables too, but many more languages could be supported in the future.

Support for kernel space binaries: Right now the `ARM` port of RetroWrite-ARM supports only userspace binaries, contrary to the `x86` version that supports linux kernel modules too. The kernel version of RetroWrite-ARM would prove to be particularly interesting as it would open new ways to efficiently fuzz Android kernel modules.

Support for more executable formats/operating systems: The current implementation of the RetroWrite-ARM tool is aimed only towards ELF files, but adding support for MACH-O and PE binaries should not require too much effort. This would also be interesting as Windows and MacOS present way more closed-source modules compared to Linux.

More instrumentation passes: While right now we implemented only the AddressSanitizer instrumentation in the `ARM` port of RetroWrite-ARM, the design of RetroWrite-ARM is modular and adding new instrumen-

tation passes or new mitigations should be easy. To name a few, the interesting ones would be:

- Shadow stack (return address protection)
- Control flow authentication using `ARM` pointer authentication (hardware-assisted)
- Coverage-guidance for fuzzing

Chapter 8

Conclusion

In summary, we present new solutions to binary rewriting problems arising from the peculiarities of the `ARM` architecture such global variable symbolization and jump table enlargement, and develop `aarch64` support for the RetroWrite project, a scalable static rewriter for linux C binaries. RetroWrite-ARM enables targeted application of static instrumentation passes where no source code is available, such as proprietary binaries, inline assembly, or code generated by a deprecated compiler. We also present an example memory sanitization instrumentation pass, AddressSanitizer, particularly useful for fuzzing purposes. To the best of our knowledge, RetroWrite-ARM is the only `aarch64` rewriter that supports heavy instrumentation passes like AddressSanitizer. We show that the total overhead of the memory sanitization instrumentation pass are competitive with the source-based AddressSanitizer. Our work shows that the symbolization approach is not limited to the `x86` architecture, but can be applied to the `aarch64` architecture and more.

Bibliography

[1] *Android operating system share worldwide*. https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/. Accessed: 2020-11-22.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Transparent dynamic optimization: The design and implementation of Dynamo". In: (1999).

[3] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[4] Bryan Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching". In: *The International Journal of High Performance Computing Applications* 14.4 (2000), pp. 317–329. DOI: 10.1177/109434200001400404.

[5] *Capstone: The Ultimate Disassembler*. http://www.capstone\protect\discretionary{\char\hyphenchar\font}{}{}engine.org. Accessed: 2020-11-02.

[6] *CloudLab project*. https://www.cloudlab.us/. Accessed: 2020-11-22.

[7] *Cortex A72 Software Optimization Guide*. https://developer.arm.com/documentation/uan0016/a/. Accessed: 2020-11-11.

[8] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." In: *USENIX security symposium*. Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[9] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. "Bistro: Binary component extraction and embedding for software security applications". In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 200–218.

[10] Artem Dinaburg and Andrew Ruef. "Mcsema: Static translation of x86 instructions to llvm". In: *ReCon 2014 Conference, Montreal, Canada*. 2014.

[11] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization". In: *IEEE International Symposium on Security and Privacy*. 2020.

[12] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. "Binary rewriting without control flow recovery." In: *PLDI*. 2020, pp. 151–163.

[13] *Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*. https://frida.re. Accessed: 2021-1-6.

[14] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. "Fuzzing binaries for memory safety errors with QASan". In: *2020 IEEE Secure Development (SecDev)*. IEEE. 2020, pp. 23–30.

[15] Dongsoo Ha, Wenhui Jin, and Heekuck Oh. "REPICA: Rewriting Position Independent Code of ARM". In: *IEEE Access* 6 (2018), pp. 50488–50509. DOI: 10.1109/access.2018.2868411.

[16] *Kernel Module Support*. https://source.android.com/devices/architecture/kernel/kernel-module-support. Accessed: 2020-11-16.

[17] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. "RevARM: A platform-agnostic ARM binary rewriter for security applications". In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. 2017, pp. 412–424.

[18] *Literal pools - ARM compiler user guide*. https://developer.arm.com/documentation/dui0473/m/dom1359731147760. Accessed: 2020-11-22.

[19] *Literal pools - ARM compiler user guide*. https://www.spec.org/cpu2017/. Accessed: 2020-11-22.

[20] *Literal Pools - The ARM documentation*. https://developer.arm.com/documentation/dui0473/m/writing-arm-assembly-language/literal-pools. Accessed: 2020-11-11.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[22] Barton P Miller. "Binary Code Patching: An Ancient Art Refined for the 21st Century". In: *NC State University Computer Science Department Seminars 2006–2007*. 2006.

[23] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[24] *PinePhone, "An Open Source Smart Phone Supported by All Major Linux Phone Projects"*. https://www.pine64.org/pinephone/. Accessed: 2021-1-10.

[25] *Purism Products Homepage*. https://puri.sm/products/. Accessed: 2021-1-10.

[26] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W Davidson, and Mary Lou Soffa. "Retargetable and reconfigurable software dynamic translation". In: *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE. 2003, pp. 36–47.

[27] Konstantin Serebryany. *Hardware Memory Tagging to make C_C++ memory safe*. https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/Hardware%20Memory%20Tagging%20to%20make%20C_C++%20memory%20safe(r)%20-%20iSecCon%202018.pdf. Accessed: 2020-11-03.

[28] *SPEC CPU 2017 Compilation flags*. https://www.spec.org/cpu2017/flags/gcc.2018-02-16.html#user_F-fno-strict-aliasing. Accessed: 2020-11-22.

[29] Robert Swiecki. "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options". In: *URl: https://github. com/google/honggfuzz (visited on 06/21/2017)* (2017).

[30] *System76 Homepage*. https://system76.com/. Accessed: 2021-1-10.

[31] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. "Ramblr: Making Reassembly Great Again." In: *NDSS*. 2017.

[32] Shuai Wang, Pei Wang, and Dinghao Wu. "Reassembleable disassembling". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 627–642.

[33] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. "From hack to elaborate technique—a survey on binary rewriting". In: *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1–37.

[34] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios

P Kemerlis. "Egalito: Layout-Agnostic Binary Recompilation". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 133–147.

[35] M. Zalewski. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl/. Accessed: 2020-11-03.