❤ ARMore: Pushing Love Back Into Binaries ❤

Luca Di Bartolomeo
EPFL

Hossein Moghaddas
EPFL

Mathias Payer
EPFL

Abstract

Static rewriting enables late-state code changes (e.g., to add mitigations, to remove unnecessary code, or to instrument for code coverage) at low overhead in security-critical environments. Most research on static rewriting has so far focused on the x86 architecture. However, the prevalence and proliferation of ARM-based devices along with a large amount of personal data (e.g., health and sensor data) that they process calls for efficient introspection and analysis capabilities on the ARM platform. Addressing the unique challenges on aarch64, we introduce ARMore, the first efficient, robust, and heuristic-free static binary rewriter for arbitrary aarch64 binaries that produces reassembleable assembly. The key improvements introduced by ARMore make the recovery of indirect control flow an option rather than a necessity. Instead of crashing, the cost of an uncovered target only causes the small overhead of an additional branch. ARMore can rewrite binaries from different languages and compilers (even arbitrary hand-written assembly), both on PIC and non-PIC code, with or without symbols, including exception handling for C++ and Go binaries, and also including binaries with mixed data and text. ARMore is sound as it does not rely on any assumptions about the input binary. ARMore is also efficient: it does not employ any expensive dynamic translation techniques, incurring negligible overhead (<1% in our evaluated benchmarks). Our AFL++ coverage instrumentation pass enables fuzzing of closed-source aarch64 binaries at three times the speed compared to the state-of-the-art (AFL-QEMU), and we found 58 unique crashes in closed-source software. ARMore is the only static rewriter whose rewritten binaries correctly pass all SQLite3 and coreutils test cases and autopkgtest of 97.5% Debian packages.

❤ 1 ❤ Introduction

Binary rewriting is the process of modifying an executable without the need to recover its source code. Use cases include, for example, the insertion of hardening measures to mitigate the exploitation of vulnerabilities, enabling high-performance fuzz testing of closed-source software, profiling executables [3], or run-time patching [14]. Binary rewriters fall into two main categories: *dynamic* rewriters, that inject code into the target executable at run-time, and *static* ones, that transform the binary before execution. The first class trades speed for flexibility as any code is translated *on-the-fly*, i.e., right before it is executed which results in a significant slowdown (e.g., 3–5x for QEMU); the static approach instead has a much smaller footprint in overhead but is only applicable to binaries where static analysis (i.e., without feedback from the application's runtime behavior) is successful. Static rewriting is preferable in contexts where performance is critical (e.g., fuzzing) and necessary in production environments for security. Dynamic binary rewriters generally execute the translator in the same process as the translated program and often have regions marked writable and executable for their just-in-time translation, drastically increasing the attack surface.

While there is no shortage of static rewriters for the x86 architecture [10, 17–20, 34, 42–44, 49, 50], only a few support ARM [22,23,32,47]. Counterintuitively, ARM's fixed-size ISA introduces new challenges. Since 64-bit pointers cannot be encoded in 32-bit instructions, executables must "craft" addresses (e.g., using arithmetic expressions to build them dynamically). Precisely detecting such dynamic pointer constructions poses a major challenge for static rewriters. As the layout of code (and data) may change during the instrumentation, static binary translators must distinguish pointer types and data types and fully recover code pointers to adjust them to the new layout.

ARM binaries rely on *compressed jump tables*, where for each possible entry in the jump table, only the offset of each case is stored (usually, only 1 or 2 bytes), making jump tables particularly hard to statically distinguish from random data. A static binary rewriter must detect and adjust those offsets to accommodate the instrumentation that might change the distance from the base case. Even state-of-the-art techniques, such as *sliced microexecution* [13] or tools like, IDA Pro [4] fail to consistently distinguish jump table data in binaries without debug information. A static binary rewriter must detect and adjust jump table offsets to accommodate the instrumentation that might change the distance from the base case.

Most previously proposed solutions rely on heuristics like pattern matching [26] or bounded data-flow analysis [47] to follow each arithmetic operation to recover the set of possible pointer values. Heuristics incur imprecision or translation errors whenever they are incomplete. Another approach is to keep the original pointers intact by inserting instrumentation using *trampolines* [23] (incurring heavy performance cost) or translating pointers on-the-fly through hashmaps [12] (but thereby dropping support for unmodified shared libraries).

ARMore, our novel, near zero-overhead static binary rewriter for aarch64 binaries, enables complex and efficient instrumentation for closed-source binaries. ARMore leverages symbolization (creating reassembleable assembly as defined by Uroboros [43]) to reconstruct control flow after the insertion of instrumentation.

Different from other rewriters, ARMore does not rely on heuristics, as it only employs sound techniques. For the recovery of pointers, we leverage *layout replication*, which preserves the original address space of the binary by adding instrumentation through a new code section, and we introduce the *rebound table*, a new solution to efficiently and seamlessly translate addresses through direct branches.

The combination of layout replication and rebound table fundamentally changes the approach to static binary translation. Without it, any non-translated code pointer results in an unrecoverable (and hard to detect) fault. ARMore ensures that, by default, any code pointer access resolves correctly at the cost of an extra branch. This extra branch can be removed if the pointer target is recovered and translated, improving performance. Additionally, this technique supports arbitrary pointer arithmetic, solving the key issue with jump tables, similarly making jump table detection an optimization opportunity instead of a necessity for the correct execution of the rewritten binary. The drawback of the rebound table is that it can only be used in binaries where the text section is smaller than the range of a non-conditional branch (256 MB).

Another key improvement is the support of data inside text sections. This is a major challenge for static rewriters, as determining if data at a memory address is treated as instructions or data (or both!) is generally undecidable. If the target architecture is at least ARMv8-A (with support for XOM, executable-only memory), ARMore uses hardware assistance to reliably detect and adjust accesses to data inside code sections.

We demonstrate ARMore's robustness by demonstrating that rewritten binaries preserve the original behavior. Our evaluation involves rewriting binaries from 239 Debian packages (including C++ and Go binaries), and running the relevant `autopkgtest` testsuite of each package, passing 97.5% tests. We also run the complete testsuite of the SQLite3 and coreutils rewritten binaries (passing all tests in both). We demonstrate ARMore's versatility through a series of instrumentation passes: security mitigations (control-flow integrity), profiling (AddressSanitizer), and fuzzing (coverage instrumentation).

Standard CPU benchmarks commonly used to evaluate binary rewriters show that ARMore incurs only about 1.0% overhead in our evaluation benchmarks. The benchmarks include large programs such as `gcc` and `perl` and verify the output of the rewritten programs, demonstrating that our implementation is correct and scales well to complex COTS binaries. ARMore includes an optional pass to enable generic support for arbitrary stack unwinding (i.e., C++ exceptions) that uses *call emulation* [50]. With call emulation enabled, the overhead on our benchmarks is around 10%. The coverage information instrumentation of ARMore enables fuzzing of closed-source aarch64 binaries at high speeds (around 3 times faster than the state-of-the-art AFL-QEMU). We used this pass to fuzz the Nvidia CUDA toolkit, unveiling 58 unique crashes. Finally, to show the performance of our zero-cost instrumentation, we implement an AddressSanitizer instrumentation pass similar to the one presented by Dinesh et al. [19]. Our implementation of AddressSanitizer is only 26% slower than the original source-based version, and almost an order of magnitude faster when compared to a dynamic instrumentation approach (Valgrind's memcheck).

Our core contributions are:

- A new and heuristic-free approach of instrumenting aarch64 binaries containing arbitrary pointer arithmetic or data interleaved with code.
- A new mechanism, the rebound table, to recover from statically-unresolvable indirect control-flow transitions leveraging our insights for fixed-size ISAs.
- Safe-fallback mechanisms that enable sound (but not necessarily complete) optimization passes.
- The development of a fully-precise efficient aarch64 static binary rewriter that scales to large COTS binaries with support for arbitrarily large instrumentation. ARMore is open-source at `https://github.com/hexhive/retrowrite`
- A systematic evaluation of the performance overhead of the above instrumentation passes, along with a discussion of the limitations and real-world applications.

2 Background

2.1 Dynamic Instrumentation

Dynamic binary rewriters modify and instrument the code of the target at runtime. The target binary is executed in a controlled environment side by side with the rewriter engine, which patches instructions and fixes references on-the-fly. The rewriter engine may leverage operating system primitives (`ptrace`) to control target execution. For performance, the rewriter engine typically leverages its own instrumentation runtime (e.g., Dynamo [11]) or implements a full featured virtual machine (e.g., STRATA [40]).

Dynamic approaches probe into the execution state of the running process to gather information such as the execution path and the contents of registers. Such information can help rewrite some of the more complex mechanisms (e.g., indirect branches), but comes at a performance cost [29, 36]. Dynamic instrumentation is—without mitigation—unfit for security-critical environments, as it increases the number of dependencies of a binary and creates additional attack surface (e.g., through mapped read-write-execute sections that are required for efficient instrumentation).

❤ 2.2 ❤ Static Instrumentation

Static rewriters process the target binary *before* execution, and produce a new binary after processing all instrumentation that adds, removes, or modifies code. The overhead introduced is low, and execution speeds are comparable to compile-time instrumentation [19].

One of the main challenges for static instrumentation is resolving pointer targets. Due to aliasing, the target for references is unknown during translation. Without runtime information, static rewriters need to rely on complex static analysis, which is inherently imprecise and often resorts to heuristics. Consequently, static rewriting cannot support packed binaries or self-modifying code. However, since they insert instrumentation before the target execution, they can leverage more advanced and computationally expensive strategies to insert instrumentation (e.g., value set analysis [15]).

❤ 2.3 ❤ Static Binary Rewriting Techniques

Briefly summarized, the following are most prominent techniques that rewriters use are the following:

Trampolines: At every instrumentation point, the rewriter replaces the corresponding instruction with a branch that redirects execution to the instrumentation. This is one of the simplest methods, but it causes substantial performance overhead due to the cost of two extra branches for each instrumented instruction.

Direct: At each instrumentation point, code is either overwritten or shifted to make space for the instrumentation.

Lifting: All code in the binary is lifted to a “high level” Intermediate Representation (IR) language similar to the one used in compilers. The instrumentation is applied to the IR which is finally compiled to a new executable. *Symbolization*: Binary code is translated into reassemblable assembly text files. Uroboros [43] introduced and Ramblr [42]/RetroWrite [19] refined this technique. Symbolization transforms all referenced constants in the executable (both in the code and data sections, including relative branches) to assembly labels, so that pointers and control flow resolve correctly even after weaving new instructions into the code. After symbolization, many existing tools can be applied to insert instrumentation or analyze the symbolized assembly.

A more comprehensive study of all the different techniques can be found in a recent survey about binary rewriting by Wenzl et al. [46].

❤ 2.4 ❤ ARM Pointer Construction

Due to the 4-byte wide fixed size ISA, a single aarch64 instruction cannot encode a full pointer (e.g., on 64-bit Linux, each pointer is encoded as 64-bit type with 48-bit actively used). One approach to solve this discrepancy is to use literal pools. Namely, storing pointers as data at compile time and loading them into a register using a PC-relative `ldr` instruction. However, this adds the cost of a memory operation each time a pointer needs to be loaded into a register.

The alternative relies on building a pointer at runtime in two or more instructions, through a mechanism we refer to as *pointer construction* from now on. A PC-relative pointer construction starts with an `adrp`, which loads the address of a page into a register (with a granularity of 4KB and a maximum range of 4GB relative from the PC).

Basic arithmetic operations such as `add` or `sub` set the last 12 bits of an address (the offset inside a 4KB page), but there is no particular rule that compilers follow. Statically recovering pointer constructions can be challenging, as compiler optimizations may mix and spread pointer constructions (especially when multiple pointers are built simultaneously). Listing 1, Listing 2, and Listing 3 show some examples of how a compiler could optimize pointer constructions. Existing aarch64 static rewriters (e.g., Egalito [47], ICFGP [32], Ddisasm [22]) exclusively rely on data flow to recover pointers. Although it may work on an extensive range of binaries, it is inherently imprecise and cannot cover all edge cases of real-world binaries.

```
adrp x0, 0xab0000
add x1, x0, 0x100 ; built pointer 0xab100
ldr x2, [x0, 0x200] ; built pointer 0xab200
add x0, x0, 0x80 ; built pointer 0xab080
```

Listing 1: Multiple pointers built from one `adrp` instruction.

```
adrp x0, 0xab0000
mov x1, x0
add x1, x1, 0x100 ; built pointer 0xab0100
```

Listing 2: Changing register during pointer construction.

```
adrp x0, 0xab0000
str x0, [sp, -16]...ldr x3, [sp, 16]
add x3, x3, 0x200 ; built pointer 0xab200
```

Listing 3: Base page register stack saving.

2.5 ARM Executable-Only Pages

Execute-only memory (XOM) is a firmware protection technique available since ARMv8.1 with the goal of stopping a potentially malicious third party from reverse engineering binary blobs or scanning for ROP gadgets. When a particular page of memory is protected with XOM, its permission bits only show as executable (`-x`) and that page is readable only through instruction fetches.

Such memory protection was added to mainstream Linux in 2016 [30] but later removed in 2020 [31] due to it being vulnerable to PAN (Privileged Access Never) bypass. Support for XOM in Linux was later restored in 2021 [33], since the hardware bug was fixed in ARMv8.7 with the introduction of EPAN (Enhanced Privileged Access Never).

3 Challenges and Key Insights

To design a static binary rewriter, several challenges must be addressed, which we review along with our survey of previous work. We order the challenges by their importance for a *robust* and *correct* rewriting engine, and highlight the insights and trade-offs made in the development of ARMore.

3.1 Challenges

C1) Static pointer detection: Binaries often contain hard-coded addresses (e.g., in data sections) that are used to access global variables, imports, and functions. Those addresses need to be detected and relocated to preserve correctness. Due to the lack of syntactic difference between a pointer and an integer, distinguishing references from scalars is undecidable in general [25].

C2) Pointer construction detection: On aarch64, compilers dynamically build most references through a series of at least two arithmetic instructions. Identifying those instructions is fundamental for the correctness of the rewriting process, as data accesses and indirect control-flow transfers frequently use dynamically computed addresses. Static rewriters struggle to reconstruct pointers since the analysis required to accurately emulate the arithmetic operations to rebuild the final value of the pointer becomes prohibitively complicated in large functions. For this reason, static translators often need to fall back to error-prone heuristics.

C3) Relative offsets and jump tables (Symbol-Symbol): Compilers use jump tables to implement switches or optimize a series of conditional branches. While normally jump tables are composed of a sequence of static addresses, sometimes they are a list of 1 or 2-byte offsets from the jump table's base case. Like for *C1*, precisely differentiating those offsets from random data is undecidable. Locating the sequence of offsets in memory is not sufficient: static rewriters must also infer the maximum number of cases supported by a given jump table to determine the length of the jump table. Typically, to detect jump table targets, rewriters assume that compilers leverage a particular set of patterns to access jump tables and look for those in the code sections; however, this may result in incomplete coverage and runtime errors. Even advanced analysis techniques (e.g., sliced microexecution [13]) and state-of-the-art static analysis tools (e.g., IDA Pro [4]) cannot recover all jump tables correctly [13].

C4) (Non) Position-independent binaries: Position-independent executables ease some of the challenges of static analysis due to the presence of relocations that mark code pointers. The absence of full relocation information in non-PIE binaries makes code pointer detection challenging, and this is why many rewriters do not support non-PIE binaries [19, 47].

C5) Stripped binaries: The lack of defined function boundaries in stripped binaries complicates the disassembly. Standard disassembly techniques are not reliable enough, and this calls for more advanced techniques to support stripped binaries.

C6) Data mixed with code: Previous studies showed that on x86 distinguishing data from code is generally undecidable [45] as the static analysis must determine that a location can never be decoded as an instruction by fully recovering all possible control flows that could reach this location. Conversely, the same logic applies to aarch64. While data inside the text section is rare in x86 (only used in heavily optimized handwritten assembly and the now rarely used Intel C compiler), it is common on aarch64 due to compilers making use of literal pools. With a few exceptions [12], most static rewriters do not support binaries that embed data inside their text sections; they instead rely on broad heuristics that work on most binaries but do not guarantee correctness (e.g., pattern-based function boundary detection, incomplete data-flow analysis).

C7) Stack-unwinding control-flow mechanisms (C++, Go): C++ exceptions unwind the stack and, for each frame, determine the correct exception handling routine by checking the value of the return address. Supporting exception handling is a major challenge for static rewriters, as parsing the DWARF metadata for stack unwinding and Language-Specific Data Area (LSDA) exception tables requires covering numerous edge cases with high engineering complexity. Similarly, Go's garbage collection routines traverse stack frames and need special handling to be supported.

C8) Function pointers passed to external libraries (callbacks): Binaries might call external libraries with function pointers in the arguments as callbacks. Rewriters must correctly detect such pointer construction and rewrite those pointers accordingly so that the (potentially unmodified) target library receives the intended callback function as argument.

C9) Instrumentation coverage: Rewriters must be able to insert instrumentation at arbitrary locations in the binary. Due to design constraints, some static rewriters cannot instrument all instructions in an executable. For example, E9Patch [20] cannot instrument single-byte instructions.

C10) Instrumentation size: Supporting arbitrarily large instrumentation is challenging, as many instructions and binary constructs have a limited range and require special handling: on aarch64, the `tbz` conditional branch has only 32KB of range. Jump table offsets are often stored in multiples of instructions (4 bytes) using a single byte, limiting the distance from the base case to 256 instructions.

❤ 3.2 ❤ Limitations of Existing Techniques

Table 1 puts state-of-the-art tools and the aforementioned challenges into perspective. In the following, we discuss this summary in more detail.

E9Patch [20] makes use of various techniques to patch instructions and inserts trampolines at each instrumentation location. Since instructions are never moved/inserted, this approach trivially solves almost all challenges except two: instrumentation coverage, as some locations such as function entries cannot always be instrumented [32] (*C9*), and distinguishing data mixed with code (*C6*). However, it incurs high overhead: empty instrumentation at every basic block causes more than 100% overhead.

Multiverse [12] is the first x86 rewriter to avoid using heuristics. Indirect control flow is rewritten by querying a mapping table between original and new addresses at every indirect jump. Data pointers are left unmodified since the original sections of the binary remain in the same place as they were. Interleaving data and code is supported by keeping a readable-only copy of the original text. It supports C++ exceptions, but since pointers passed as callbacks to libraries are not rewritten, Multiverse requires rewriting of libraries that might be loaded by the binary (*C8*), and cannot support other generic stack unwinding mechanisms such as Go (*C7*). Finally, the frequent querying of the mapping table incurs considerable overhead of 30–60% with empty instrumentation.

Egalito [47] is a binary *recompiler* that lifts binaries to a custom IR to insert instrumentation, incurring low overhead (~0.5%). It supports stripped binaries but relies on heuristics such as data-flow and pattern matching to detect jump tables (*C3*), to recover pointer constructions (*C2*), and to approximate function boundaries (*C5*). Egalito supports neither non-PIE (*C4*) nor stack unwinding mechanisms as used in C++ and Go (*C7*).

StochFuzz [50] is designed specifically for fuzzing applications. It makes clever use of the stochastic nature of fuzzing to self-correct their rewriting. It supports almost any construct without heuristics, but it works only inside a fuzzing environment, as there is no guarantee that incremental rewriting will ever terminate. Since the continuous execution of the binary is part of the rewriting process, we did not assign a value in the "overhead" column of Table 1. The only drawback is the use of probabilistic rules upon distinguishing instructions from data (*C6*).

Repica [23] uses a technique similar to reassembly, and relative jumps are carefully adjusted to keep control-flow intact. Jump tables and indirect pointers are detected through backward slicing, which is imprecise and might lead to failures in pointer construction recovery (*C2*) and jump table symbolization (*C3*). It also does not support non-PIE (*C4*) or C++ binaries (*C7*).

Incremental CFG patching [32] is based on Dyninst [14], and relies on using trampolines to relocate functions to an instrumented area with low overhead. ICFGP employs failure analysis modes to prevent crashes in the case of an undetected jump table (*C3*) by not allowing to instrument the function that contains the misdetection. However, this leads to lower instrumentation coverage (*C9*) since not all functions can be instrumented. Pointer construction detection is still based on heuristics and may lead to crashes (*C2*). ICFGP is one of the few tools that support C++ and Go binaries (*C7*). It uses a custom implementation of libunwind and of the Go runtime that adjusts pointers on the stack, avoiding having to deal with DWARF, but with the drawback of having to manually adapt unwinding instrumentation to support new unwinding runtimes.

Ddisasm [22] uses the reassembly technique and relies on a large set of reassembly heuristics to detect jump table constructs and similar patterns. These heuristics have inherent uncertainty and may fail in *C2* and *C3* (as we noticed in our evaluation). Ddisasm supports C++ exception rewriting, but not other stack unwinding mechanisms such as Go (*C7*) [39]. Furthermore, Ddisasm may also fail *C10*, since it cannot expand the range of a jump table; in the case of a single-byte range jump table, this can prove to be a very limiting restriction.

❤ 3.3 ❤ Our Technique

After enumerating and discussing the above challenges, we introduce ARMore. Previous approaches relied on either imprecise techniques (e.g., data flow) or precise but expensive ones (e.g., dynamic translation, trampolines).

Our key insight is the following: given that all pointer arithmetic must begin by either an `adrp`/`adr`, we can leverage a combination of two techniques (*layout replication* and *rebound table*), to rewrite all pointer arithmetic (including pointer constructions (*C2*), jump tables (*C3*) and callbacks (*C8*)) by rewriting *only* the first instruction (`adrp`/`adr`). With hardware support, we detect loads to data inside text (*C6*). As a result, ARMore does not rely on any expensive dynamic translation or imprecise data-flow techniques.

First, we delineate the layout replication and rebound table techniques. Then, we explain how ARMore leverages them to address pointer-related challenges (*C1*, *C2*, *C3*, *C8*). Finally, we illustrate our approach to solve the rest of the challenges, thereby enabling support for arbitrary binary rewriting with arbitrary instrumentation for aarch64.

❤ 3.3.3 ❤ Layout Replication

ARMore enforces the original virtual address space layout of the binary. The idea of fixing objects in memory was intro-

Table 1: Comparison of recent static binary rewriters based on: *C1*) No-heuristics static pointer detection *C2*) No-heuristics pointer construction detection *C3*) No-heuristics jump tables *C4*) Full PIE/No-PIE support *C5*) Stripped binaries support *C6*) No-heuristics Mixed data/code *C7*) Stack unwinding (C++, Go) *C8*) External libraries *C9*) Full instrum. coverage *C10*) Unlimited instrum. size Fast) Negligible overhead (0-3%) SB) Stand-alone binary

				Ptr Arith.			Lack of metadata					Instrum.		
	Technique	Fast	SB	*C1*	*C2*	*C3*	*C4*	*C5*	*C6*	*C7*	*C8*	*C9*	*C10*	Archs
RetroWrite [19]	symbolization	✓	✓	✓							✓	✓	✓	x64
Repica [23]	symbolization	✓	✓	✓				✓			✓	✓	✓	aarch64
Egalito [47]	IR lifting	✓	✓	✓				✓			✓	✓	✓	aarch64, x64
DDisasm [22]	symbolization	✓	✓	✓			✓	✓		$*^1$	✓	✓		aarch64, x64
ICFGP [32]	trampolines	✓	✓	✓		✓	✓	✓		✓	✓	✓		aarch64, x64, ppc
StochFuzz [50]	stoch. rewriting	N/A^3		✓	✓	✓	✓	✓		✓	✓	✓	✓	x64
E9Patch [20]	trampolines		✓	✓	✓	✓	✓	✓		✓	✓		✓	x64
Multiverse [12]	recompilation		✓	✓	✓	✓	✓	✓	✓	$*^1$	$*^2$	✓	✓	x64
ARMore	symbolization	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	$*^4$	aarch64

^{1} : Support for C++, but not Go or other unwinding mechanisms.
^{2} : Requires rewriting of all libraries loaded by the binary.
^{3} : Rewriting may never terminate; use-case is exclusively fuzzing.
^{4} : Up to 256MB of text section

duced by SecondWrite [10] and is also used in Multiverse [12]. In this way, data pointers do not need to be rewritten, as they point to the same data after rewriting. Only code pointers need to be corrected to point to the new section that contains the instrumentation. To avoid using heuristics, Multiverse relies on instrumenting every indirect call with a lookup to a hashmap that would translate the original address to the instrumented one. This approach addresses multiple important challenges (*C1*, *C2*, *C3*, *C4*, *C5*, *C6*); however, it has two drawbacks: (1) It introduces prohibitive overhead (30-60%), and (2) Pointer callbacks passed to external libraries are not translated (*C8*). Multiverse solves the callback problem by rewriting all libraries that the binary might load, thereby inducing even more overhead.

Instead of using a hashmap to rewrite code pointers, ARMore avoids this unnecessary overhead. The core difference lies in how ARMore handles the original `.text` section: Multiverse changes its permissions to read-only to preserve it as data; ARMore substitutes it with the *rebound table* section. Another copy of the `.text` is kept read-only for supporting data interleaved with code (more details in Section 3.3.3). Figure 1 illustrates an example of ARMore's layout replication.

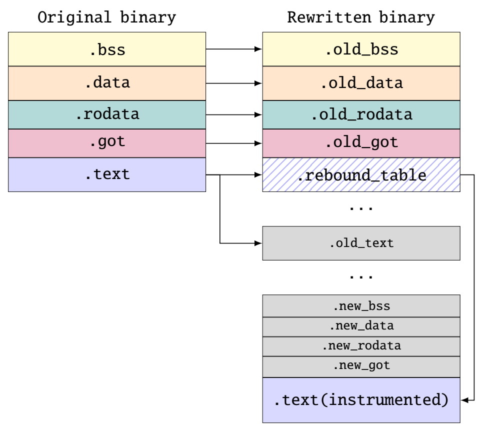
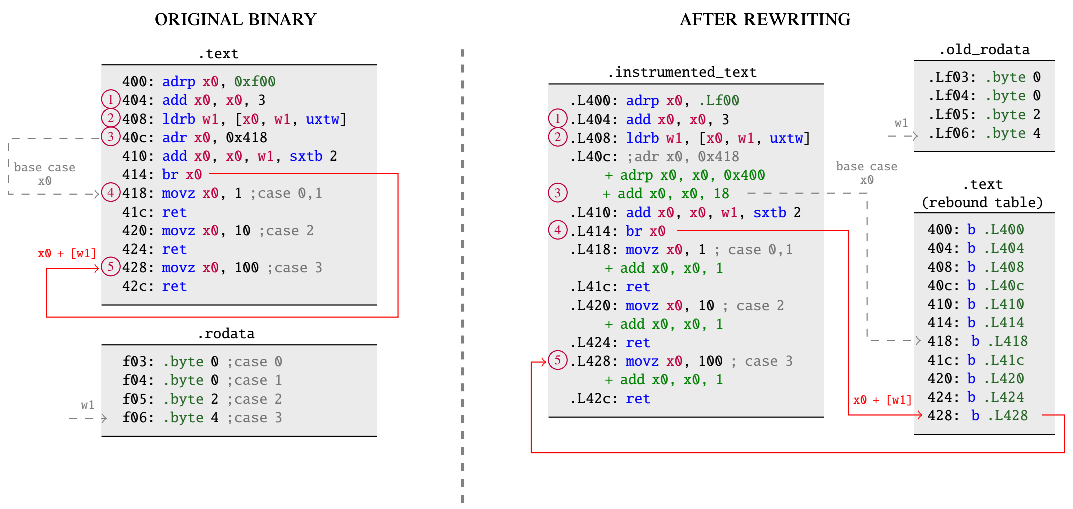


Figure 1: Layout replication: the left side shows the sections of the original binary. The right side shows the section layout of the rewritten binary.

❤ 3.3.3 ❤ Rebound Table

Static rewriting approaches cannot universally reliably resolve indirect control-flow transfers. An undetected pointer construction points to its original address in the original code section. As a result, a recovery mechanism is needed. However, previous approaches follow the classic idea of replacing the code section with expensive trap-based trampolines [35, 37, 50], which introduce noticeable overhead.

On the contrary, a major insight of ARMore exploits the fixed size ISA of aarch64 by establishing a one-to-one mapping from each original instruction to its counterpart in the instrumented section. In a fixed-size ISA such as aarch64, it becomes ingeniously feasible to overwrite each instruction in the code section without worrying about imprecise disassembly. In addition, control flow never ends up in the middle of an instruction, since the program counter must always be 4-byte aligned. Instead of overwriting the code with trap instructions, ARMore leverages direct branches whose destination in the instrumented code section is already known at link-time. Consequently, an undetected pointer construction always ends up in the rebound table from where the control flow is redirected to the corresponding target instruction. Algorithm 1 illustrates and Section 4.1 details ARMore's approach to reflow pointer constructions to target the rebound table.



```
.text
  400: adrp x0, 0xf00
① 404: add x0, x0, 3
② 408: ldrb w1, [x0, w1, uxtw]
③ 40c: adr x0, 0x418
  410: add x0, x0, w1, sxtb 2
  414: br x0
④ 418: movz x0, 1 ;case 0,1
  41c: ret
  420: movz x0, 10 ;case 2
  424: ret
⑤ 428: movz x0, 100 ;case 3
  42c: ret

.rodata
  f03: .byte 0 ;case 0
  f04: .byte 0 ;case 1
  f05: .byte 2 ;case 2
  f06: .byte 4 ;case 3
```

```
.instrumented_text
  .L400: adrp x0, .Lf00
① .L404: add x0, x0, 3
② .L408: ldrb w1, [x0, w1, uxtw]
  .L40c: ;adr x0, 0x418
      + adrp x0, x0, 0x400
③     + add x0, x0, 18
  .L410: add x0, x0, w1, sxtb 2
④ .L414: br x0
  .L418: movz x0, 1 ; case 0,1
      + add x0, x0, 1
  .L41c: ret
  .L420: movz x0, 10 ; case 2
      + add x0, x0, 1
  .L424: ret
⑤ .L428: movz x0, 100 ; case 3
      + add x0, x0, 1
  .L42c: ret

.old_rodata
  .Lf03: .byte 0
  .Lf04: .byte 0
  .Lf05: .byte 2
  .Lf06: .byte 4

.text
(rebound table)
  400: b .L400
  404: b .L404
  408: b .L408
  40c: b .L40c
  410: b .L410
  414: b .L414
  418: b .L418
  41c: b .L41c
  420: b .L420
  424: b .L424
  428: b .L428
```

Figure 2: This jump table example demonstrates the on-the-fly translation functionality of the rebound table. On the left, the original code of the binary. On the right, the binary after rewriting. Instrumentation is inserted during symbolization (the green instructions). The `adr` at `40c` is substituted with a pointer construction to the rebound table, and the offsets in `.old_rodata` remain valid as the indirect jump goes through the rebound table.

Figure 2 shows how the rebound table recovers an indirect control flow transfer (in this case, a jump table). In the figure, register `w1` contains the index number of the case of the jump table (in this example, case number three). The first two instructions of the jump table (step ①) build a pointer to the `.rodata` section where the offsets for each case is stored. During rewriting, the first `adrp` instruction is adjusted to point to the replicated layout (specifically, to `.old_rodata`, as shown in Figure 1). Then, the correct offset is fetched at step ② using register `w1` as the "case number" of the jump table. In step ③, the program builds a pointer to the jump table base case (where offsets are calculated from) and stores it in `x0`. A single `adr` instruction is sufficient since the pointer to be built is close to the program counter. During rewriting, the `adr` will be converted to an `adrp`+`add` pointer construction to be able to target the replicated layout (instead of the original text, it will now point to the rebound table, as shown in Figure 1). At step ④, the relative offset is added to the base case to obtain the address of the intended case of the jump table (case 3 in the example). After this step, register `x0` will point to `0x428` both before and after rewriting. In the rewritten binary, address `0x428` is inside the rebound table. It points to a branch to a label corresponding to the third case of the jump table (`.L428`). Finally, after an indirect branch, at step ⑤ control flow correctly reaches the third case in both the original and the rewritten binary.

In other words, the rebound table implicitly translates addresses on-the-fly, conceptionally similar to many other binary rewriters that use hashmaps [12, 16, 44, 49]. On the other hand, ARMore's rebound table translation comes at the cost of only one additional branch, instead of an expensive lookup through a dynamic structure. Furthermore, the rebound table allows the insertion of unlimited instrumentation (*C10*) everywhere in the code (*C9*). The range of jump table entries

Algorithm 1: Pointer construction rewriting

```
Input: Disassembly listing dis of the code section
Output: Disassembly outdis in which built pointers target the replicated layout
outdis ← { };
foreach instruction ins ∈ dis do
    if ins.opcode == adr then
        reg ← ins.operand[0];
        mem ← ins.operand[1];
        page ← mem & 0xfffffffffffff000;
        if mem ∈ .text then
            page ← rebound_table_start + (page − text_start);
        outdis ← outdis ∪ ["adrp reg, page"];
        outdis ← outdis ∪ ["add reg, reg, :lo12:mem"];
    else if ins.opcode == adrp then
        reg ← ins.operand[0];
        page ← ins.operand[1];
        if page ∈ .text then
            page ← rebound_table_start + (page − text_start);
        outdis ← outdis ∪ ["adrp reg, page"];
    else
        outdis ← outdis ∪ [ins];
```

is not a concern since jump table offsets are relative to the rebound table, where the original layout is preserved (i.e., the distance between jump table cases remains the same).

Due to the rebound table containing a branch for every instruction of the code section, the size of the text of the binary is tripled, affecting both disk usage and memory usage at runtime.

❤ 3.3.3 ❤ Pointer Arithmetic

Pointer construction comes in two forms: PC-relative (for PIE binaries) or static (non-PIE). Due to the combination of layout replication and rebound table, ARMore supports non-PIE static pointers out of the box (including any arithmetic performed over them), as they always target the replicated layout, where relative offsets are preserved even after instrumentation is inserted.

For PC-relative pointer constructions, ARMore forces dynamically computed pointers to target the replicated layout by symbolizing the initial `adrp`/`adr`—the prologue of a pointer construction—to point it to the same address on the replicated layout. The remaining instructions that fix the last 12 bits of a pointer remain untouched. All future calculations on that pointer are then referring to the replicated layout. It is important to note that `adrp` must precede each PC-relative pointer construction (see Appendix C).

This new way of dealing with dynamic pointer constructions is simple yet powerful: ARMore just needs to symbolize `adrp` instructions to correctly handle all pointers. As shown in our evaluation, previous aarch64 rewriters were limited by the complexity and imprecision of trying to emulate arithmetic operations to recover pointers (e.g., through dataflow).

This allows ARMore to support arbitrary pointer arithmetic, including static pointers (*C1*), pointer constructions (*C2*), and jump tables (*C3*). Even callbacks passed to libraries work as expected since they are translated through the rebound table after being dereferenced (*C8*).

3.3.3 Call Emulation

Despite having layout replication, all the code that ARMore produces is at a different location than the old `.text` (which is substituted by the rebound table). Return addresses on the stack refer to the instrumented part.

Call emulation is a widely used technique by rewriters in the past to preserve the correct return addresses [50]. It substitutes every `bl` and `blr` call instructions with a `adr`+`mov`+`b` sequence that puts the original return address in the link register (which, in our case, now points to the rebound table) and then uses a direct branch to a function instead of a call. This technique is safe; however, it has been often criticized for introducing noticeable overhead. We note that contrary to other static rewriters that use call emulation, ARMore does not need to instrument any `ret` instruction to re-adjust the return address, as the rebound table seamlessly translates it. In our evaluation, we record that using call emulation incurs 10.74% additional overhead. As mentioned in Section 3.4, while call emulation is enabled by default, disabling it is a reasonable optimization in most instances.

3.3.3 C++ and Go: Stack unwinding

To support C++'s exception handling (*C7*), we design two techniques. One is to re-use call emulation: by storing original return addresses on the stack, C++ exception metadata does not need to be updated. The alternative instead parses and rewrites the LSDA table entries that describe exception handling information. This second method does not introduce call emulation overhead but, as highlighted by previous work [32], it may be prone to error due to the variety and complexity of DWARF encodings. The user is free to choose between the two techniques.

To support Go garbage collection routines, ARMore uses call emulation. To avoid the call emulation overhead, some engineering effort is required to parse and rewrite Go's stack unwinding information.

3.3.3 Data Mixed With Text

Related work keeps a readable-only copy of the original text sections at the original virtual address [10, 12, 44, 49] to support interleaved data and code. As ARMore places the rebound table at the address of the original text section, it cannot rely on this approach. We instead introduce an alternative approach that leverages XOM (executable-only pages, available from ARMv8.1), reintroduced in the Linux kernel as of 2021 [33] With XOM, pages marked as executable-only (`-x`) are exclusively readable for instruction fetches. We mark the rebound table as executable-only, and whenever the program tries to access data inside the text region, the fetch will trigger a segmentation fault. ARMore catches the segfault through a signal handler. The handler returns the correct value by reading it from a copy of the original text section (`.old_text` in Figure 1). Assuming that data inside text is rare, this approach introduces low overhead even if the overhead per access is relatively high due to the involved trap.

3.3.3 Stripped and PIE/non-PIE Binaries

While rewriting non-PIE binaries is a challenge for other rewriters [19, 47], ARMore does not need to detect pointers and supports both PIC and non-PIC code through layout replication (*C4*).

Thanks to the fixed-size ISA, it suffices to linearly disassemble all the code sections to rewrite stripped binaries without heuristics. Since ARMore does not depend on function boundary detection, no particular technique is required to address *C5*.

3.4 Heuristics as an Opportunity

Another advantage of our approach is that the rebound table introduces an *opportunity for sound (but not necessarily complete) optimizations*. Any code pointer that can be correctly recovered through data-flow analysis can be adjusted to point to the translated code section. The rebound table serves as a catch-all fallback that adds very low overhead in the case a pointer escapes detection. The key difference to prior approaches is that the price of a missed pointer construction is not a terminal fault but simply the cost of going through an additional branch: *imprecision is no longer a correctness issue, but a tiny performance hit.*

One of the heuristics we implemented data-flow analysis to detect common jump table patterns. We update the pointer to the base case and the offsets that store the distance from the base case in `.rodata` to avoid the extra branch through the rebound table. Figure C.1 illustrates an example of a jump table that needs to be rewritten because of instrumentation inserted between its cases (in green). The example shows how the cost of failing to detect the jump table is just an additional jump instruction instead of a crash like in other binary rewriters.

Another possible optimization pass is to disable call emulation for functions that do not use alternative ways of reading the program counter (see Appendix C). Note that for binaries generated by well-behaved compilers, this optimization is sound for all functions.

4 Implementation

ARMore forks RetroWrite [19] and adds about 3,000 lines of Python code to implement the *rebound table* and *layout replication* techniques, along with support for C++ binaries, Go binaries, stripped binaries, data inside text, non-PIE binaries, and aarch64-specific pointer analyses.

RetroWrite provides a reasonable baseline for reassembly and we reuse the existing structure and glue code to parse ELF information. The code depends only on the libraries `archinfo`, `elftools`, and `capstone`.

4.1 Fixing the Address Space

To exactly replicate the layout of the original binary, ARMore requires precise placement of the individual ELF sections. Our process creates scripts that leverage `ld`'s command line flags such as `-section-start` to specify for each section the virtual address it must be placed at. Some of the sections are renamed (e.g., from "`.got`" to "`.old_got`"), to avoid some linker-internal heuristics that would move or modify them. In addition, we create stubs for these renamed sections to refer to new functionality. For example, using the AddressSanitizer instrumentation pass requires additional imports to be stored in the `.got` section. By using the name `.old_got` the linker adds the imports in the new `.got` stub, without modifying the replicated `.old_got`. The *old* sections have the same permissions and behavior as their respective originals.

Hardcoded pointers (e.g., literal pools, `.got` entries) in non-PIE binaries do not need to be symbolized, as they will already refer to the copied sections (e.g., `.old_got`). Hardcoded pointers in PIE binaries instead need to be symbolized. Notably, each pointer in a position-independent executable is accompanied by a corresponding relocation. ARMore parses relocation information in a binary and emulates the behavior of a dynamic loader when applying relocation rules, replacing pointers with assembly labels that point into the replicated layout. Instead, to force dynamically computed PC-relative pointers to target the replicated layout, Algorithm 1 illustrates our approach. ARMore symbolizes the initial `adrp`—the prologue of a pointer construction—to make it point to the same address in the replicated layout. The rest of the instructions that fix the last 12 bits of a pointer remain untouched since offsets are preserved on the replicated layout. Similarly, each `adr` instruction is substituted with a symbolized `adrp`+`add` pointer construction, since the replicated layout is usually out of the ± 1 MB range of the `adr` instruction. Given that on aarch64 PC-relative pointers are built either through an `adrp` pointer construction or through a simple `adr` instruction, every single dynamically computed PC-relative pointer is symbolized correctly, including jump tables and callbacks.

The rebound table leverages a linear sequence of aarch64 branch instructions, each of which redirects to the corresponding translated address in the new instrumented code section. Figure C.1 provides an example for a rebound table in the output of ARMore.

4.2 Support of Data Mixed With Text

Since the linker does not allow to define sections as executable-only through the assembly directive `.section .rebound_table, "x"`, ARMore makes the text sections executable-only by instrumenting `_start` with a call to `mprotect`. A call to the `sigaction` later registers the segmentation fault handler. The handler takes care of catching loads to executable-only regions, reading the relevant correct value from `.old_text`, and restoring the execution context as if the load never triggered a memory violation. Finally, ARMore installs seccomp filters to prevent the application from registering and overwriting our signal handler; ARMore keeps track of user-registered segfault handlers and calls them accordingly from its own handler.

4.3 Instrumentation Passes

When inserting instrumentation, ARMore uses any free registers and spills occupied ones onto the stack. To determine the set of free registers at any point in the binary, we perform a sound over-approximative live registers analysis. The following instrumentation passes are currently implemented in ARMore:

Coverage Information: this pass adds AFL-compatible instrumentation at every basic block location, in a similar manner to what `afl-gcc` [6] does on source code. We also provide an AFL++ [21] style forkserver implementation to avoid the overhead of an `exec` syscall for each run.

Control-Flow Integrity (CFI) [9]: protects the program against control flow hijacking by checking the runtime target of indirect function calls. ARMore's CFI instrumentation pass protects forward edges by restricting indirect call targets to function entries. Furthermore, backward edges are checked by leveraging ARMv8-A pointer authentication to implement a safe version of stack canaries as proposed by Liljestrand et al. [28]. Our instrumentation marks the function beginnings and relies on a set of function addresses in the binary which is either extracted from the ELF symbol table or can be provided through an external analysis that detects function headers.

AddressSanitizer: The ASan instrumentation pass is similar to the original ASan compiler pass [41]. RetroWrite [19] similarly implements a retrofitted binary ASan pass. We adjust the pass for aarch64 and enable compatibility with already pre-instrumented libraries. ASan is a perfect candidate to test the speed and scalability of ARMore due to the instrumentation pass' size and complexity. Binary ASan instruments every memory operation checking if the target address is poisoned (except for stack and global variable accesses, due to the lack of bounds information without source code).

It is important to note that, due to the presence of limited-range short jumps in aarch64 (e.g., `cbz, tbz`), ARMore substitutes them with longer jumps if it detects that the amount of instrumentation inserted would make their target out of reach.

5 Evaluation

The evaluation is aimed at answering the following three high-level questions about ARMore:

- **Q1:** Does it preserve the original behavior of binaries? (*correctness*)
- **Q2:** Does it scale to *real-world*, complex targets? (*robustness*)
- **Q3:** Is the instrumentation overhead of ARMore competitive with source-based instrumentation passes? (*performance*)

To address the above questions, we compare ARMore on SPEC CPU2017 in different configurations, we rewrite 239 Debian packages, run rewritten SQLite and coreutils binaries through their respective large test suites, study ASan and coverage instrumentation, fuzz real world software finding 58 bugs (from 90 crashes), and leverage ARMore to hot patch buggy applications. In total, we rewrite 486MB of code adding up to over 121 million instructions.

Our benchmark machine is a 2.4 GHz APM X-GENE A57 with 64GB of DDR3 memory and a Micron M500 M.2 disk, provided by the Cloudlab (`https://cloudlab.us/`) academic datacenter. For the fuzzing experiment, we used a Macbook Air M1 running an Ubuntu 20.04 virtual machine.

5.1 SQLite, Coreutils, Debian (Q1, Q2)

To estimate the correctness of ARMore, we leverage two extensive test suites of open-source projects (SQLite and Coreutils). To establish the robustness, we download and rewrite a multitude of real-world binaries from the Debian repositories, running their relevant `autopkgtest`. For both of those benchmarks, we symbolize and reassemble the binaries without inserting instrumentation, with call emulation enabled.

5.1.1 Open Source Test Suites

We use the SQLite test suite (which has 100% branch coverage [5]) and the coreutils test suite. The two test suites combined use more than 30 binaries and run a total of over two million extensive different tests. Rewritten binaries by ARMore passed all tests in both test suites.

5.1.1 Debian Packages

Many Debian packages include an `autopkgtest` feature through which package maintainers can verify that package binaries are built correctly and they pass the testsuite that comes with their source code. We decided to test ARMore's robustness by running those testsuites against the rewritten version of said binaries (with empty instrumentation).

Our evaluation harness automatically downloads and extracts `deb` packages, rewrites the relevant binaries, and runs the included testsuite with `autopkgtest`. We let the script run for 24 hours. We found out that the vast majority of packages do not contain tests: of the over 10,000 packages downloaded, all but 239 packages had to be skipped due to lack of tests, missing binaries, or outdated dependencies. Out of the 239 testsuites that were run, 232 passed correctly all tests, six of them failed to link due to unmet/outdated dependencies and one failed to rewrite. We manually inspected the reason for the failure in rewriting, and we found that the problem was caused by one of ARMore's dependencies (PyElfTools) that crashed because of an unsupported DWARF instruction.

Out of the 232 packages that passed all tests, 76 were in C, 45 were in Go, 18 were in C++, one in D, one in Fortran, one in Lua, and one in Ocaml. We determined the language they were implemented in by looking at the `implemented-in` and `Built-Using` tags in the `control` file of each package. The remaining 89 packages that passed did not have such tags. We manually inspected some of them and determined that the vast majority of the untagged packages are in C/C++.

In summary, 6 failures out of 233 testsuites gives ARMore a 97.5% pass rate on average. We are confident that all the failures are due to implementation bugs in our prototype of ARMore and could be fixed with some engineering effort.

A similar evaluation was conducted by Egalito (although limited to C/C++ packages). Their reported accuracy was 82.6%, i.e., 19 failures out of 109 [47].

5.2 Reassembly: SPEC CPU2017 (Q1, Q3)

The SPEC CPU2017 test suite is a common compiler benchmark that consists of a variety of diverse reasonably large applications. SPEC CPU2017 self-verifies its results to ensure correct computation. Our system runs Ubuntu 18.04 which comes with gcc 7.5.0 by default, Appendix A details our full setup.

To measure the overhead introduced by the rebound table, we benchmark the rewritten binaries by ARMore without instrumentation, and compare the runtime against the original

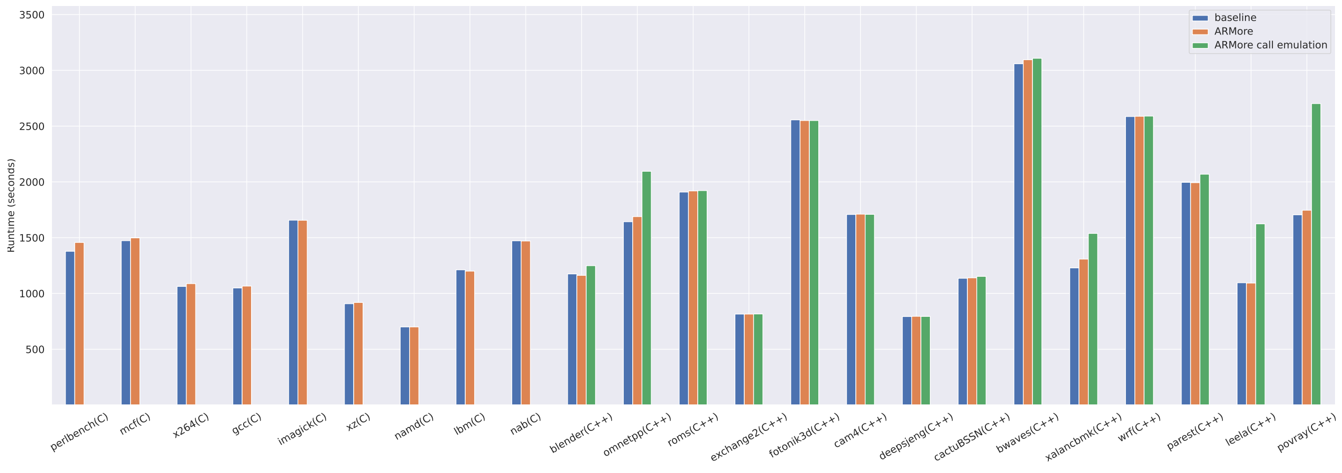


Figure 3: Benchmark runtime on SPEC CPU2017.

binaries. To be as conservative as possible and measure the worst-case rebound table overhead, we use ARMore with heuristic optimizations (e.g., jump table detection) turned off.

For the C++ binaries, we also measure the overhead of enabling call emulation (because of a failure in table exception information parsing). Notice that for C binaries, call emulation is not necessary, so we kept it turned off.

Figure 3 shows the result of the experiment. The average overhead introduced by the rebound table is 0.99%. The average overhead introduced by call emulation (calculated only on C++ binaries) is, on average, 10.74%.

The high variance of the overhead of call emulation is due to the difference in the number of calls each benchmark performs. To verify this, we instrument each benchmark binary to count the number of indirect branches (`br`), direct calls (`bl`) and indirect calls (`blr`) it performed during its execution, and run the SPEC CPU2017 suite again. Table B.1 shows the totals of those counts. Notably, C++ benchmarks that executed a high number of calls (both direct and indirect) are most affected by the call emulation overhead (`povray`, `omnetpp`, `leela`).

We conclude that ARMore introduces negligible overhead, yet maintains correctness as the output of the binaries is verified by the SPEC CPU2017 suite. We would like to note that while the C++ call emulation adds considerable overhead, it should be used only in the edge cases where exception information recovery fails.

5.3 Case Study I: Address Sanitization

We now evaluate ARMore in how well our binary address sanitization pass compares to source-based address sanitization. Address sanitization is a prime example of a heavy-weight pass that requires extensive instrumentation.

Similar to Section 5.2 we benchmark this pass on the SPEC CPU2017 binaries, comparing the results to a dynamic rewriter (Valgrind's memcheck), since we are not aware of other ARM static rewriters supporting ASan. The baseline for this comparison is compiler-based ASan (using the `-fsanitize=address` flag). We also run the same benchmark without the register savings optimization to measure its impact on such a heavy-weight instrumentation pass. On average, ARMore instrumentation pass is 26% slower than the baseline. Without register savings, it is 60% slower than the baseline. Meanwhile, Valgrind is almost 725% slower than the baseline.

Note that this is not a completely fair comparison, since ARMore's ASan and Valgrind's memcheck instrumentation pass cannot instrument stack allocations due to the inability to infer the layout of stack frames. In comparison to a compiler pass, they overestimate the number of memory operations that need to be instrumented and result in higher register pressure.

5.4 Case Study II: Coverage Information

To evaluate the effectiveness of ARMore's AFL-compatible coverage instrumentation pass, we compare it with current alternatives to fuzz aarch64 targets.

We evaluate ARMore against AFL-QEMU [48] on its default configuration, using source-based coverage instrumentation provided by the AFL++ [21] compiler as a baseline. We used Magma [24] as our benchmark as it was the only one readily available for ARM. Appendix B details our configuration.

Following established fuzzing evaluation guidelines [27], each target was executed five times for 24 hours each. The box plot in Figure 4 presents the throughput of the three fuzzers: ARMore fuzzes code about 3 times faster than AFL-QEMU, on average. Compared to AFL++ (compiler-based instrumentation), ARMore is around 25% slower.

However, the number of executions per second alone does not yield a complete picture, since inaccurate insertion of instrumentation could lead to higher throughput at the cost of fewer bugs found. For this reason, we verified that the coverage growth of using ARMore to fuzz was similar to the other two

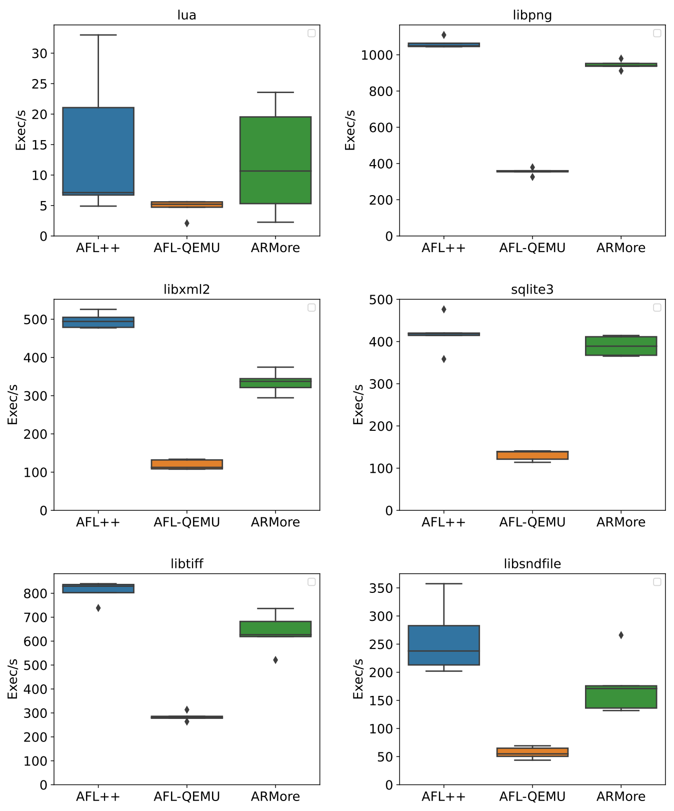


Figure 4: Executions per second over 5 runs of 24 hours on the MAGMA fuzzing benchmark. Higher is better.

Format	Version	Total binaries	Frequency
Ubuntu Server	18.04.6	129	0%
Ubuntu Server	20.04.5	901	0.0035%
Ubuntu Server	22.04.2	748	0.0047%
Ubuntu docker	18.04.6	85	0%
Ubuntu docker	20.04.5	300	0%
Ubuntu docker	22.04.2	292	0%

Table 2: Estimated data-inside-text frequency across all binaries in the `/bin` folder of stock Ubuntu distributions.

AFL-based fuzzers (AFL-QEMU and AFL++) in Figure B.1.

ARMore's coverage information instrumentation is evidently a viable alternative to AFL-QEMU and approaches the performance of source-based AFL++.

5.5 Case Study III: Fuzzing Real World Closed-source Software

To show the effectiveness of the fuzzing speed that ARMore enables, we ran a campaign targeting real-world proprietary binaries. We evaluated the Nvidia CUDA toolkit for Linux, a set of utilities to compile, debug and inspect CUDA applications.

After 10 hours of fuzzing for each binary, we found 87 crashes in `cuobjdump` and three crashes in `ptxas`. To ensure that our instrumentation did not cause the crashes, we verified that each crash was reproducible on the original binary. The flexibility of ARMore helped us discover the severity of the bugs: by instrumenting the target binaries with ASan, we discovered that there are 33 unique heap-buffer-overflows and 11 unique use-after-free bugs amongst the crashes; the remaining crashes are null pointer dereferences. By classifying each crash by the program counter of the crash site, we approximate that our fuzzing campaign found 58 unique disambiguated crashes. We responsibly disclosed our findings to the vendor and are waiting for a response from them regarding their security implications.

5.6 Case Study IV: Binary Patching

Defenders must quickly patch exploitable bugs to stop attacks, but the long lead times for developing, testing, and deploying patches put defenders at a disadvantage. Binary patching provides a fast alternative, enabling analysts to develop stop-gap solutions by adding code directly to the binary.

However, ARMore's symbolization enables the patching of vulnerable ELF binaries without brittle binary editing tools. ARMore lifts and symbolizes the binaries to rewritable assembly code. The analyst can then locate the vulnerable code section and develop an assembly-level patch (potentially aided by visualization from binary analysis tools like Ghidra or IDA). Applying the patch to symbolized code and reassembling the binary fixes the vulnerability and mitigates exploitation. Ultimately, the relative ease of writing assembly patches instead of binary patches accelerates the mitigation of exploits. In Listing B.1, we demonstrate how a few lines of added assembly code successfully patch CVEs (CVE-2018-7584 [2] and CVE-2014-9912 [1]) in PHP. We used the vulnerable `php` binary present in the Magma fuzzing benchmark [24] to test the patches in Figure B.1.

5.7 Frequency of Data Inside Text

As explained in Section 4.2, ARMore uses signal handlers to catch and recover from reads to data inside code sections. Since this is an extensive operation, we assess the frequency of data inside text in COTS binaries. Table 2 gives a lower bound of the data-inside-text frequency for six different Ubuntu configurations. By linearly disassembling all code sections with `capstone`, we count the number of invalid instructions (excluding literal pools and `00000000` as it serves as padding). All invalid instructions are classified as data inside text. Only 0.2% of binaries contained invalid instructions in their code sections. In total, we disassembled 44 million instructions, 0.0011% of which were invalid. This number serves as a concrete lower bound for the amount of data inside text sections.

However, there might still be data inside text that disassembles correctly to a valid instruction. To adjust for this difference, we calculate the chance that random data represents a valid aarch64 instruction by disassembling all 2^{32} possible 4-bytes values, and the result is around 33.80%. We argue that a rough estimate of the frequency of data inside text is $p \cdot \frac{1}{q}$, where p is the ratio of invalid instructions, and q is the chance of random data being a valid instruction. Our final result for the lower-bound estimate of data inside text sections is thus 0.0032%.

5.8 Comparison to Other Rewriters

We compare ARMore against other static aarch64 rewriters, namely ICFG, Egalito, and Ddisasm. Section 3.2 shows a list of features used to compare against other static reweriters.

Incremental CFG Patching [32]. We ran the SQLite test suite and the SPEC CPU2017 benchmarks using ICFGP's `BlockTrampoline func-ptr` mode with empty instrumentation. Unfortunately, we could not confirm the authors' correctness claims on aarch64. The SQlite testsuite reported 12 failed tests, and out of the 23 SPEC CPU2017 benchmarks, only 10 finished successfully. We believe the source of the failures is the usage of heuristics when detecting pointer constructions (*C2*). The overhead introduced by ICFGP is on par with ARMore (<2%).

Egalito and Ddisasm. Egalito and Ddisasm are the only other static binary rewriters that claim aarch64 support. Unfortunately, we believe their support of aarch64 is experimental, or not as well-tested as their x86 support. In our tests, Egalito could not rewrite *any* binaries we tried. In their paper, they also report a low success rate (82.6%) on Debian package `autopkgtests`. Ddisasm instead proved to be successful on hand-crafted small binaries (such as `"int main() {}"`) but failed on any larger binary we tried. Previous work similarly struggled to run these two translators, i.e., ICFGP failed to run Egalito [32] and StochFuzz failed to run Ddisasm [50].

5.9 Evaluation Summary

Our evaluation demonstrates that ARMore correctly rewrites diverse binaries (486MB of code) at minimal overhead (1.0% for SPEC CPU2017). Its instrumentation passes (ASan, coverage information) are competitive with their compiler-based counterparts. ARMore is robust and scales to real-world, closed-source binaries (finding 58 bugs in our fuzzing test).

Based on our extensive evaluation covering test suites, standard benchmarks, and Debian packages, we claim ARMore's general support for aarch64. As evaluated, no other static binary translation engine provides similar performance and robustness. We commit to submitting ARMore to the artifact evaluation, fully open-sourcing it, and making Docker files and virtual machines available for testing to provide an open environment for efficient and effective aarch64 instrumentation.

6 Related Work

ARMore draws inspiration from diverse related static and dynamic binary rewriting work. In this section, we discuss the related works that are not part of our comparison in Section 3.2.

BISTRO [17] is aimed at rewriting individual components in an executable and works by *stretching* the binary to make space for the new instrumentation. It uses trampolines ("anchors") to avoid breaking indirect call targets.

McSema [18] is an excellent example of an LLVM IR lifting approach that supports x86_64 ELF and PE binaries, with C++ exceptions and aarch64 support under development. The disadvantages in lifting to an IR are that the lifting requires hefty static analysis (in fact, McSema uses IDA Pro as its backend), and some overhead might be introduced by the non-optimal lifting of the original binary. Zafl [34] is a recent static rewriter for x86_64 binaries. It focuses on fuzzing instrumentation and internally uses GCC's IR. Zafl does not employ heavy-weight dependencies to perform its static analysis.

Uroboros [43] was the first approach utilizing *reassembleable assembly* from x86 binaries. It suffers from the classic challenges of static analysis, such as relying on heuristics to differentiate between scalars and references. Ramblr [42] significantly improved Uroboros' approach with better heuristics and less overhead. RetroWrite [19] restricted the class of target binaries to position-independent ones, avoiding the problem of distinguishing between scalars and references.

Stir [44] and BinCFI [49] rely on a mapping table similar to dynamic binary translation approaches that keep a translation table between the original and the new addresses. At runtime, whenever a pointer is dereferenced, it is dynamically translated from its original address to the new address incurring overhead for the translation. This mapping mechanism has two drawbacks: performance overhead (frequently querying the mapping table for each indirect control-flow transfer, the overhead is around 30–60% [12]); and not supporting library callbacks. Pointers passed to external modules may be dereferenced without prior translation. Instrumenting libraries is the only reliable solution but incurs even higher runtime overhead.

SecondWrite [10] lifts binaries to LLVM IR before instrumenting and recompiling them. Like Multiverse, it keeps a copy of the original binary mapped at its original address to support arbitrary data pointer arithmetic. Code pointers are translated at their point of usage through a dynamic lookup, with similar drawbacks as other dynamic translation approaches.

RevARM [26] precisely detects jump tables on ARM32 binaries. However, on ARM32, jump tables are explicitly marked by special ARM32 instructions. They still rely on pattern matching to recover pointer constructions, and their support of stripped binaries relies on the IDA Pro disassembler. μSBS [38] is another binary rewriter for 32-bit ARM architectures that can precisely rewrite stripped firmware by considering only THUMB binaries.

❤ 7 ❤ Discussion

To achieve sound static rewriting, some assumptions made during the development of ARMore potentially limit its real-world applicability.

Notably, ARMore does not support self-rewriting binaries. Self-rewriting binaries are generally challenging to reliably rewrite statically due to the difficulty of correctly reproducing the self-modifying behavior. Self-rewriting binaries are mainly used by malware to obfuscate their behavior and packers to minimize size. These use cases are out of scope for ARMore and may be more easily supported by dynamic binary rewriting.

ARMore is restricted to rewritten binaries with a resulting `.text` size of up to 128MB. This restriction originates from the limited range of aarch64 branch instructions, limiting the reach from the rebound table to the corresponding instruction in the rewritten section. In our evaluation, the only binary suffering from this limitation was `chromium`, with a (non-rewritten) `.text` section of 131MB.

❤ 8 ❤ Conclusion

ARMore is a fast and precise static rewriter for Linux binaries targeting complex and challenging binaries, including non-PIC, stripped, or C++/Go targets. We address key issues in static rewriting on ARM, like dynamic pointer construction, jump table detection, and data inside code sections in a safe way. So far, these challenges either resulted in significant overhead (e.g., expensive trampolines) or imprecision (e.g., incomplete data-flow analysis). We introduce layout replication and rebound tables to fundamentally solve these challenges at complete coverage without heuristics and low overhead (1.0% for SPEC CPU2017).

ARMore is more robust and works on a wide variety of binaries (rewriting 486MB of code and over 121 million instructions). We present several instrumentation passes on top of the symbolization engine: ASan, CFI, and coverage information for fuzzing (finding 58 bugs in closed-source code).

To the best of our knowledge, ARMore is the only aarch64 rewriter that is efficient, fully precise, and supports complex real-world software. We show that the total overhead of the symbolization and the instrumentation passes are competitive with compiler-based instrumentation passes. Heuristics-free binary rewriting is not limited to the x86 architecture but is also feasible for aarch64.

❤ 9 ❤ Acknowledgments

We ❤ our shepherd and the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, and DARPA HR001119S0089-AMP-FP-034.

References

[1] Cve-2014-9912. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9912.

[2] Cve-2018-7584. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7584.

[3] Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re. Accessed: 2021-01-06.

[4] Hexrays homepage. https://hex-rays.com/. Accessed: 2021-09-10.

[5] How sqlite is tested. https://www.sqlite.org/testing.html. Accessed: 2022-04-28.

[6] More about afl. https://afl-1.readthedocs.io/en/latest/about_afl.html. Accessed: 2022-04-28.

[7] 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.

[8] Arm reference manual: A64 general instructions in alphabetical order. https://developer.arm.com/documentation/dui0802/a/a64_general_alpha, 2021.

[9] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[10] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 295–308, 2013.

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[12] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[13] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Efficient binary-level coverage analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1153–1164, 2020.

[14] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, nov 2000.

[15] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. SelectiveTaint: Efficient data flow tracking with static binary rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1665–1682, 2021.

[16] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on arm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 333–346, New York, NY, USA, 2017. Association for Computing Machinery.

[17] 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*, pages 200–218. Springer, 2013.

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

[19] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, May 2020.

[20] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[22] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092. USENIX Association, August 2020.

[23] Dongsoo Ha, Wenhui Jin, and Heekuck Oh. Repica: Rewriting position independent code of arm. *IEEE Access*, 6:50488–50509, 2018.

[24] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), nov 2020.

[25] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[26] 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*, pages 412–424, 2017.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[28] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.

[29] 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. *Acm sigplan notices*, 40(6):190–200, 2005.

[30] Catalin Marinas. Linux kernel mailing list: [patch] arm64: Introduce execute-only page access permissions. https://lore.kernel.org/lkml/1470937490-7375-1-git-send-email-catalin.marinas@arm.com/, 2016.

[31] Catalin Marinas. Linux kernel git history: arm64: Revert support for execute-only user mappings. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=24cecc37746393432d994c0dbc251fb9ac7c5d72, 2020.

[32] Xiaozhu Meng and Weijie Liu. Incremental cfg patching for binary rewriting. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 1020–1033, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Vladimir Murzint. Linux kernel mailing list: [patch v4 0/2] arm64: Support enhanced pan. https://www.spinics.net/lists/arm-kernel/msg881621.html, 2021.

[34] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[35] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[36] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, page 157–168, New York, NY, USA, 2011. Association for Computing Machinery.

[37] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[38] Majid Salehi, Danny Hughes, and Bruno Crispo. μsbs: Static binary sanitization of bare-metal embedded devices for fault observability. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 381–395, 2020.

[39] Eric Schulte, Vlad Folts, and Michael Brown. Binary lifter evaluation. *arXiv preprint arXiv2203.13231*, 2022.

[40] 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.*, pages 36–47. IEEE, 2003.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. 2012.

[42] 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.

[43] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[44] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168, 2012.

[45] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.

[46] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37, 2019.

[47] 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*, pages 133–147, 2020.

[48] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2020-11-03.

[49] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, 2013.

[50] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676. IEEE, 2021.

A SPEC CPU2017 Configuration

The benchmarks were compiled with `gcc` 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_r benchmark. The "`-fno-strict-aliasing`" flag disables GCC's aggressive aliasing compilation, and is recommended to be used by the SPEC CPU2017 manual [7] to avoid problems with the perlbench_r benchmark.

B Fuzzing Configuration

The same machine as above was used to run the Magma fuzzing benchmark. We completed fuzzing campaigns for 6 of the 9 available targets. The three excluded targets are the following: `openssl` did not compile as Magma's own instrumentation interferes with openssl's generation of armv8 neon instructions; `php` and `poppler` built successfully but Magma's instrumentation caused crashes when running them.

To fuzz the Nvidia closed-source binaries, a Macbook Air M1 running an Ubuntu 20.04 virtual machine was used.

C PC-relative Pointer Construction Detection

We carefully analyzed the aarch64 ISA [8] to determine that every possible way of reading the PC register to perform a PC-relative pointer construction is handled by ARMore. We determined that the are only two instructions that would let the user access the PC register: one is `adr`/`adrp`, and the other one is `bl`/`blr`, which stores the program counter in the link register (`x30`). Unlike the PC, the link register is a general-purpose register that can be used in standard arithmetic instructions.

It is then possible to perform a PC-relative pointer construction without `adrp` by using the link register; however, we would like to note that ARMore also supports those alternative pointer constructions thanks to call emulation. In fact, call emulation makes sure that even the return address (`x30`) points to the replicated layout, having the same effect as a symbolized `adrp`.

Nonetheless, in Linux, there are more intricate ways of performing PC-relative pointer constructions through the use of certain syscalls. It would be possible to access the value of the PC register and perform pointer calculations on it through the `sigreturn` syscall or through the `ptrace` syscall. We consider such kernel-enabled pointer constructions out of scope.

```
@ patch-CVE-2018-7584.s:10063740
@ php_stream_url_wrap_http_ex:
sub x0, x0, #1
.LC509860:
add x1, sp, #0xfb8
+ cmp x0, 1
+ b.lt .LC5098a0
.LC509864:
ldrb w0, [x1, x0]
.LC509868:

@ patch-CVE-2018-7584.s:10063760
@ php_stream_url_wrap_http_ex:
sub x0, x0, #1
.LC509884:
add x1, sp, #0xfb8
+ cmp x0, 1
+ b.lt .LC5098a0
.LC509888:
ldrb w0, [x1, x0]
.LC50988c:
```

```
@ patch-CVE-2014-9912.s:8614072
@ get_icu_disp_value_src_php:
bl .LC8c6d3c
.LC28853c:
ldr x0, [sp, #0x50]
+ cmp x0, 157
+ b.le .LC288540
+ ldr x0, [sp, #0x18]
+ movz w1, #0x2
+ str w1, [x0, #8]
+ b .LC2888f4
.LC288540:
cmp x0, #0
.LC288544:
```

Listing B.1: Patch for CVE-2018-7584 (top) and for CVE-2014-9912 (bottom)

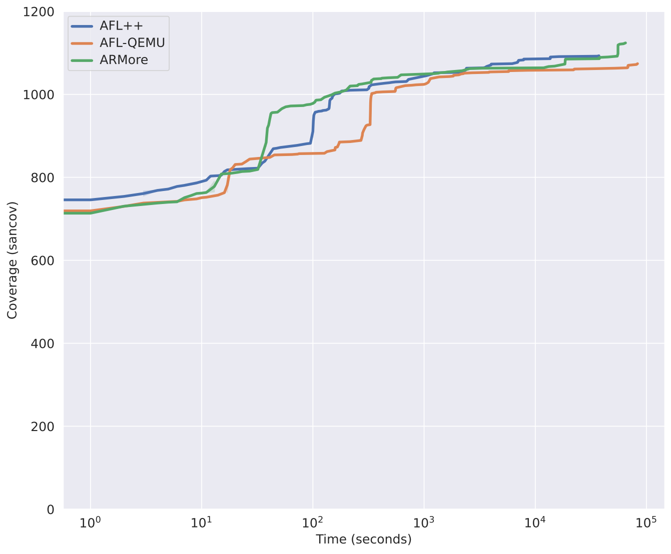
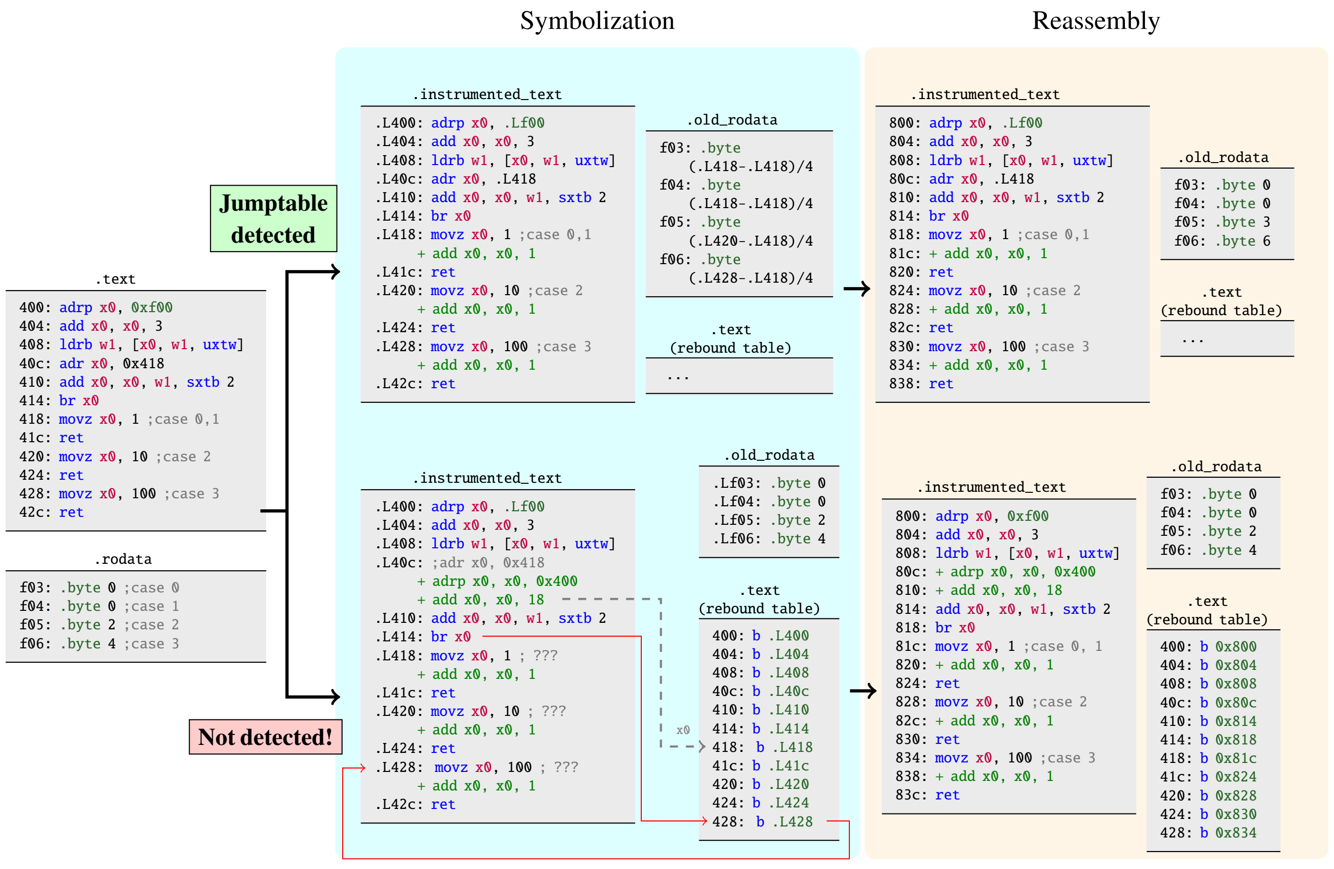


Figure B.1: Coverage growth obtained through AddressSanitizer's SanitizerCoverage over a 24 hours long campaign of Magma's libpng fuzzing benchmark.

Benchmark	ind. branches	direct calls	ind. calls	call emu. overhead	Benchmark	ind. branches	direct calls	ind. calls	call emu. overhead
perlbench	9,839,056,435	11,810,639,918	1,721,778,534	n/a	exchange2	0	842,168,009	1	0.12%
mcf	0	520,753,233	19,996,295,731	n/a	fotonik3d	0	63,245,152	1	-0.23%
x264	2,473,804	549,379,828	572,040,319	n/a	cam4	50	10,940,860,882	29,589	0.06%
gcc	323,630,018	3,370,258,184	326,599,342	n/a	deepsjeng	0	37,382,796,458	1	0.13%
imagick	38,276,695	24,167,325,317	18,757,658	n/a	cactuBSSN	62,604	1,151,872,346	166,403	1.5%
xz	67,894	446,404,721	167,440,973	n/a	bwaves	0	248,306,860	1	1.6%
namd	0	385,028,671	786,243	n/a	xalancbmk	1,704,852,967	7,359,895,165	5,473,559,842	25.14%
lbm	0	2,632,380	1	n/a	wrf	6,902	13,867,246,237	13,216	0.15%
nab	0	6,205,496,203	249	n/a	parest	277,732	6,103,591,316	2,641,122,850	3.66%
blender	9,629,315	5,852,242,234	467,613,407	6.30%	leela	0	55,572,754,413	7	48.18%
omnetpp	1,008,124,033	16,053,013,858	9,850,558,943	27.51%	povray	122,816,388	52,852,836,367	28,258,792,224	58.48%
roms	156	3,357,706,737	1	0.68%					

Table B.1: Number of indirect calls/branches executed at runtime by each of the SPEC CPU2017 binaries during their respective benchmark.



.text
```
400: adrp x0, 0xf00
404: add x0, x0, 3
408: ldrb w1, [x0, w1, uxtw]
40c: adr x0, 0x418
410: add x0, x0, w1, sxtb 2
414: br x0
418: movz x0, 1 ;case 0,1
41c: ret
420: movz x0, 10 ;case 2
424: ret
428: movz x0, 100 ;case 3
42c: ret
```

.rodata
```
f03: .byte 0 ;case 0
f04: .byte 0 ;case 1
f05: .byte 2 ;case 2
f06: .byte 4 ;case 3
```

.instrumented_text
```
.L400: adrp x0, .Lf00
.L404: add x0, x0, 3
.L408: ldrb w1, [x0, w1, uxtw]
.L40c: adr x0, .L418
.L410: add x0, x0, w1, sxtb 2
.L414: br x0
.L418: movz x0, 1 ;case 0,1
     + add x0, x0, 1
.L41c: ret
.L420: movz x0, 10 ;case 2
     + add x0, x0, 1
.L424: ret
.L428: movz x0, 100 ;case 3
     + add x0, x0, 1
.L42c: ret
```

.old_rodata
```
f03: .byte
    (.L418-.L418)/4
f04: .byte
    (.L418-.L418)/4
f05: .byte
    (.L420-.L418)/4
f06: .byte
    (.L428-.L418)/4
```

.text
(rebound table)
```
...
```

.instrumented_text
```
800: adrp x0, .Lf00
804: add x0, x0, 3
808: ldrb w1, [x0, w1, uxtw]
80c: adr x0, .L418
810: add x0, x0, w1, sxtb 2
814: br x0
818: movz x0, 1 ;case 0,1
81c: + add x0, x0, 1
820: ret
824: movz x0, 10 ;case 2
828: + add x0, x0, 1
82c: ret
830: movz x0, 100 ;case 3
834: + add x0, x0, 1
838: ret
```

.old_rodata
```
f03: .byte 0
f04: .byte 0
f05: .byte 3
f06: .byte 6
```

.text
(rebound table)
```
...
```

.instrumented_text
```
.L400: adrp x0, .Lf00
.L404: add x0, x0, 3
.L408: ldrb w1, [x0, w1, uxtw]
.L40c: ;adr x0, 0x418
     + adrp x0, x0, 0x400
     + add x0, x0, 18
.L410: add x0, x0, w1, sxtb 2
.L414: br x0
.L418: movz x0, 1 ; ???
     + add x0, x0, 1
.L41c: ret
.L420: movz x0, 10 ; ???
     + add x0, x0, 1
.L424: ret
.L428: movz x0, 100 ; ???
     + add x0, x0, 1
.L42c: ret
```

.old_rodata
```
.Lf03: .byte 0
.Lf04: .byte 0
.Lf05: .byte 2
.Lf06: .byte 4
```

.text
(rebound table)
```
400: b .L400
404: b .L404
408: b .L408
40c: b .L40c
410: b .L410
414: b .L414
418: b .L418
41c: b .L41c
420: b .L420
424: b .L424
428: b .L428
```

.instrumented_text
```
800: adrp x0, 0xf00
804: add x0, x0, 3
808: ldrb w1, [x0, w1, uxtw]
80c: + adrp x0, x0, 0x400
810: + add x0, x0, 18
814: add x0, x0, w1, sxtb 2
818: br x0
81c: movz x0, 1 ;case 0, 1
820: + add x0, x0, 1
824: ret
828: movz x0, 10 ;case 2
82c: + add x0, x0, 1
830: ret
834: movz x0, 100 ;case 3
838: + add x0, x0, 1
83c: ret
```

.old_rodata
```
f03: .byte 0
f04: .byte 0
f05: .byte 2
f06: .byte 4
```

.text
(rebound table)
```
400: b 0x800
404: b 0x804
408: b 0x808
40c: b 0x80c
410: b 0x814
414: b 0x818
418: b 0x81c
41c: b 0x824
420: b 0x828
424: b 0x830
428: b 0x834
```

Figure C.1: Jump table rewriting example. Instrumentation is inserted during the symbolization phase (the green add instructions). Above, the jump table is correctly detected: the `adr` at `.L40c` is updated to point to the base case and the offsets in .rodata are symbolized and corrected. Below, the jump table is not detected: the `adr` at `.L40c` is substituted with a pointer construction to the rebound table, and the offsets are the old ones but still valid as the indirect jump goes through the rebound table.