ARM Wrestling:
Efficient binary rewriting for aarch64

Luca Di Bartolomeo

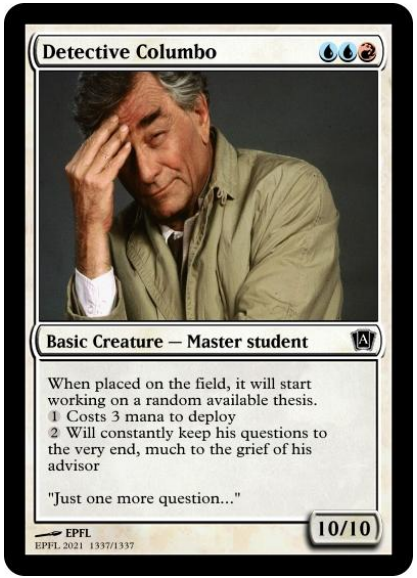
Advisor: Mathias Payer
Supervisor: Kenny Paterson

EPFL

hexhive

ETH

Credits



Luca Di Bartolomeo

Author

"Do you guys think this meme is too offensive for a master thesis?"

Prof. Mathias Payer

Advisor (EPFL)

"Don't get attached to what you write, as I'll make you rewrite it four times..."

Prof. Kenny Paterson

Ext. Supervisor (ETH)

"You, shall not, RC4!"

Binary rewriting, what

Binary rewriting, why

- Hardening
- Optimization
- Profiling
- Translation

Binary rewriting, why

- ***Hardening***
- Optimization
- Profiling
- Translation

Uses:
Stack canary protection
Address Space Layout Randomization
Address/Memory sanitization
Sandboxing

Examples:
Stackguard
RevARM
QASAN

Binary rewriting, why

- Hardening
- ***Optimization***
- Profiling
- Translation

Uses:
Cache misses optimizations
Run-time patching (no restart)

Examples:
DynInst
Frida

Binary rewriting, why

- Hardening
- Optimization
- ***Profiling***
- Translation

Uses:
Performance measurements
Memory leak detection
Fuzzing coverage information
Taint analysis

Examples:
Valgrind
AFL-QEMU

Binary rewriting, why

- Hardening
- Optimization
- Profiling
- ***Translation***

Uses:
Syscall translation for a foreign OS
Emulation of foreign architectures
Obfuscation/deobfuscation (packing)

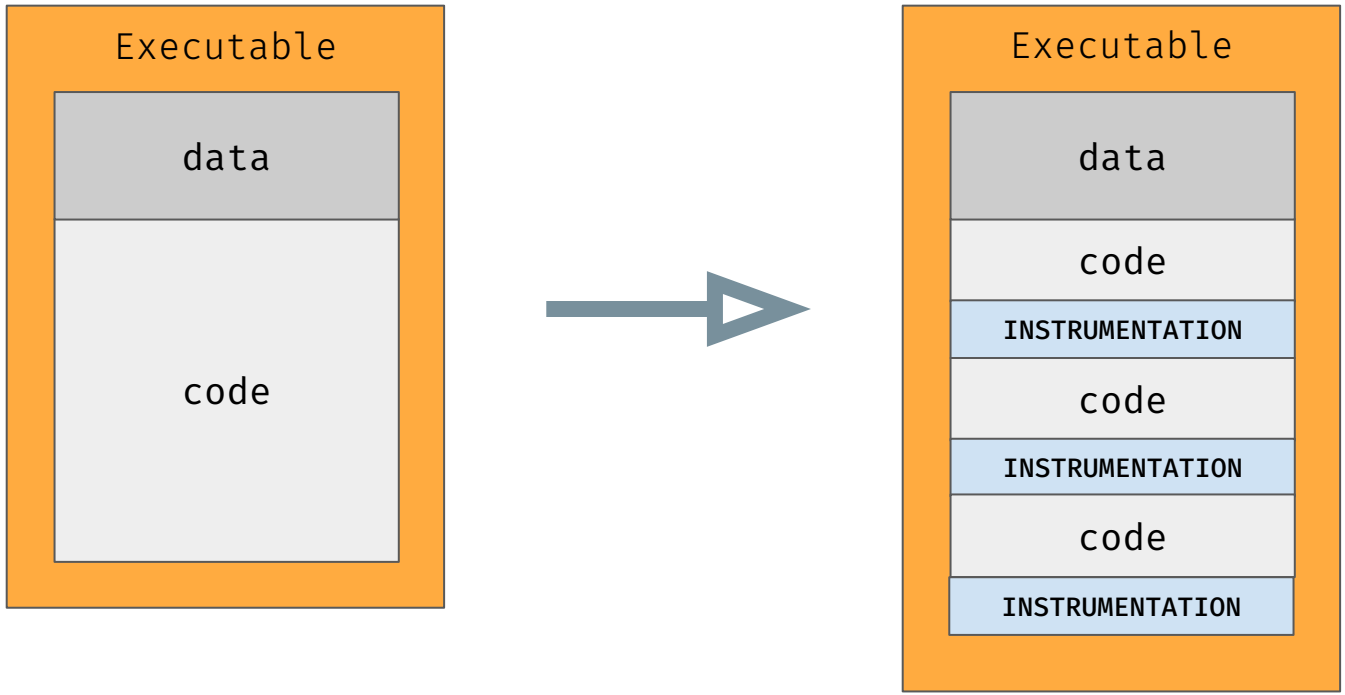
Examples:
QEMU
movfuscator

Binary rewriting, how

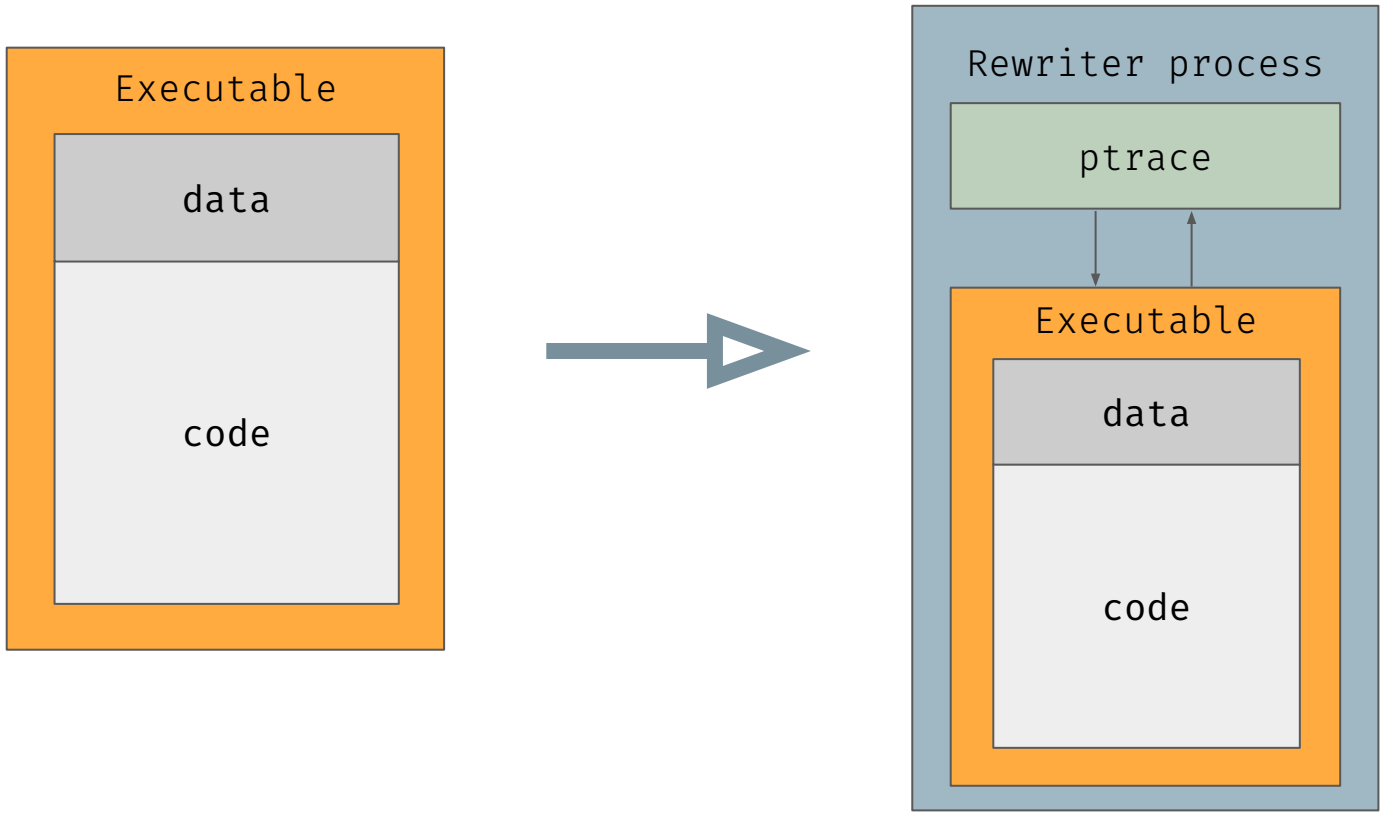
Binary rewriting techniques are split in two big categories:

- **Dynamic** rewriting
- **Static** rewriting

Static rewriting



Dynamic rewriting



Static analysis vs Dynamic analysis

Static analysis is like judging raw pasta from its look, color, weight, texture, personality, and, most importantly, bounciness.

Hard to do reliably, but possible. Requires expert eye, a fine palate, and unconditional love for pasta.

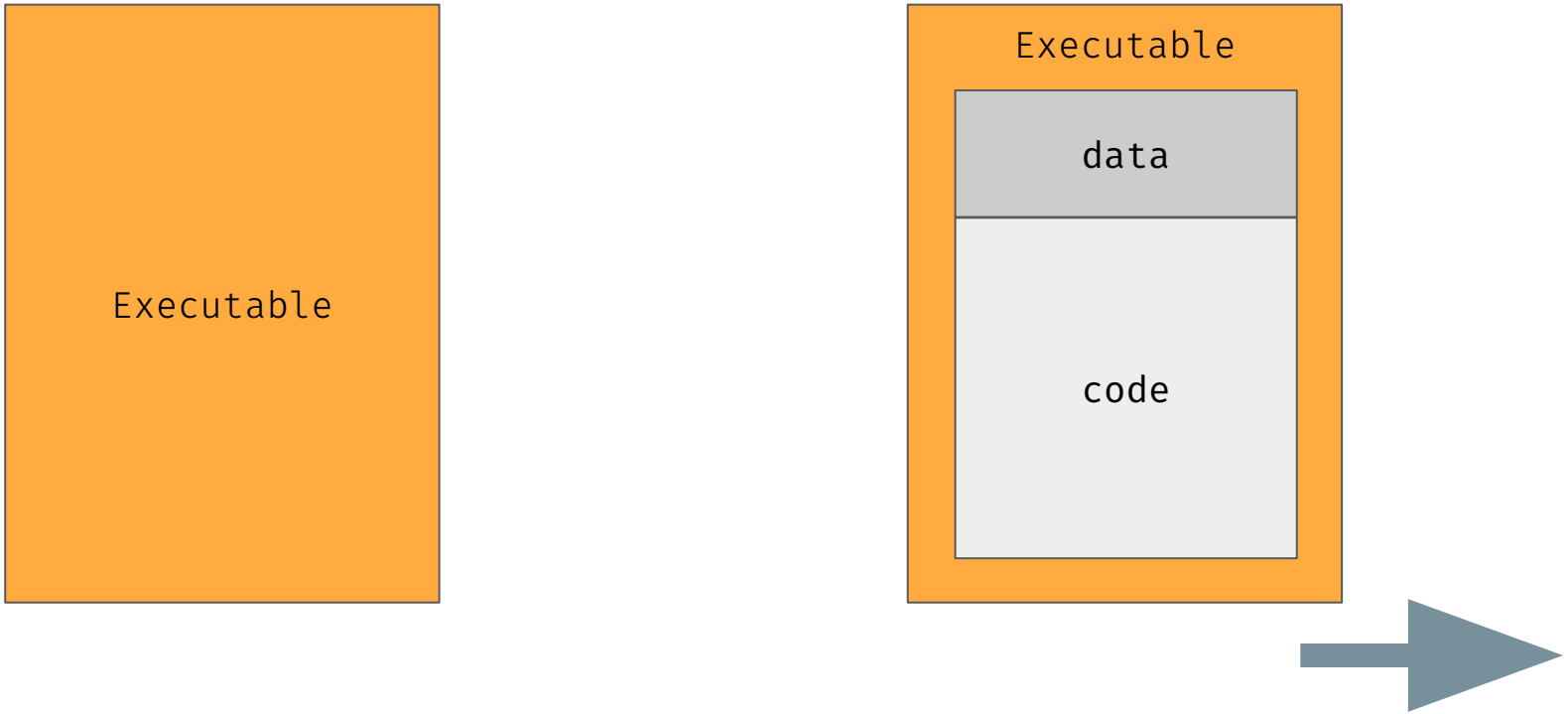
Dynamic analysis is like trying out pasta while it's boiling, to check that taste, salt and overall al-dente-ness are perfect.

Even a first timer can spot problems with dynamic pasta analysis.

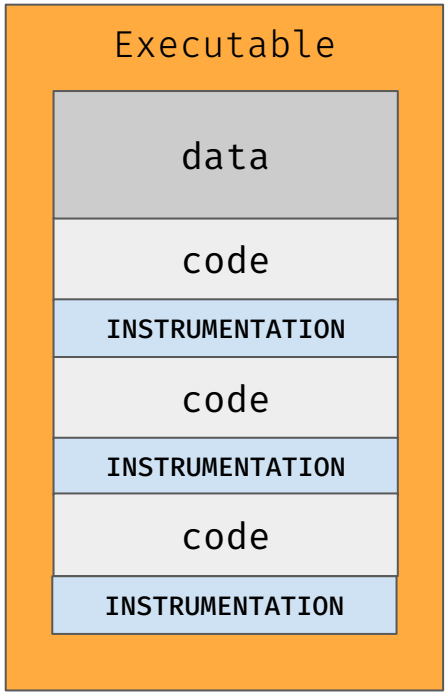
Static vs dynamic

What	Static analysis	Dynamic analysis
Code vs Data	`Problem`	`No problem`
Code coverage	`Kinda problem`	`No problem`
Self-modifying code	`Big Problem`	`No problem`
Just-In-Time code	`Big Problem`	`No problem`
Time requirements	`No problem`	`Big problem`
What are we doing	`This one :(`	`Not this one`

Static rewriting



Static rewriting



This is called ***in-place*** instrumentation.

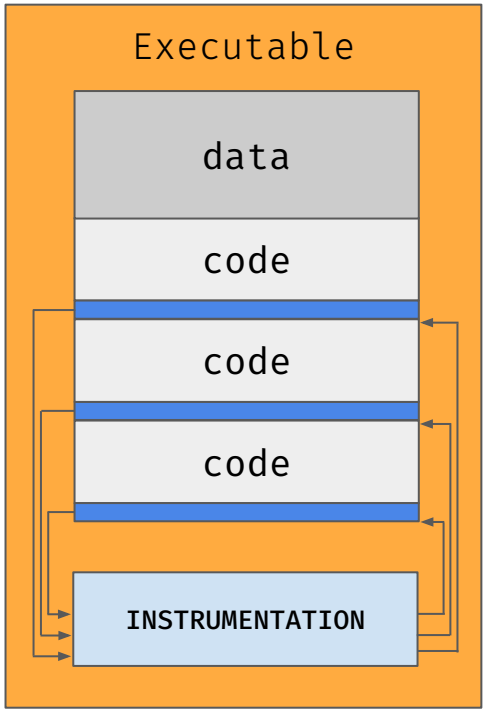
Advantages:

- Lowest possible overhead

Disadvantages:

- Platform-dependent
- Unflexible
- All control flow AND references broken
  - Need to rely on complex static analysis and instruction patching to readjust the layout

Static rewriting



= unconditional jump

This is ***trampoline*** based instrumentation.

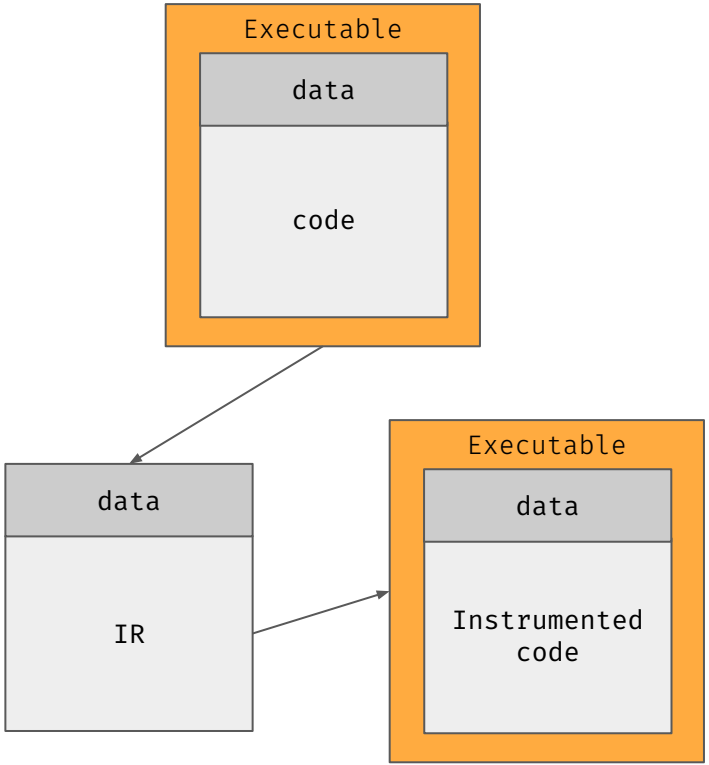
Advantages:

- Easy and fast to implement
- Does not break references/control flow

Disadvantages:

- Slow, two jumps inserted at each instrumentation point

Static rewriting



This process is called ***IR lifting*** (Intermediate Representation)

Advantages:

- Very flexible, can support many architectures at the same time

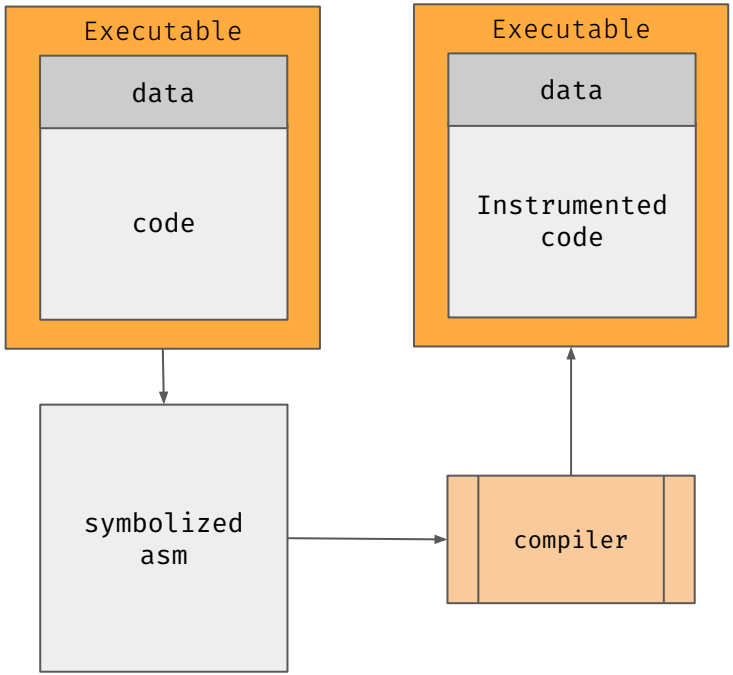
Disadvantages:

- Hard to implement
- Translation can be inaccurate and lead to slowdown

Symbolization

sym′bol·i·za′tion (sĭm-bə-lĭ-zā′shən) *n.*
“The process of producing reassemblable assembly from a binary.

In other words,
Symbolization = disassembly + substituting references with assembly labels
The result can be directly fed to an assembler to produce a binary with the same functionality as the original one. That’s why it’s called *reassemblable* assembly.

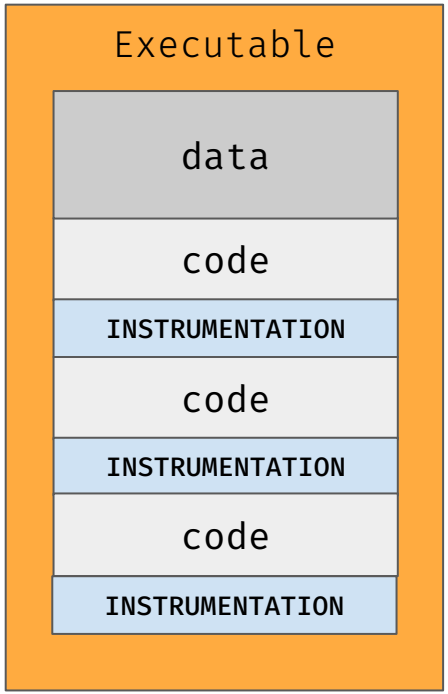


Symbolization example

```
Myfunc:
    movz x0, 3
    bl 0×400
    cbz x0, 4
    mov x0, 1
    ret
    mov x0, 0
    ret
```

```
Symbolized_Myfunc:
    movz x0, 3
    bl .LC400
    cbz x0, .LC802
    mov x0, 1
    ret
.LC802:
    mov x0, 0
    ret
```

Static rewriting



This is called ***in-place*** instrumentation.

Advantages:

- Lowest possible overhead

Disadvantages:

- Platform-dependent
- *Unflexible*
- *All control flow AND references broken*
  - *Need to rely on complex static analysis and instruction patching to readjust the layout*

Distinguishing between code and data

```
Myfunc:
    movz x0, 3
    bl 0×400
    cbz x0, 4
    mov x0, 1
    ret
    mov x0, 0
    ret
```

```
Symbolized_Myfunc:
    movz x0, 3
    bl .LC400
    cbz x0, .LC802
    mov x0, 1
    ret
.LC802:
    mov x0, 0
    ret
```

A not-so-trivial example

```
Myfunc:
    cbz x0, 4
    mov x0, 1
    movz x0, 0×400
    br x0
    ret
```

Distinguishing between code and data

A very hard problem, like many others in life

Retrowrite

RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization

Sushant Dinesh
Purdue University

Nathan Burow
Purdue University

Dongyan Xu
Purdue University

Mathias Payer
EPFL

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

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

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

In this paper, we show that static binary rewriting, leveraging reassembleable assembly, can produce sound and efficient code for an important class of binaries: *64-bit position-*

Retrowrite

Solves the data/reference distinction by focunsing only on PIE (position-independent executables).

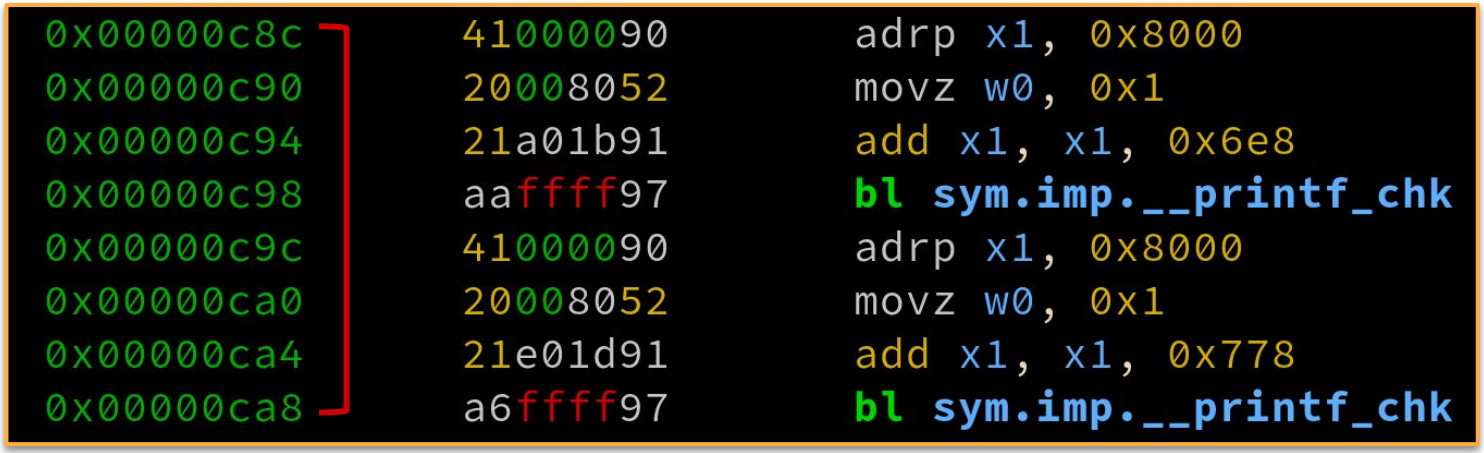
Originally developed for x86_64, we extend it to aarch64

Not a simple porting job! ARM has many specific quirks that introduced many challenges

Fixed Size Instruction Set

```
0x00000c8c      41000090        adrp x1, 0x8000
0x00000c90      20008052        movz w0, 0x1
0x00000c94      21a01b91        add x1, x1, 0x6e8
0x00000c98      aaffff97        bl sym.imp.__printf_chk
0x00000c9c      41000090        adrp x1, 0x8000
0x00000ca0      20008052        movz w0, 0x1
0x00000ca4      21e01d91        add x1, x1, 0x778
0x00000ca8      a6ffff97        bl sym.imp.__printf_chk
```

Fixed Size Instruction Set



ARM-specific issues

- Detecting and fixing pointer constructions
- Detecting and symbolizing jump tables
- Symbolizing large jump tables

ARM-specific issues

- *Detecting and fixing pointer constructions*
- Detecting and symbolizing jump tables
- Symbolizing large jump tables

Global variables

Each instruction on ARM is 4 bytes large. This is why it is called a fixed-size instruction set.

Unfortunately, on 64-bit processors, addresses are 8 bytes long. So, you cannot include an address in a single instruction.

Global variables

Each instruction on ARM is 4 bytes large. This is why it is called a fixed-size instruction set.

Unfortunately, on 64-bit processors, addresses are 8 bytes long. So, you cannot include an address in a single instruction.

```
movz eax, <pointer>
```

Global variables

Each instruction on ARM is 4 bytes large. This is why it is called a fixed-size instruction set.

Unfortunately, on 64-bit processors, addresses are 8 bytes long. So, you cannot include an address in a single instruction.

Global variables

Each instruction on ARM is 4 bytes large. This is why it is called a fixed-size instruction set.

Unfortunately, on 64-bit processors, addresses are 8 bytes long. So, you cannot include an address in a single instruction.

~~`movz eax, <pointer>`~~

There are two alternative ways to use addresses on ARM:

- Using *literal pools*
- Using multiple instructions in a process called *pointer construction* or *pointer building*

Using addresses on aarch64

- Literal pools
- Pointer constructions

Using addresses on aarch64

- ***Literal pools***
- Pointer constructions

Literal pools are special memory regions in which the compiler stores **absolute addresses** that can be then stored into a register through a standard memory load.

```
ldr x0, =<pointer>
```

With the above instruction, we will store the address `<pointer>` in register `x0`. The address will be stored as a constant (a *literal*) in a manually specified memory region.

Using addresses on aarch64

- Literal pools
- ***Pointer constructions***

This means arithmetically building pointers through a set of computations.

```
adrp x0, <pointer>
add x0, x0, :lo12:<pointer>
```

The first adrp will load the base page in `x0`, and then the internal page offset (the lowest significant 12 bits of `<pointer>`) is added to `x0`

Using addresses on aarch64

- Literal pools
- Pointer constructions

The problem:

> Pointer constructions require 2 instructions (or more). Literal pools require a single one, but it's a memory load, so it is slower than pointer constructions in general.

> Compilers almost always use pointer constructions.

> Pointer constructions need to be detected, as pointers need to be symbolized (substituted with an assembly label)

> Pointers are very common, and compilers absolutely love optimizations. They will mix and match pointer constructions until they are *very hard to detect*

Literal pools vs pointer constructions

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%**

We implemented both and ran a benchmark to compare times. We decided to use pointer constructions for our final implementation.

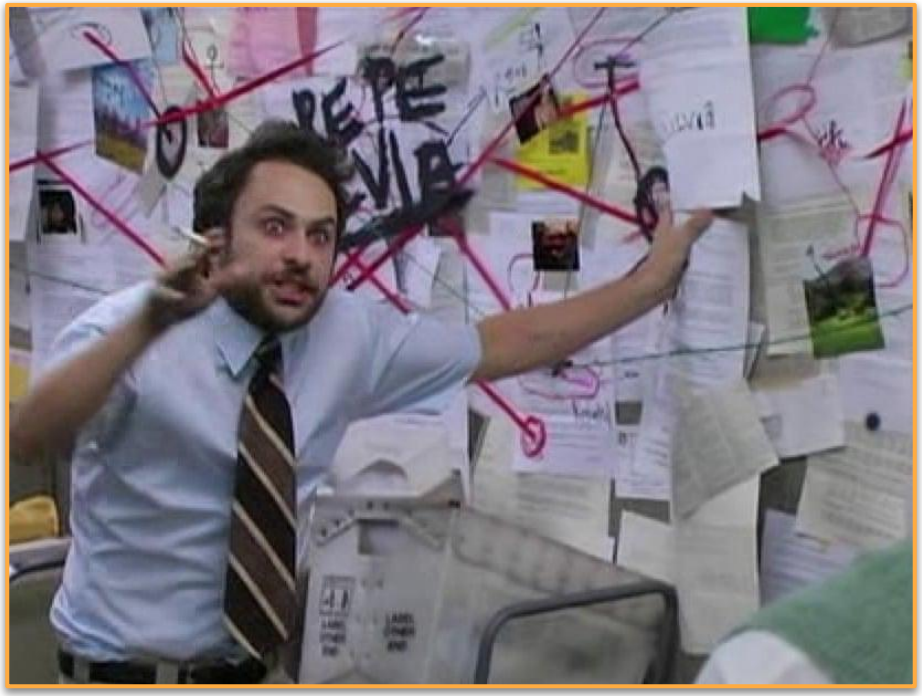
Literal pools vs pointer constructions

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%**

We implemented both and ran a benchmark to compare times. We decided to use pointer constructions for our final implementation.
> The big problem left was the detection of pointer constructions

First approach:

First approach: Pattern Matching



Static analysis becomes hard very fast

Static analysis

Static analysis is like pasta: you are never done with it.

First approach: Pattern Matching

We implement detection after spotting repeating simple patterns:

```
adrp x0, 0×8000
add x0, x0, 0×128
. . .
. . .
adrp x0, 0×8000
sub x2, x2, x3
add x0, x0, 0×128
```

First approach: Pattern Matching

We implement detection after spotting repeating simple patterns:

```
adrp x0, 0×8000
add x0, x0, 0×128
. . .
. . .
adrp x0, 0×8000
sub x2, x2, x3
add x0, x0, 0×128
```

But way too many edgecases popped up, mostly out of compiler optimizations

First approach: Pattern Matching

We implement detection after spotting repeating simple patterns:

```
adrp x0, 0×8000
str x0, [sp, -0×8]
div x1, x2, x4
br  x3
ldr x0, [sp, -0×8]
add x0, 0×128
```

```
adrp x0, 0×8000
add x0, x0, 0×128
. . .
. . .
adrp x0, 0×8000
sub x2, x2, x3
add x0, x0, 0×128
```

```
adrp x0, 0×8000
add x0, x0, x1
and x0, x0, x2
add x0, 0×128
```

But way too many edgecases popped up, mostly out of compiler optimizations

First approach: Pattern Matching

We implement detection after spotting repeating simple patterns:

```
adrp x0, 0×8000
str x0, [sp, -0×8]
div x1, x2, x4
br  x3
ldr x0, [sp, -0×8]
add x0, 0×128
```

```
adrp x0, 0×8000
add x0, x0, 0×128
. . .
. . .
adrp x0, 0×8000
sub x2, x2, x3
add x0, x0, 0×128
```

```
adrp x0, 0×8000
add x0, x0, x1
and x0, x0, x2
add x0, 0×128
```

But way too many edgecases popped up, mostly out of compiler optimizations

> Static analysis becomes hard very fast

Second approach:

Second approach: section pruning

Cutting corners - apparently in academia it's a very praised practice and it's called "*research*"

Second approach: section pruning

We try to fix the `adrp` only, ignoring all the other instructions used to build the pointer

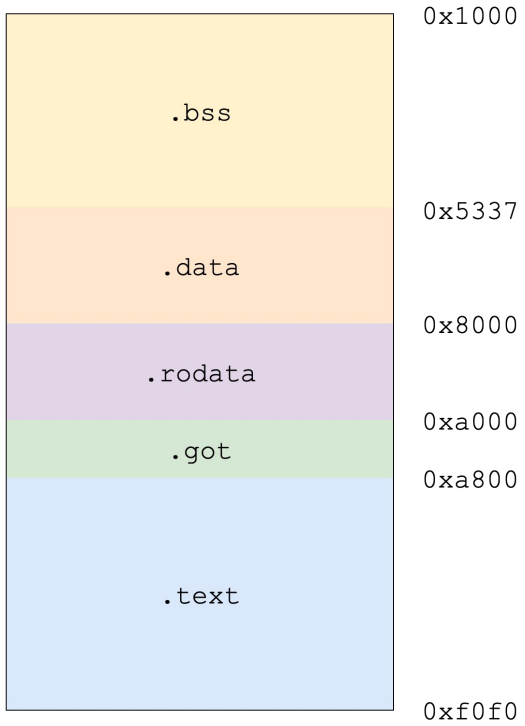
```
adrp x0, 0×3000
add x0, x0, 0×128
```

Second approach: section pruning

We try to fix the `adrp` only, ignoring all the other instructions used to build the pointer

```
adrp x0, 0×3000
add x0, x0, 0×128
```

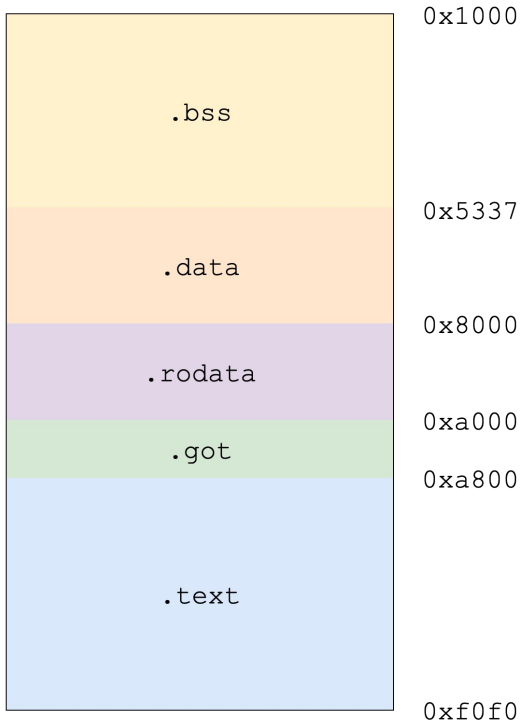
Retrowrite does not change sections other than the `.text`, as instrumentation is relevant only to the code of the binary, not the data.



Second approach: section pruning

We prune until we have only one possible section left

```
adrp x0, 0×3000
add x0, x0, 0×128
```

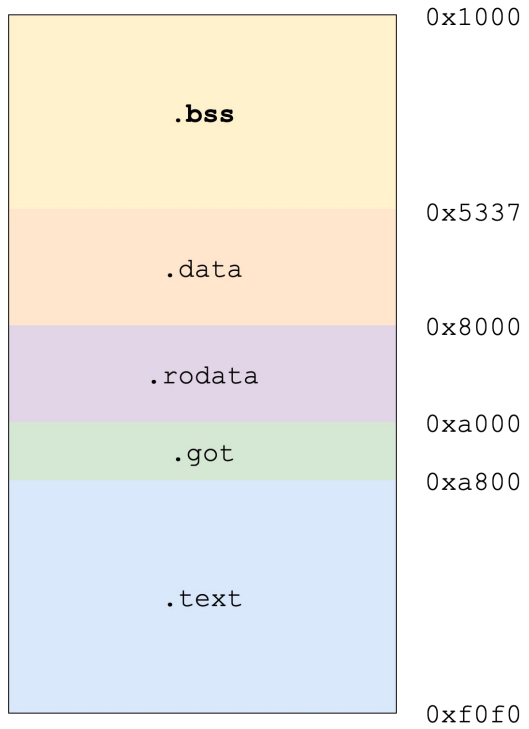


Second approach: section pruning

We prune until we have only one possible section left

```
adrp x0, 0×3000
add x0, x0, 0×128
```

We identify the regions which lie in the +- 1 KB range from the adrp



Second approach: section pruning

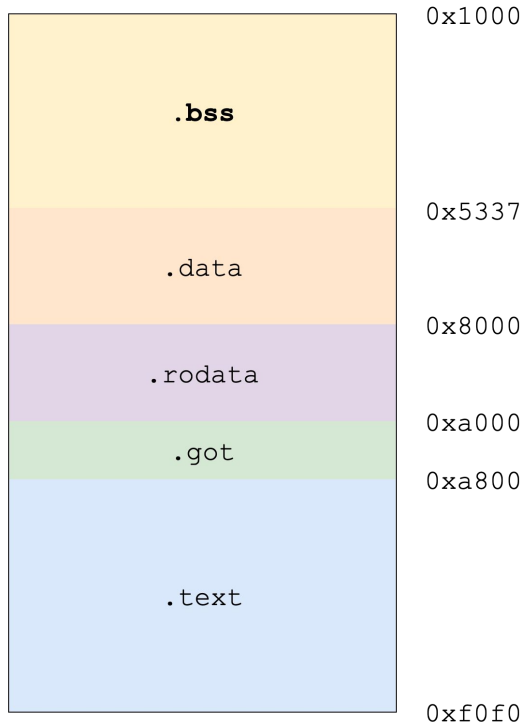
We prune until we have only one possible section left

```
adrp x0, 0×3000
add x0, x0, 0×128
```

We identify the regions which lie in the +- 1 KB range from the adrp

If there is only one section, then the symbolization is easy:

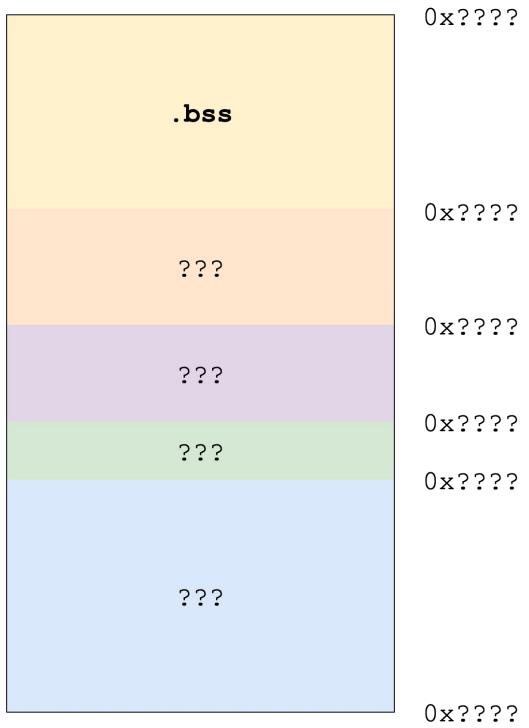
```
adrp x0, (.bss + 0×2000)
add x0, x0, 0×128
```



Second approach: section pruning

We prune until we have only one possible section left

```
adrp x0, (.bss + 0×2000)
add x0, x0, 0×128
```

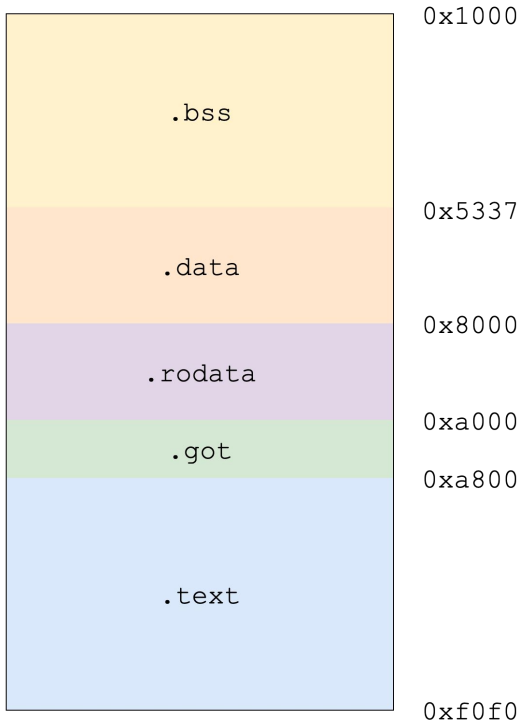


> sections other than .text are not modified

> offsets inside a single sections will stay the same no matter the memory layout

Second approach: section pruning

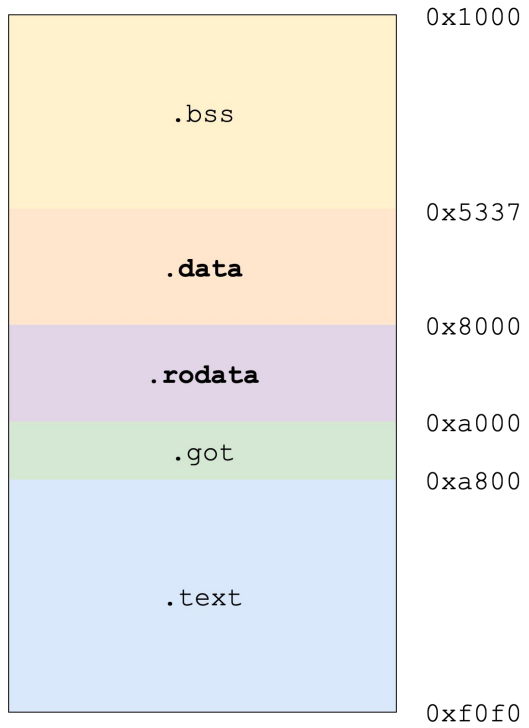
Sometimes this is not possible, as multiple sections overlap the 1 KB range.



Second approach: section pruning

Sometimes this is not possible, as multiple sections overlap the 1 KB range.

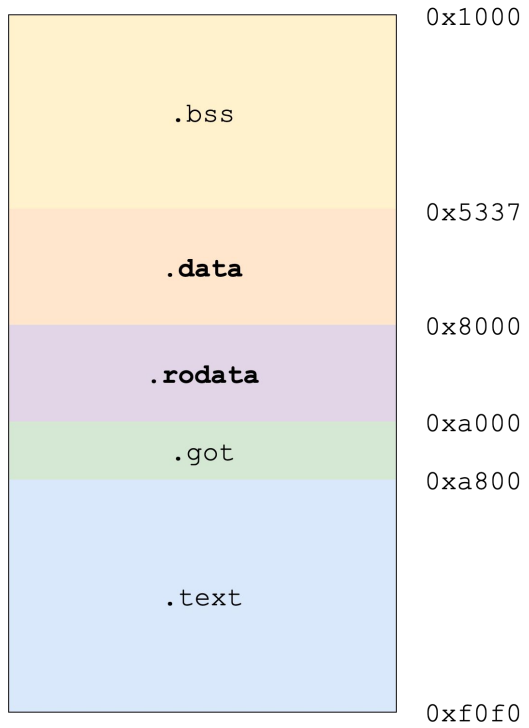
```
adrp x0, 0×7ff0
add x0, x0, 0×128
```



Second approach: section pruning

Sometimes this is not possible, as multiple sections overlap the 1 KB range.

```
adrp x0, 0×7ff0
add x0, x0, 0×128
```



> In this case, we fall back to old pattern matching

Symbolizing pointer constructions

Using a combination of the first approach (pattern matching) and the second (section pruning), we can rewrite binaries as large as the gcc benchmark of SPEC CPU2017 (10 MB binary), correctly rewriting all pointer constructions.

We also rewrote the entire coreutils software corpus with retrowrite and verified that the resulting binaries still pass all tests of the coreutils testing framework

ARM-specific issues

- Detecting and fixing pointer constructions
- *Detecting and symbolizing jump tables*
- Symbolizing large jump tables

Jump table detection

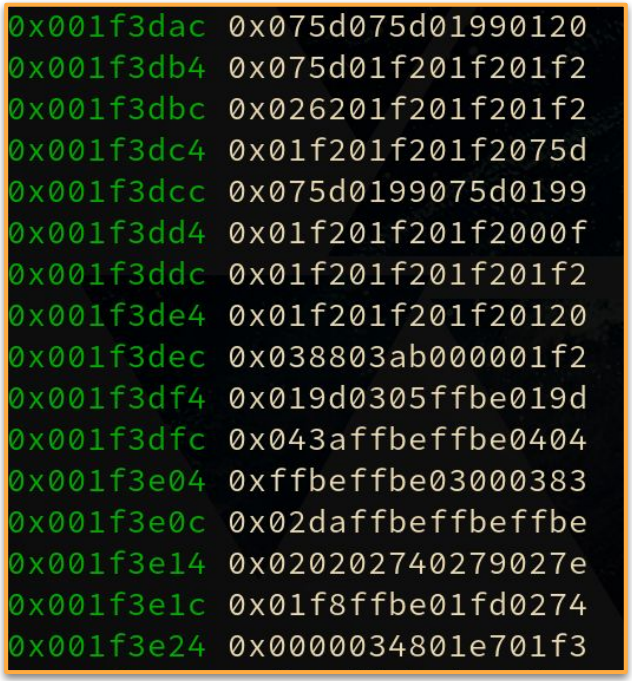
Contrary to x86, jump tables in ARM can be hard to detect

Jump tables on ARM are stored *compressed*, since instead of storing absolute addresses, they store *offsets* from the *base case*.

This makes a jump table indistinguishable from random memory, as sometime a every case is only 1 or 2 bytes large.

Jump table detection

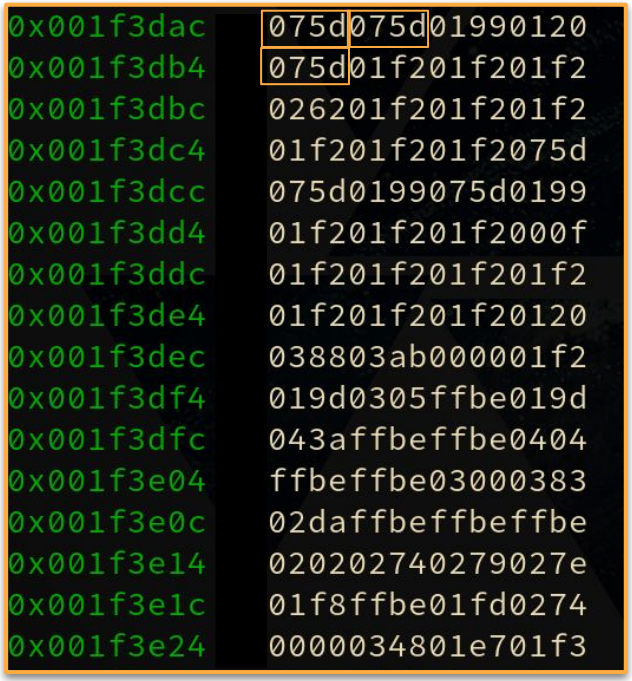
x86_64



ARM

Jump table detection

x86_64



ARM

An example jump table

```
adrp x0, <table_page_addr>      // pointer construction
add  x0, x0, <table_page_off>   // to jump table
ldrb w1, [x0, w1, uxtw]         // load the offset in reg w1
adr  x0, <base_case_addr>       // pointer to the first case of the switch
add  x0, x0, w1, sxtb 2         // add the (offset << 2) to the base case
br   x0, <base_case_addr>       // jump there :)
. . .
```

An example jump table

```
adrp x0, <table_page_addr>      // pointer construction
add  x0, x0, <table_page_off>   // to jump table
ldrb w1, [x0, w1, uxtw]         // load the offset in reg w1
adr  x0, <base_case_addr>       // pointer to the first case of the switch
add  x0, x0, w1, sxtb 2         // add the (offset << 2) to the base case
br   x0, <base_case_addr>       // jump there :)
. . .
```

> How to detect that it’s a jump table?

We thought about pattern matching again, but we chose a more robust solution this time

Jump table detection with backwards slicing

We implemented a bare-bones symbolic emulator that backwards slices from each indirect jump to check for offset-based branches:

Jump table detection with backwards slicing

We implemented a bare-bones symbolic emulator that backwards slices from each indirect jump to check for offset-based branches:

```
cmp  w1, 16
b.hi <default_case>
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 2
br   x0, <base_case_addr>
. . .
```

Jump table detection with backwards slicing

We implemented a bare-bones symbolic emulator that backwards slices from each indirect jump to check for offset-based branches:

```
cmp  w1, 16
b.hi <default_case>
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 2    // x0 = x0 + (w1 << 2)
br   x0, <base_case_addr> // x0
. . .
```

Jump table detection with backwards slicing

We implemented a bare-bones symbolic emulator that backwards slices from each indirect jump to check for offset-based branches:

```
cmp  w1, 16
b.hi <default_case>
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]   // x0 = 0×418 + (*(x0 + w1) << 2)
adr  x0, 0×418            // x0 = 0×418 + (w1 << 2)
add  x0, x0, w1, sxtb 2   // x0 = x0 + (w1 << 2)
br   x0, <base_case_addr> // x0
. . .
```

Jump table detection with backwards slicing

We implemented a bare-bones symbolic emulator that backwards slices from each indirect jump to check for offset-based branches:

```
cmp  w1, 16
b.hi <default_case>
adrp x0, 0x8000           // x0 = 0x418 + (*(0x8128 + w1) << 2)
add  x0, x0, 0x128        // x0 = 0x418 + (*(x0 + 0x128 + w1) << 2)
ldrb w1, [x0, w1, uxtw]   // x0 = 0x418 + (*(x0 + w1) << 2)
adr  x0, 0x418            // x0 = 0x418 + (w1 << 2)
add  x0, x0, w1, sxtb 2   // x0 = x0 + (w1 << 2)
br   x0                   // x0
. . .
```

Jump table detection with backwards slicing

We implemented a bare-bones symbolic emulator that backwards slices from each indirect jump to check for offset-based branches:

```
cmp  w1, 16
b.hi <default_case>        // Exit if w1 higher than 16
adrp x0, 0×8000            // x0 = 0×418 + (*(0×8128 + w1) << 2)
add  x0, x0, 0×128         // x0 = 0×418 + (*(x0 + 0×128 + w1) << 2)
ldrb w1, [x0, w1, uxtw]    // x0 = 0×418 + (*(x0 + w1) << 2)
adr  x0, 0×418             // x0 = 0×418 + (w1 << 2)
add  x0, x0, w1, sxtb 2    // x0 = x0 + (w1 << 2)
br   x0                    // x0
. . .
```

Jump table detection with backwards slicing

We implemented a bare-bones symbolic emulator that backwards slices from each indirect jump to check for offset-based branches:

```
cmp  w1, 16
b.hi <default_case>          // Exit if w1 higher than 16
adrp x0, 0×8000             // x0 = 0×418 + (*(0×8128 + w1) << 2)
add  x0, x0, 0×128          // x0 = 0×418 + (*(x0 + 0×128 + w1) << 2)
ldrb w1, [x0, w1, uxtw]     // x0 = 0×418 + (*(x0 + w1) << 2)
adr  x0, 0×418              // x0 = 0×418 + (w1 << 2)
add  x0, x0, w1, sxtb 2     // x0 = x0 + (w1 << 2)
br   x0                     // x0
. . .

                            Results:
                            Base case: 0×418          Case register: w1
                            Jump table addr: 0×8128   Number of cases: 16
```

Jump table detection with backwards slicing

The small symbolic emulator was an interesting project on its own, ended up with the following features:

- Over 30 ARM instructions supported
- Support for interleaved instructions
- Support for nested jump tables
- Support for control-flow-interleaved jump tables

Dongsoo Ha, Wenhui Jin, and Heekuck Oh. “REPICA: Rewriting Position Independent Code of ARM”

ARM-specific issues

- Detecting and fixing pointer constructions
- Detecting and symbolizing jump tables
- *Symbolizing large jump tables*

Approaches we tried

1. Make a brand new jumptables with more space available for each case
    a. Feasible, but wasted space and required changes in the jump table code
2. Change the memory layout to make more space for the existing jump table
    a. Hard to implement and would have broken the “not instrumenting data”
3. Tradeoff some of the jump table precision by “enlarging” it
    a. Easy to implement and required only minor changes in the code

Symbolizing jump tables

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1
br   x0, <base_case_addr>
. . .
```

```
.LCd568:
    .byte 8
.LCd569:
    .byte 12
.LCd56a:
    .byte 12
.LCd56b:
    .byte 20
.LCd56c:
    .byte 12
.LCd56d:
    .byte 20
.LCd56e:
    .byte 40
```

Symbolizing jump tables

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 2
br   x0, <base_case_addr>
. . .
```

```
.LCd568:
    .byte 2
.LCd569:
    .byte 3
.LCd56a:
    .byte 3
.LCd56b:
    .byte 5
.LCd56c:
    .byte 4
.LCd56d:
    .byte 5
.LCd56e:
    .byte 10
```

Symbolizing jump tables

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 2
br   x0, <base_case_addr>
. . .
```

```
.LCd568:
    .byte (.LCa3a0-.LCa350)/4
.LCd569:
    .byte (.LCa3a0-.LCa350)/4
.LCd56a:
    .byte (.LCa3bc-.LCa350)/4
.LCd56b:
    .byte (.LCa3bc-.LCa350)/4
.LCd56c:
    .byte (.LCa384-.LCa350)/4
.LCd56d:
    .byte (.LCa384-.LCa350)/4
.LCd56e:
    .byte (.LCa350-.LCa350)/4
```

Symbolizing jump tables

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 4
br   x0, <base_case_addr>
. . .
```

```
.LCd568:
    .byte (.LCa3a0-.LCa350)/16
.LCd569:
    .byte (.LCa3a0-.LCa350)/16
.LCd56a:
    .byte (.LCa3bc-.LCa350)/16
.LCd56b:
    .byte (.LCa3bc-.LCa350)/16
.LCd56c:
    .byte (.LCa384-.LCa350)/16
.LCd56d:
    .byte (.LCa384-.LCa350)/16
.LCd56e:
    .byte (.LCa350-.LCa350)/16
```

Jump table enlargement

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1
br   x0, <base_case_addr>
. . .

0×8000: 0
0×8001: 8
0×8002: 16
0×8003: 24
0×8004: 32
0×8005: 40
```

	br x1
0	**movz** x0, 0
4	**ret**
8	**movz** x0, 1
12	**ret**
16	**movz** x0, 2
20	**ret**
24	**movz** x0, 3
28	**ret**

Jump table enlargement

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 2
br   x0, <base_case_addr>
. . .

0×8000: 0
0×8001: 2
0×8002: 4
0×8003: 6
0×8004: 8
0×8005: 10
```

	br x1
0	**movz** x0, 0
4	**ret**
8	**movz** x0, 1
12	**ret**
16	**movz** x0, 2
20	**ret**
24	**movz** x0, 3
28	**ret**

Jump table enlargement

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 3
br   x0, <base_case_addr>
. . .

0×8000: 0
0×8001: 1
0×8002: 2
0×8003: 3
0×8004: 4
0×8005: 5
```

	br x1
0	**movz** x0, 0
4	**ret**
8	**movz** x0, 1
12	**ret**
16	**movz** x0, 2
20	**ret**
24	**movz** x0, 3
28	**ret**

Jump table enlargement

```
adrp x0, 0×8000
add  x0, x0, 0×128
ldrb w1, [x0, w1, uxtw]
adr  x0, 0×418
add  x0, x0, w1, sxtb 4
br   x0, <base_case_addr>
. . .

0×8000: 0
0×8001: 1
0×8002: 2
0×8003: 3
0×8004: 4
0×8005: 5
```

	br x1
0	**movz** x0, 0
4	**ret**
8	**NOP**
12	**NOP**
16	**movz** x0, 1
20	**ret**
24	**NOP**
28	**NOP**

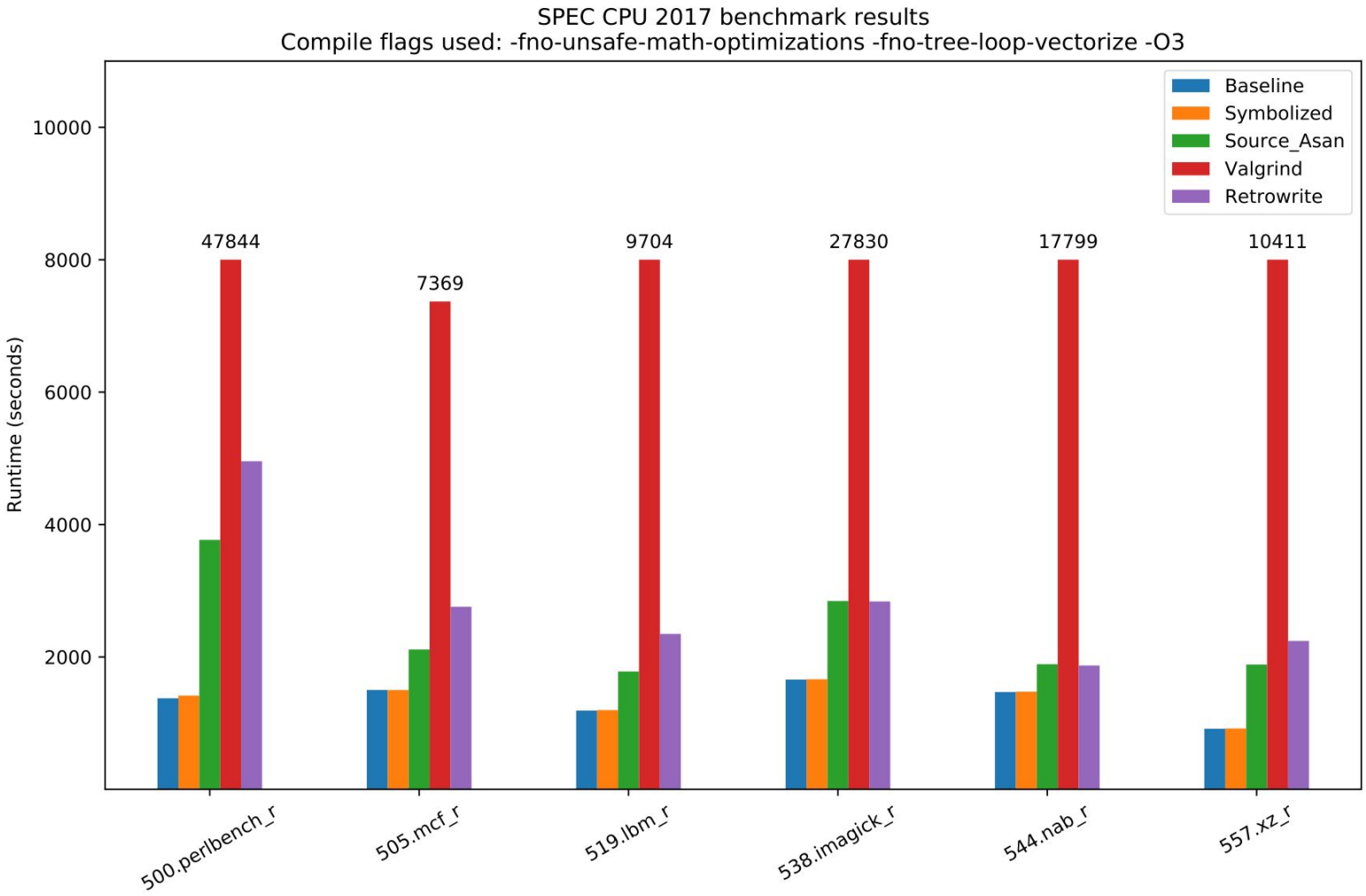
Results

SPEC CPU2017 benchmarks

Memory sanitization instrumentation (quite heavy)

30% slower than source-based memory sanitization

Orders of magnitude faster than dynamic rewriting


SPEC CPU 2017 benchmark results
Compile flags used: -fno-unsafe-math-optimizations -fno-tree-loop-vectorize -O3
Runtime (seconds)
10000
8000
6000
4000
2000
Baseline
Symbolized
Source_Asan
Valgrind
Retrowrite
47844
7369
9704
27830
17799
10411
500.perlbench_r
505.mcf_r
519.lbm_r
538.imagick_r
544.nab_r
557.xz_r

Key improvements over previous works

- First *zero-overhead* aarch64 rewriter that supports enlarged jump-tables
- First attempt at symbolizing ARM
- *Fast* static memory sanitization instrumentation pass
- Focus on modularity and ease of extensibility.

  The codebase is around ~2k LOC of python, and we have already numerous issues on github by people extending it with their own instrumentation

Limitations

- Does not work on non-C binaries (no C++, java, handwritten assembly)
- Works only on binaries produced by well-behaved compilers (gcc, clang)
  - No self-modifying code
  - No inline assembly!
  - No obfuscated code/packed payloads/malware hiding techniques
- Other common limitations of static analysis:
  - Disassembly is generally undecidable
  - If there are no symbols in a binary, forced to rely on imperfect heuristics for things like function detection

Future Work

- **More programming languages** support, like C++ exceptions
- Support for instrumenting **kernel modules**: there are subtle differences for symbolizing kernel modules, but this could open up the way to efficiently fuzz Android vendor modules and other embedded firmwares
- Support for **more operatings systems**: right now there is only support for Linux ELF executables, but targeting Windows and OSX would be interesting
- Thanks to the modular design of RetroWrite, **interesting additional instrumentation passes** can be implemented such as shadow stack, control-flow authentication with PAC codes, coverage guidance for fuzzing, and more.

Special thanks to...

In no particular order, the people in this list had a major impact on my journey
Thanks :D

giuliagl
chiccosala
linus14
picorana
neon
lamberto_lamberti
frattack
neopt
lightblue
ak

gannimo
filippog
tulymyhero
gallileo
matteo_chen
dariusk
andreafioraldi
null
kennyog
robin_jadoul

frigol
francesco_iadevaia
jean_michel
flagbot
p0lyglots

Thanks for listening!

Appendix

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.

BASAN

Binary ASAN

ASAN = Address SANitizer

Developed by Google, as a compiler pass for LLVM to provide memory safety in C based languages

Works by hooking malloc, free and friends and inserting a check before every memory load/store done by the binary

BASAN = Binary Address SANitizer

It’s the name of our instrumentation pass. The functionality is very similar to the original ASAN, with differences from the fact that BASAN is applied to an already compiled binary, without using its source code

Register Savings

We can use static analysis to determine which registers can be safely used inside a function without overwriting important values to avoid wasting time pushing them on the stack and popping them back later

```
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).

Register Savings

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.

Comparison to trampolines

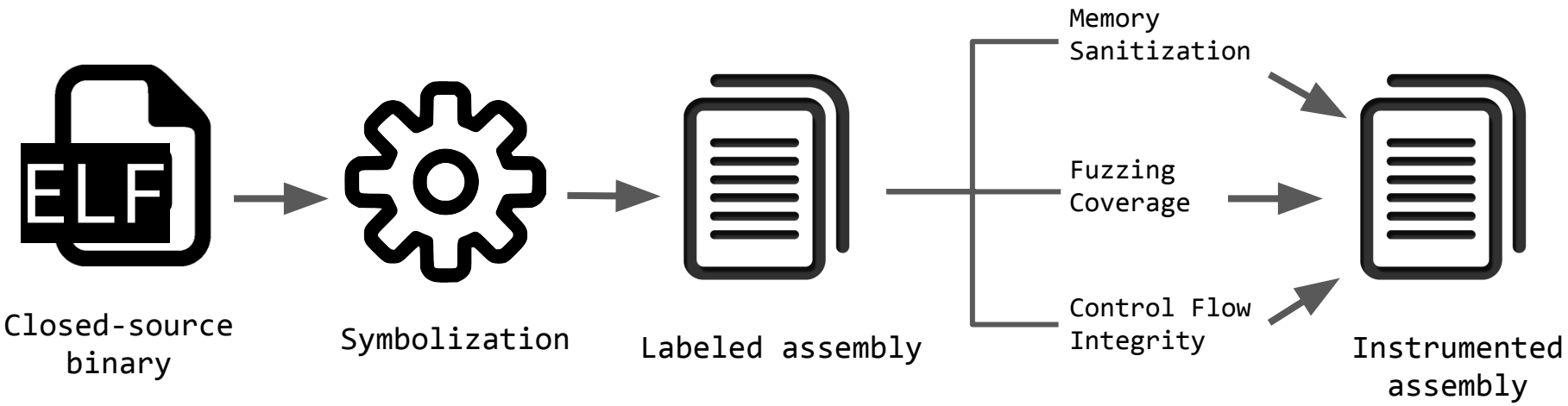
Name	Binary ASAN	Trampoline ASAN
perlbench_r	260.10%	636.41%
imagick_r	71.17%	188.96%
nab_r	27.21%	77.82%
xz_r	144.76%	232.64%
mcf_r	83.74%	135.31%
lbm_r	97.31%	104.79%
Average	**109.73%**	**227.38%**

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.

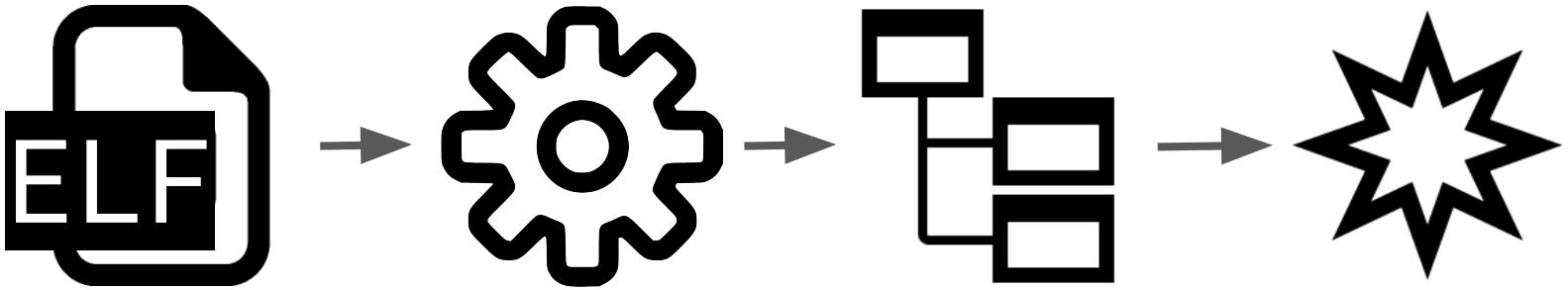
END


WARNING
The following slides contains
bright, rapidly flashing
ARM assembly
If this will be a problem
or negatively affect your health,
please do not watch.

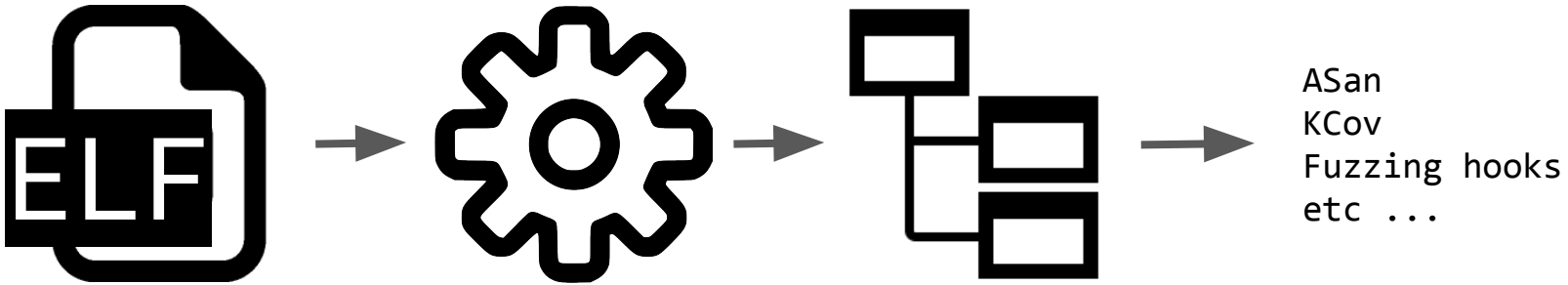
A quick recap of Retrowrite



A quick recap of retrowrite



A quick recap of retrowrite



A small history lesson

Sushant dinesh + Mathias Payer	original retrowrite tool for x86-64
Matteo Rizzo	kretrowrite for x86-64 kernel modules
Me + Jean-Michel	retrowrite for ARM 64 (aarch64)

A small history lesson

Sushant dinesh + Mathias Payer	original retrowrite tool for x86-64
Matteo Rizzo	kretrowrite for x86-64 kernel modules
Me + Jean-Michel	retrowrite for ARM 64 (aarch64)

www.github.com/hexhive/retrowrite	official public repo (old, but will get updated)
www.github.com/hexhive/ku_retrowrite	kernel version + ARM branch in development

Disclaimer

While aarch64 it’s an architecture we all love and cheer (well, at least before you actually start working with it), I realize that not everyone is comfortable reading ARM asm.

No worries, I’ve got you covered, any time you see the following symbol:

I will google-translate the ARM assembly back to our beloved x86!
You’re welcome.

The problems I’m facing

-

People is stupid

firefoxsucks

The problems I’m facing

- Detecting global variable access is hard
- Detecting switch statements (jump tables) is hard

Why are those necessary?
For the reassembly after the Instrumentation step to work correctly.
In short words, instrumentation is going to insert arbitrary assembly in the middle of the binary, to do that I need to relocate stuff correctly, and to do that I need to know where stuff *is* in the first place.

Part 1: Global variable access

Global variable access on x86

```
mov rax, [0x7fffff008]
```

Global variable access on ARM

```
adrp x0, 0x7fffff000
ldr x1, [x0, #8]
```

Global variable access on ARM

```
adrp x0, 0x7fffff000
ldr x1, [x0, #8]
```

```
mov rax, 0x7fffff000
add rax, 8
mov rbx, [rax]
```

Global variable access on ARM

```
adrp x0, 0x7fffff000
ldr x1, [x0, #8]
```

```
mov rax, 0x7fffff000
add rax, 8
mov rbx, [rax]
```

This is because ARM has fixed 4-byte instructions, so you cannot fit a 64 bit address in them, you need to compute it using adrp (with which you can only select a 4KB page).

Global variable access on ARM

```
adrp x0, 0x7fffff000
ldr x1, [x0, #8]
```

```
mov rax, 0x7fffff000
add rax, 8
mov rbx, [rax]
```

This is because ARM has fixed 4-byte instructions, so you cannot fit a 64 bit address in them, you need to compute it using adrp (with which you can only select a 4KB page).

This is a problem ‘cause the global address can (and will) be built in much more convoluted ways than the example above

Pattern matching

Pattern matching

```
adrp x0, <page>
...
ldr x1, [x0 + <off>]
```

Detected

Pattern matching

```
adrp x0, <page>
...
ldr x1, [x0 + <off>]
```

Detected

```
adrp x0, <page>
...
add x0, x0, <off>
...
ldr x1, [x0]
```

Detected

Pattern matching

```
adrp x0, <page>
...
ldr x1, [x0 + <off>]
```

Detected

```
adrp x0, <page>
...
add x0, x0, <off>
...
ldr x1, [x0]
```

Detected

```
adrp x0, <page>
...
ldr x1, [memory]
...
add x0, x0, x1
ldr x2, [x0]
```

NOT detected

Pattern matching

```
adrp x0, <page>
...
ldr x1, [x0 + <off>]
```

Detected

```
adrp x0, <page>
...
add x0, x0, <off>
...
ldr x1, [x0]
```

Detected

```
adrp x0, <page>
...
ldr x1, [memory]
...
add x0, x0, x1
ldr x2, [x0]
```

NOT detected

The first two cases cover 99% of global accesses, especially with no compiler optimization.

Pattern matching

- I stop doing this and instead use a full-blown emulator, such as Unicorn or QEMU
  - This is not great, it will slow down progress by at least 1-2 weeks as I'm not familiar with any of them.
- I continue to improve the pattern matching, adding more and more edge cases
  - This is not well going in the long run, as even detecting the "simple" cases shown before was not easy at all, and the complexity of the code is exploding pretty quickly. At some point, this will become just me implementing my own emulator

Part 2: Switches!

Switches on ARM

```
adr x1, <first case addr>
adrp x19, <page>
ldrb w0, [x19, <off>]
add x0, x1, w0, sxtb 2
br x0
```

Switches on ARM

```
adr x1, <first case addr>
adrp x19, <page>
ldrb w0, [x19, <off>]
add x0, x1, w0, sxtb 2
br x0
```

```
mov rbx, <first case addr>
mov rax, word [<page>+<off>]
sal rax, 2
add rbx, rax
jmp rbx
```

Switches on ARM

```
adr x1, <first case addr>
adrp x19, <page>
ldrb w0, [x19, <off>]
add x0, x1, w0, sxtb 2
br x0
```

```
mov rbx, <first case addr>
mov rax, word [<page>+<off>]
sal rax, 2
add rbx, rax
jmp rbx
```

```
- offset -   0 1  2 3
0x00020d74  001f 1f17
0x00020d78  1f1f 0e1f
0x00020d7c  081f 1f1f
0x00020d80  1f1f 1c00
```

Switches on ARM

```
adr x1, <first case addr>
adrp x19, <page>
ldrb w0, [x19, <off>]
add x0, x1, w0, sxtb 2
br x0
```

```
mov rbx, <first case addr>
mov rax, word [<page>+<off>]
sal rax, 2
add rbx, rax
jmp rbx
```

```
- offset -   0 1  2 3
0x00020d74  001f 1f17
0x00020d78  1f1f 0e1f
0x00020d7c  081f 1f1f
0x00020d80  1f1f 1c00
```

The observant reader will notice that this is just a particular case of global variable access.
I need to symbolize *all* possible cases of the switch as while instrumenting the offsets will change for sure!

Yes but what about other disassemblers?

- Radare2:
  - Correctly detects jump tables on x86, but not on ARM. They use their own implementation of code emulation called ESIL.
- IDA:
  - Correctly detects jump tables on both x86 and ARM. I have no idea on what they do use.
- Ghidra:
  - **Correctly detects jump tables on both x86 and ARM.** I am almost sure they use their own symbolic execution**, but I didn’t spend that much time digging the java code**.

Yes but what about other disassemblers?

- Radare2:
  - Correctly detects jump tables on x86, but not on ARM. They use their own implementation of code emulation called ESIL.
- IDA:
  - Correctly detects jump tables on both x86 and ARM. I have no idea on what they do use.
- Ghidra:
  - **Correctly detects jump tables on both x86 and ARM.** I am almost sure they use their own symbolic execution**, but I didn’t spend that much time digging the java code**.

ghidra / Ghidra / Features / Base / src / main / java / ghidra / util / state / analysis / **RelativeJumpTableSwitch.java**

Conclusions?

- More pattern matching
- Code emulation
- Symbolic execution

A quick recap of retrowrite