Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing

Zachary Yedidia

Stanford University

Background: Software-based Fault Isolation (SFI)

Goal: Isolate untrusted programs within a single address space.

Language-based security

source language
↓ .c
compile
↓ .wasm
verify
↓ .wasm
compile
↓ .elf
run

Classic SFI

source language
↓ .c
compile
↓ .s
instrument
↓ .elf
verify
↓ .elf
run

Key idea: Verify machine code directly, so that the compiler can be untrusted.

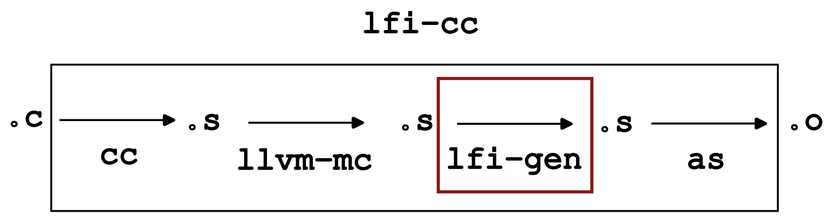
ARM64 Overview

- Fixed-width encoding.
- 32 64-bit registers (x0-x30, sp).
- Stack pointer register (sp).
- Dedicated return address register (x30).
- 32-bit register subsets (w0-w30, wsp).
- A 32-bit addressing mode.

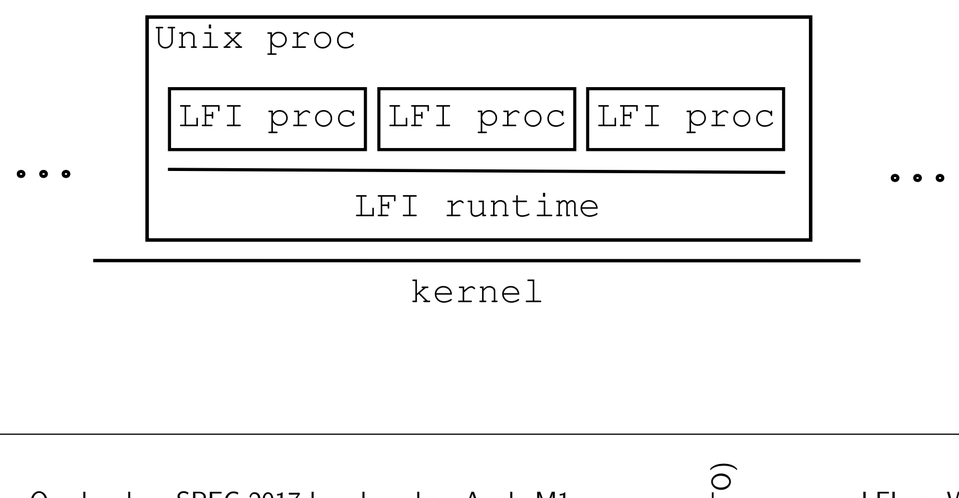
```
<fib>:
a9be53f3  stp   x19, x20, [sp, #-32]!
2a0003f3  mov   w19, w0
52800014  mov   w20, #0x0
f9000bfe  str   x30, [sp, #16]
34000113  cbz   w19, 30 <fib+0x30>
7100067f  cmp   w19, #0x1
540000c0  b.eq  30 <fib+0x30>
51000660  sub   w0, w19, #0x1
51000a73  sub   w19, w19, #0x2
94000000  bl    0 <fib>
0b000294  add   w20, w20, w0
17fffff9  b     10 <fib+0x10>
0b140260  add   w0, w19, w20
f9400bfe  ldr   x30, [sp, #16]
a8c253f3  ldp   x19, x20, [sp], #32
d65f03c0  ret
```

Implementation

1. Assembly transformer: 2,000 LoC (untrusted).



2. Static verifier: 323 lines of Rust, 30 MB/s throughput (trusted).
3. Runtime implementation: 4,000 LoC (trusted).



Lightweight Fault Isolation (LFI)

This work presents LFI, a new SFI system for ARM64.

- **Simple** implementation made possible by "peephole sandboxing."
- Low runtime overhead: **6.5%**.
- Many sandboxes in a single address space (around **65,000**).
- Fast and simple static verifier (**secure**).

LFI Sandboxing Scheme

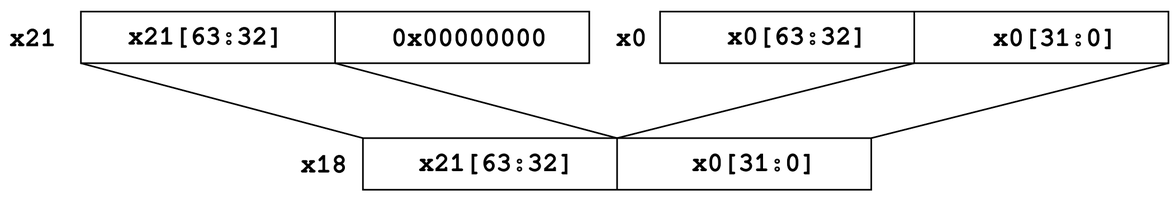
Reserved registers:

- x21: sandbox base address (4GiB-aligned).
- x18: contains a valid sandbox address.
- x30: contains a valid sandbox address.
- sp: contains a valid sandbox address.

```
ldr x0, [x18]     // safe
str x0, [sp, #8] // safe
blr x18           // safe
blr x30           // safe
ldr x0, [x1]    // unsafe
svc #0          // unsafe
```

Question: safely write to x18?

```
mov x18, x0           // unsafe
add x18, x21, w0, uxtw // safe
```



Original code	Sandboxed equivalent
`br xN`	`add x18, x21, wN, uxtw` `br x18`
`ldr rt, [xN]`	`add x18, x21, wN, uxtw` `ldr rt, [x18]`
`ldr x30, [x18]`	`ldr x30, [x18]` `add x30, x21, w30, uxtw`

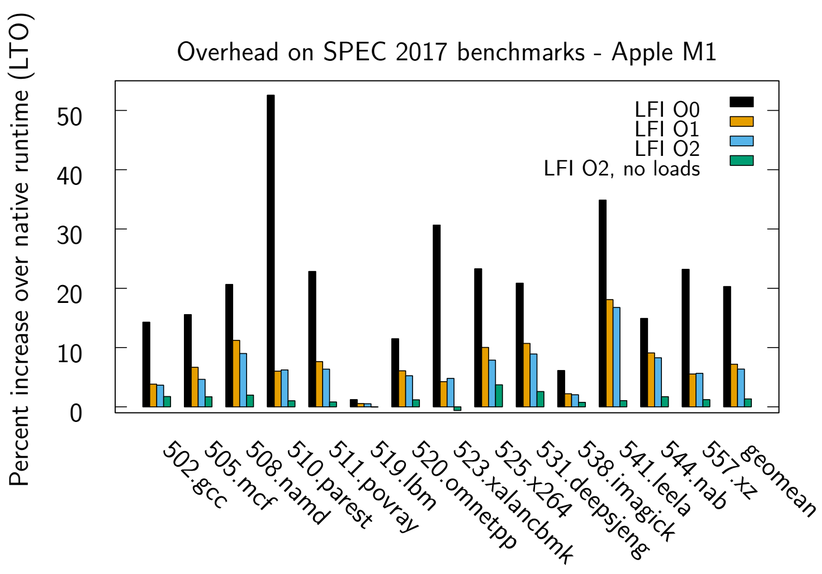
Optimization: perform the guard inside a load/store addressing mode.

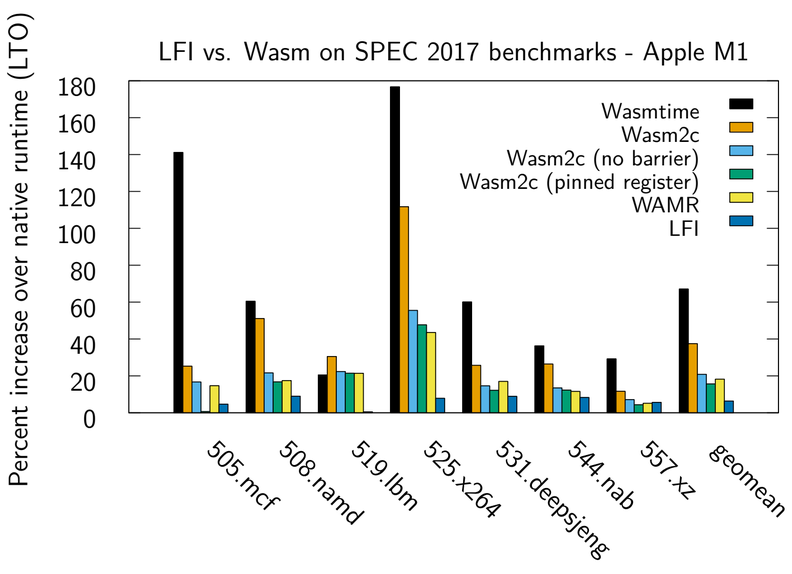
Original code	Sandboxed equivalent
`ldr rt, [xN]`	`ldr rt, [x21, wN, uxtw]`

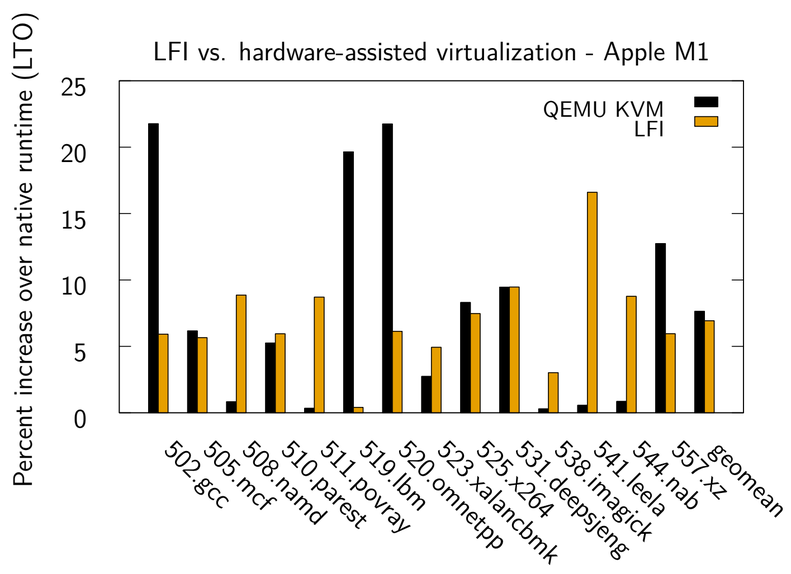
Runtime calls: first page (read-only).

```
ldr x30, [x21, #n]
blr x30
```

Evaluation







System	Geomean (T2A)	Geomean (M1)
Wasmtime	47.0%	67.1%
Wasm2c	40.7%	37.5%
Wasm2c (no barrier)	21.5%	20.8%
Wasm2c (pinned reg)	16.5%	15.7%
WAMR	22.3%	18.2%
LFI	7.3%	6.4%

Microbenchmarks: GCP T2A VM, 2.8 GHz

Platform	Syscall (ns)	Ctxsw (ns)
LFI	26	46
Linux	162	2,494
gVisor	12,019	22,899

Microbenchmarks: Apple M1, 3.2 GHz

Platform	Syscall (ns)	Ctxsw (ns)
LFI	22	48
Linux	129	1,504