Classic Software Fault Isolation and WebAssembly

Zachary Yedidia

Stanford University

Outline

Part 1: Overview of software sandboxing.

Part 2: Lightweight Fault Isolation (LFI).

Part 3: WebAssembly and LFI.

Isolation and Sandboxing

Isolation is central to multi-tenant systems.

Hardware-based

- Memory protection and privilege levels.
- Hardware-assisted virtualization.

Challenges

- Slow to start/stop.
- Slow to context switch.
- Requires hardware/OS support.

Isolation and Sandboxing

Isolation is central to multi-tenant systems.

Software-based

- Language-based security.
- Classic Software Fault Isolation (SFI).

Challenges

- Overhead.
- Trusted software.
- . . .

Software Sandboxing: Desires

Efficiency

- Low overhead.
- Low context switch time.

Security

- Small trusted code base (TCB).
- Spectre-hardened.

Simplicity

- Simple to design/implement.

Scalability

- Thousands of tenants.

Flexibility

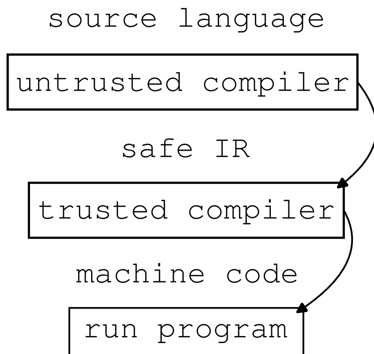
- Configurable sandbox size.
- Unrestricted source language support.

Portability

- Multi-architecture.
- Self-contained.

Language-based Security

Approach: enforce security properties in a source/intermediate language that is then compiled into a binary.



Language-based Security

Approach: enforce security properties in a source/intermediate language that is then compiled into a binary.

Pros

- Portable: source language can target multiple architectures.
- Powerful: many types of safety properties can be enforced.

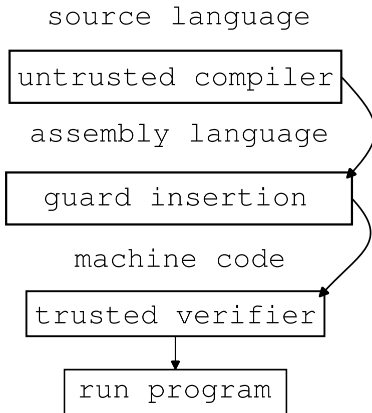
Cons

- Large trusted code base (TCB): compiler is generally fully trusted.
- Complex to design/implement.
- Source language may be restricted.

Examples: WebAssembly, JVM, eBPF, Singularity OS (C#).

Classic Software-based Fault Isolation (SFI)

Approach: before running a compiled binary, verify it to make sure it will not violate security properties.



Classic Software-based Fault Isolation (SFI)

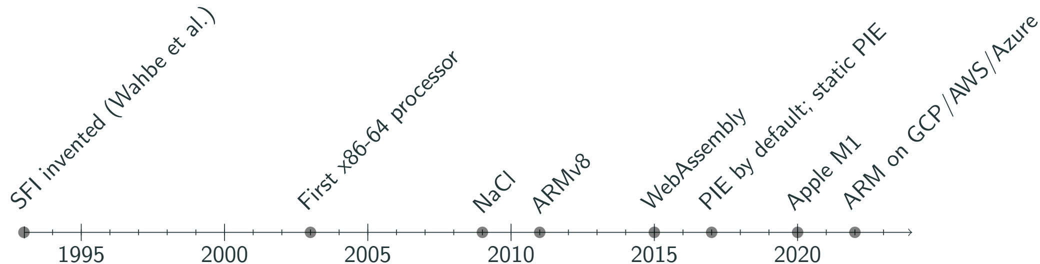
Approach: before running a compiled binary, verify it to make sure it will not violate security properties.

Two components: an untrusted compiler that can generate binaries that pass verification, and a verifier to validate the generated binaries.

- Small TCB.
- Unrestricted source language.
- Very low overhead, while still supporting many tenants (claim).
- Simple to implement (claim).

Examples: NaCl, LFI.

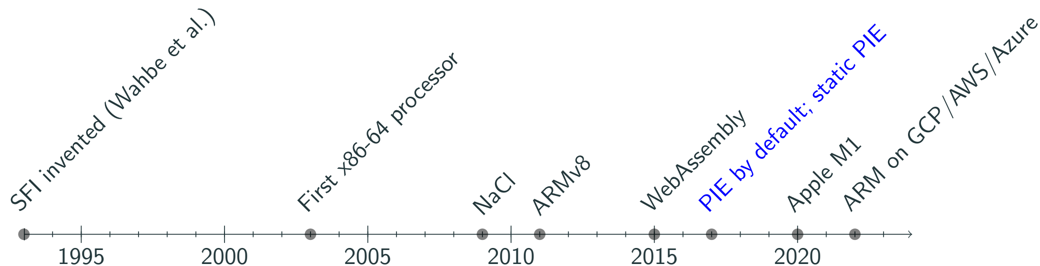
What has changed?



Key points:

- Position-independent code is ubiquitous (2017+).
- ARM64 is starting to see widespread deployment (2020+).

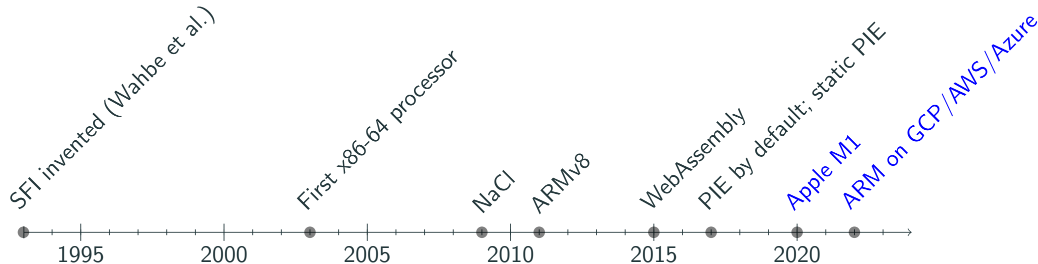
What has changed?



Key points:

- Position-independent code is ubiquitous (2017+).
- ARM64 is starting to see widespread deployment (2020+).

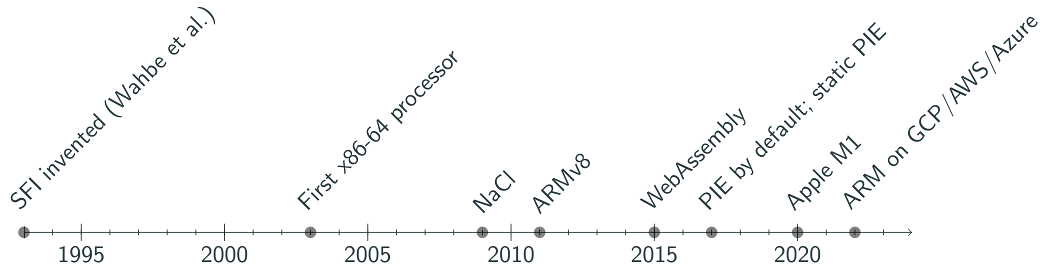
What has changed?



Key points:

- Position-independent code is ubiquitous (2017+).
- ARM64 is starting to see widespread deployment (2020+).

What has changed?



Key points:

- Position-independent code is ubiquitous (2017+).
- ARM64 is starting to see widespread deployment (2020+).

Claim: ARM64 is very amenable to efficient SFI.

Lightweight Fault Isolation (LFI)

A new SFI system that is simple and low overhead.

- Supports 32K or 64K[1] sandboxes per address space.
- Low overhead: 7% runtime and 14% code size overhead on SPEC 2017.
- No modifications to existing compiler source code.
- Fast and simple static binary verifier (small TCB).
- No alignment constraints.
- Spectre-resistant to some extent.
- Targets ARM64.

[1] ARM has two pagetables: size of virtual address space depends on whether both are accessible.

Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding.
- 31 64-bit registers (x0-x30).
- Stack pointer register (sp).
- Dedicated return address register (x30).
- 32-bit register subsets (w0-w30).
- 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
```

Overview of ARM64

Important ARM64 features for SFI:

- **Fixed-width encoding.**
- 31 64-bit registers (x0-x30).
- Stack pointer register (sp).
- Dedicated return address register (x30).
- 32-bit register subsets (w0-w30).
- 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
```

Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding.
- **31 64-bit registers (`x0`-`x30`).**
- **Stack pointer register (`sp`).**
- **Dedicated return address register (`x30`).**
- 32-bit register subsets (`w0`-`w30`).
- 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
```

Overview of ARM64

Important ARM64 features for SFI:

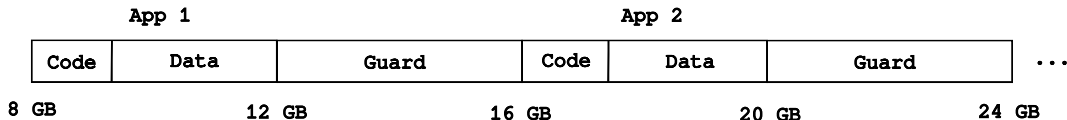
- Fixed-width encoding.
- 31 64-bit registers (x0-x30).
- Stack pointer register (sp).
- Dedicated return address register (x30).
- **32-bit register subsets (w0-w30).**
- **A 32-bit addressing mode.**

```
<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
```

LFI Sandbox Environment

Each sandbox is given 4GB of virtual memory, with 4GB guard pages on both sides.

The MMU prevents writing code and executing data.

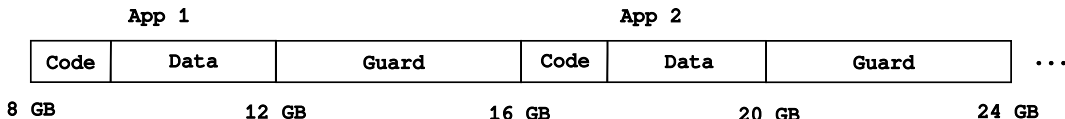


Note: code is statically linked and position-independent.

LFI Sandbox Environment

Each sandbox is given 4GB of virtual memory, with 4GB guard pages on both sides.

The MMU prevents writing code and executing data.



How many sandboxes can we fit in the virtual address space?

Page size	User pagetable	User+kernel pagetable
4KB	32K	64K
64KB	512K	1024K

Note: extended address spaces for 64KB pages require `FEAT_LVA` from ARMv8.2.

Basic Implementation: Registers

Special/reserved registers:

- `x21`: sandbox base address (aligned to 4GB).
- `x18`: always contains a valid sandbox address.
- `x30`: always contains a valid branch target (sandbox or runtime call address).
- `sp`: always contains a valid sandbox address.

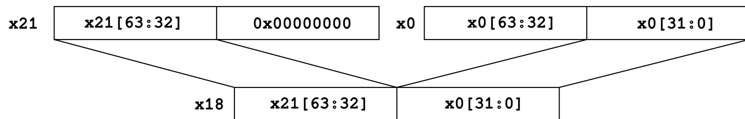
Reserved registers may only be modified in ways that maintain these invariants.

```
ldr rt, [x18]     // safe
str rt, [sp, #8]  // safe
br x18            // safe
br x30            // safe
```

Basic Implementation: Guard Instruction

How to safely modify a reserved register?

```
mov x18, x0 // unsafe
```




```
add x18, x21, w0, uxtw // safe
```

Note: this instruction executes with 2-cycle latency.

Sandboxing Memory Accesses

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

Note: skipping guards is legal (same trick from the original '93 SFI paper).

→ No bundle alignment or CFI requirements.

ARM64 Addressing Modes

Addressing mode	Generated address
`[xN]`	`addr = xN`
`[xN, #i]`	`addr = xN + i`
`[xN, #i]!`	`addr = xN + i; xN += i`
`[xN], #i`	`addr = xN; xN += i`
`[xN, xM, lsl #i]`	`addr = xN + xM << #i`
`[xN, wM, uxtw #i]`	`addr = xN + zx(wM) << #i`
`[xN, wM, sxtw #i]`	`addr = xN + sx(wM) << #i`

Guard pages ensure `[x18, #i]` never accesses another sandbox.

Register-register modes must be re-written.

Note: some loads/stores (`ldp`, atomics, ...) cannot use the complex modes.

ARM64 Addressing Modes

Addressing mode	Generated address
[xN]	addr = xN
[xN, #i]	addr = xN + i
[xN, #i]!	addr = xN + i; xN += i
[xN], #i	addr = xN; xN += i
[xN, xM, lsl #i]	addr = xN + xM << #i
[xN, wM, uxtw #i]*	addr = xN + zx(wM) << #i
[xN, wM, sxtw #i]	addr = xN + sx(wM) << #i

Guard pages ensure [x18, #i] never accesses another sandbox.

Register-register modes must be re-written.

Note: some loads/stores (ldp, atomics, . . .) cannot use the complex modes.

Optimization: 32-bit Addressing Modes

Original code	Sandboxed equivalent	Cycles of overhead
`ldr rt, [xN]`	`ldr rt, [x21, wN, uxtw]`	0

Optimization: 32-bit Addressing Modes

Original code	Sandboxed equivalent	Cycles of overhead
`ldr rt, [xN]`	`ldr rt, [x21, wN, uxtw]`	0
`ldr rt, [xN, #i]`	`add w24, wN, #i` `ldr rt, [x21, w24, uxtw]`	1
`ldr rt, [xN, #i]!`	`add xN, xN, #i` `ldr rt, [x21, wN, uxtw]`	1
`ldr rt, [xN], #i`	`ldr rt, [x21, wN, uxtw]` `add xN, xN, #i`	1
`ldr rt, [xN, xM, lsl #i]`	`add w24, wN, wM, lsl #i` `ldr rt, [x21, w24, uxtw]`	2
`ldr rt, [xN, wM, uxtw #i]`	`add w24, wN, wM, uxtw #i` `ldr rt, [x21, w24, uxtw]`	2
`ldr rt, [xN, wM, sxtw #i]`	`add w24, wN, wM, sxtw #i` `ldr rt, [x21, w24, uxtw]`	2

Optimization: Guard Hoisting

Introduce two more reserved registers:

- x22: always valid.
- x23: always valid.

```
ldr x2, [x1, #8]
str x2, [x0, #8]
ldr x2, [x1, #16]
str x2, [x0, #16]
ldr x2, [x1, #24]
str x2, [x0, #24]
```

```
add x22, x21, w0, uxtw
add x23, x21, w1, uxtw
ldr x2, [x23, #8]
str x2, [x22, #8]
ldr x2, [x23, #16]
str x2, [x22, #16]
ldr x2, [x23, #24]
str x2, [x22, #24]
```

Optimization: Stack Pointer

The `sp` register is assumed to always contain a valid address.

$\rightarrow$ No guards necessary for stack accesses.

Guards are necessary when modifying `sp`, but not in all cases.

Original code	Sandboxed equivalent
`add sp, sp, #n`	`add w24, wsp, #n` `add sp, x21, x24`

`add sp, sp, #n`
. . . (no branches) . . .
`ldr rt, [sp, #m]`
	No change necessary

`str rt, [sp, #n]!`	No change necessary

Runtime Calls

How to safely call a runtime routine outside the sandbox?

Runtime Calls

How to safely call a runtime routine outside the sandbox?

Idea 1: reserve yet another register to store the runtime entrypoint.

Runtime Calls

How to safely call a runtime routine outside the sandbox?

Idea 2: use the first page of the sandbox to store the runtime call table (read-only).

Runtime Calls

How to safely call a runtime routine outside the sandbox?

Idea 2: use the first page of the sandbox to store the runtime call table (read-only).

- The address of the runtime call table is already stored in `x21`!
- Use `x30`, a special register used only for valid branch targets.

Runtime Calls

How to safely call a runtime routine outside the sandbox?

Idea 2: use the first page of the sandbox to store the runtime call table (read-only).

- The address of the runtime call table is already stored in `x21`!
- Use `x30`, a special register used only for valid branch targets.

```
svc #0
```

```
str x30, [sp, -16]!
ldr x30, [x21, #n]
blr x30
ldr x30, [sp], 16
add x30, x21, w30, uxtw
```

Benefit: application can select the exact runtime call it wants statically.

Implementation: Assembly Transformation

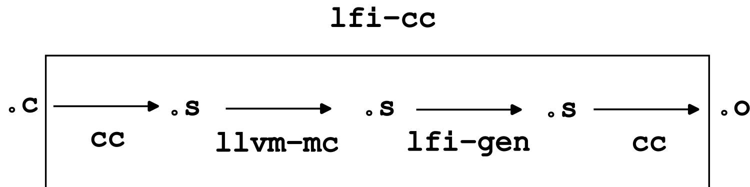
Problem: How to implement without modifying/maintaining a compiler toolchain?

Implementation: Assembly Transformation

Problem: How to implement without modifying/maintaining a compiler toolchain?

Solution: Text processing pass on GNU assembly files (`.s`).

Use `llvm-mc` to normalize the assembly (externally maintained by LLVM).



`lfi-gen` is a 2,000 line Go program.

Implementation: Compiler Toolchain

A full C/C++ LLVM compiler toolchain can be built using the following libraries:

- `compiler-rt`.
- `musl-libc`.
- `libc++`.
- `libc++abi`.
- `libunwind`.

Installation process: download LFI-instrumented sysroot and LFI wrapper tools. Use your system's default Clang/LLVM package.

Implementation: Compiler Toolchain

A full C/C++ LLVM compiler toolchain can be built using the following libraries:

- `compiler-rt`.
- `musl-libc`.
- `libc++`.
- `libc++abi`.
- `libunwind`.

Installation process: download LFI-instrumented sysroot and LFI wrapper tools. Use your system's default Clang/LLVM package.

Why no GCC? The GCC build system is less modular, so building LFI-instrumented `libgcc` is more gnarly (but should be possible).

Static Verifier

- Core logic is 290 lines of Rust (not counting disassembler).
- No alignment requirements.
- Verifies executable ELF segments (link with `--rosegment` to separate code/data).
- Verifies 30 MB/s on a Macbook Air.

Static Verifier

- Core logic is 290 lines of Rust (not counting disassembler).
- No alignment requirements.
- Verifies executable ELF segments (link with `--rosegment` to separate code/data).
- Verifies 30 MB/s on a Macbook Air.

Uses the ARM Machine Readable Specification:

- Find all instructions that can branch or read/write memory.
- Find all instructions that can modify a register.
- Generate the disassembler.

BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	Rn					0	0	0	0	0
							Z		op											A	M						Rm				

BLR <Xn>

```
integer n = UInt(Rn);
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

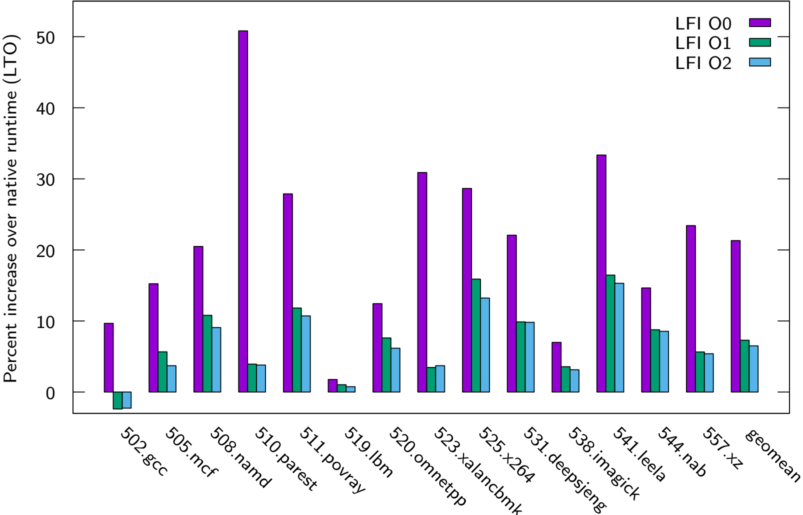
Operation

```
bits(64) target = X[n, 64];

if HaveGCS() && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC[] + 4);
X[30, 64] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
```

LFI Overhead



WebAssembly and LFI

Portability

WebAssembly as a portable IR.

- Wasm binaries encode all necessary dependencies.
- Wasm binaries can target multiple architectures.

Isolation

Ensure program cannot escape sandbox.

- Bounds checks/32-bit enforcement.
- Dynamic indirect branch checks.

Goal: decouple portability and isolation.

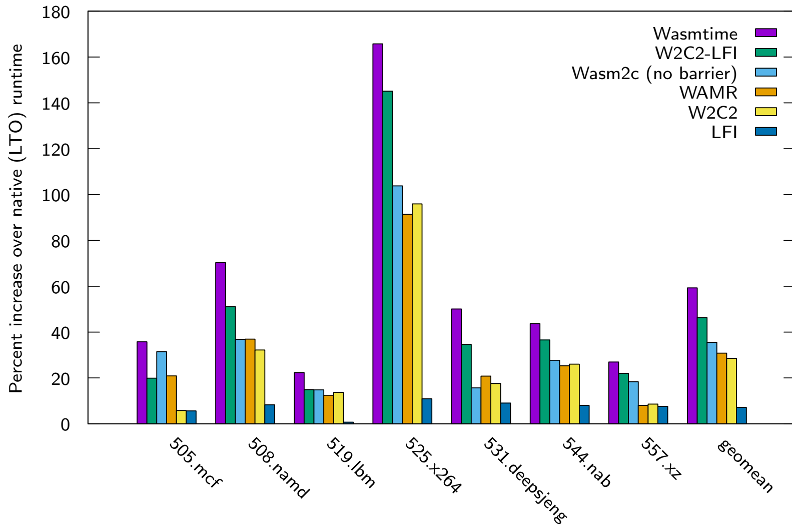
WebAssembly for portability, LFI for isolation.

WebAssembly and LFI: Overhead

The contestants:

- Wasmtime: widely-used Wasm compiler built on Cranelift.
- WAMR: ahead-of-time Wasm compiler using LLVM.
- Wasm2c: Wasm to C convertor, then compiled with Clang (barrier removed).
- W2C2: Wasm to C convertor, then compiled with Clang (not full sandboxing).
- W2C2-LFI: W2C2 running within an LFI sandbox.
- LFI.

WebAssembly and LFI: Overhead



WebAssembly and LFI: Overhead

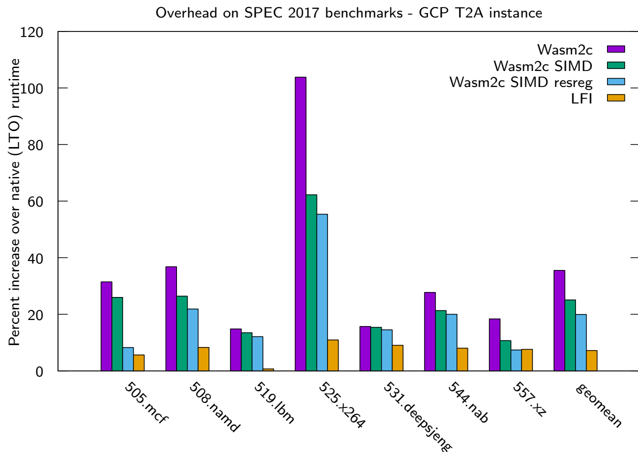
Result: LFI overhead is 7% by default.

Result: LFI causes an additional 20% overhead when used with WebAssembly.

$\rightarrow$ tradeoff for security (TCB size), and ability to safely precompile.

Optimizing WebAssembly Performance on SPEC 2017

1. Enable the SIMD 128 proposal.
2. Reserve a register for the heap base (not supported for all architectures).



Thank you!

You can follow further development at:

https://github.com/zyedidia/lfi

Spectre Safety

LFI does not rely on any fine-grained control-flow integrity for sandbox correctness.

$\rightarrow$ Speculative sandbox breakout attacks are mitigated.

Spectre Safety

LFI does not rely on any fine-grained control-flow integrity for sandbox correctness.

$\rightarrow$ Speculative sandbox breakout attacks are mitigated.

Problem: Speculative cross-sandbox and host poisoning attacks.

Spectre Safety

LFI does not rely on any fine-grained control-flow integrity for sandbox correctness.

→ Speculative sandbox breakout attacks are mitigated.

Problem: Speculative cross-sandbox and host poisoning attacks.

Solution: ARM software context numbers.

D13.2.121 SCXTNUM_EL0, EL0 Read/Write Software Context Number

The SCXTNUM_EL0 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL0 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

Configurations

This register is present only when FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented. Otherwise, direct accesses to SCXTNUM_EL0 are UNDEFINED.

Attributes

SCXTNUM_EL0 is a 64-bit register.