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

Zachary Yedidia

Stanford University

# Outline

**Part 1**: Overview of sandboxing techniques.

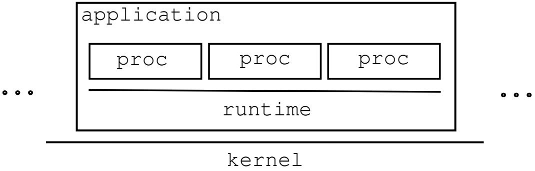
**Part 2**: Lightweight Fault Isolation (LFI).

**Part 3**: Evaluation and discussion.

# Usermode Sandboxing

Applications:

- Serverless computing, FaaS, cloud computing.
- Web browsers.
- Software compartmentalization.



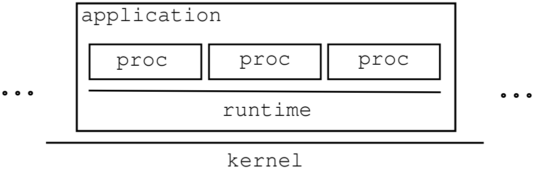
# Usermode Sandboxing

Applications:

- Serverless computing, FaaS, cloud computing.
- Web browsers.
- Software compartmentalization.

Techniques:

- Hardware: multiple address spaces.
- Software: single address space.



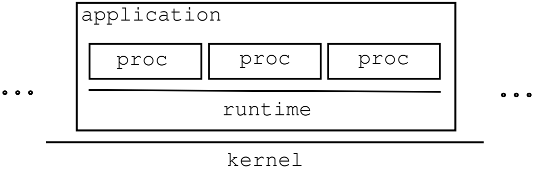
# Usermode Sandboxing

Applications:

- Serverless computing, FaaS, cloud computing.
- Web browsers.
- Software compartmentalization.

Techniques:

- Virtualization (Dune).
- Containerization (gVisor).
- Software sandboxing (WebAssembly).



# Usermode Sandboxing: Performance

Two primary performance metrics:

## CPU overhead

| Approach | CPU overhead[1] |
|---|---|
| Wasm+LLVM | ˜25% |
| Wasm+Cranelift | ˜60-90% |
| Virtualization | ˜5% |
| gVisor | ˜0% |

## Context switch time

| Approach | Ctxsw (cyc) | Syscall (cyc) |
|---|---|---|
| Software switch | ˜70 | ˜70 |
| Hardware switch | ˜500 | ˜300 |
| Linux | ˜3,000 | ˜300 |
| gVisor | ˜20,000 | ˜10,000 |

Hardware protection incurs some additional switch overhead when virtualized.

[1] Measured on a subset of SPEC 2017.

# Usermode Sandboxing: Performance

Two primary performance metrics:

### CPU overhead

| Approach | CPU overhead[1] |
|---|---|
| Wasm+LLVM | ˜25% |
| Wasm+Cranelift | ˜60-90% |
| Virtualization | ˜5% |
| gVisor | ˜0% |

### Context switch time

| Approach | Ctxsw (cyc) | Syscall (cyc) |
|---|---|---|
| Software switch | ˜70 | ˜70 |
| Hardware switch | ˜500 | ˜300 |
| Linux | ˜3,000 | ˜300 |
| gVisor | ˜20,000 | ˜10,000 |

Hardware protection incurs some additional switch overhead when virtualized.

LFI (this work): 6% CPU overhead, with software switching.

[1] Measured on a subset of SPEC 2017.

## Software Sandboxing

**Goal**: isolate without the need to change hardware structures when context switching.

Approaches:

**Language-based security (LBS)**

Use a safe source/intermediate language that is then compiled to machine code.

Examples: WebAssembly, eBPF, JVM.

**Classic software fault isolation[2] (SFI)**

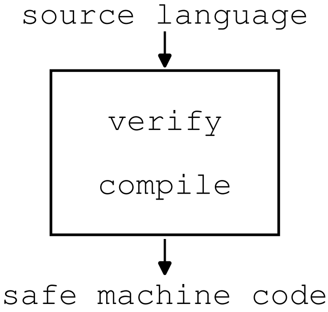
Use a machine code verifier to ensure a binary is safe before running it.

Examples: PittSFIeld, Native Client, LFI.
Note: Native Client is single-sandbox SFI.

[2] Wahbe et al., SOSP 1993.

# LBS vs. SFI: Approach

**LBS**



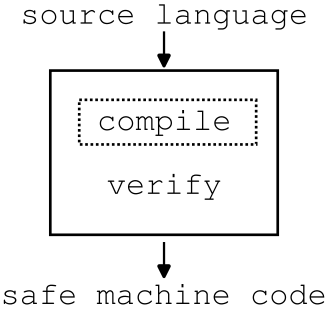
source language

verify

compile

safe machine code

**Classic SFI**



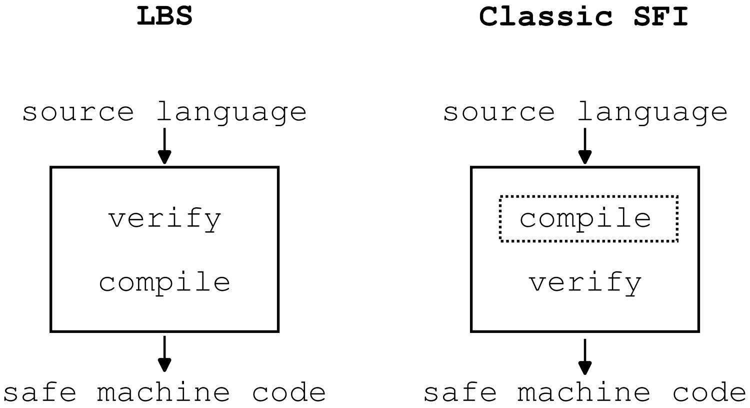
source language

compile

verify

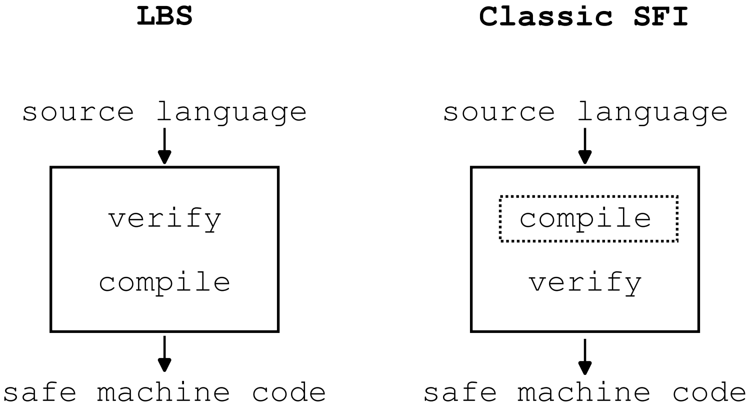
safe machine code

# LBS vs. SFI: Approach



The verifier, and every step afterwards, is trusted.

# LBS vs. SFI: Approach



**LBS**

source language

verify

compile

safe machine code

**Classic SFI**

source language

compile

verify

safe machine code

Problem: trusting a language verifier and compiler can be dangerous.

# LBS Danger #1: the language verifier

Many “safe” languages are not designed with isolation in mind.

# LBS Danger #1: the language verifier

## Many “safe” languages are not designed with isolation in mind.

Picking on Rust: is Safe Rust actually safe?

```
trait Object<U> { type Output; }
impl<T: ?Sized, U> Object<U> for T { type Output = U; }
fn transmute_obj<T: ?Sized, U>(x: <T as Object<U>>::Output) -> U { x }
fn transmute<T, U>(x: T) -> U { transmute_obj::<dyn Object<U, Output = T>, U>(x) }

fn main() {
    // make a null pointer
    let p = core::ptr::null_mut();
    // "safely" transmute it into a reference
    let x = transmute::<*mut i64, &’static i64>(p);
    // access the reference
    println!("x: {}", *x);
}
```

```
% cargo run
Segmentation fault (core dumped)
```

github.com/rust-lang/rust#57893 has been open since 2019 with no resolution in sight.

# LBS Danger #1: the language verifier

From the Rust issue tracker:

- 81 open unsoundness bugs.
- 20/81 are LLVM-related.
- 32/81 are marked high priority.

Conclusion: simpler languages like WebAssembly or eBPF are easier to validate.

Note: these languages are still not necessarily easier to validate than machine code.

→ Validation logic is still thousands of lines of code.

# LBS Danger #2: the compiler

Compilers are not necessarily designed with isolation in mind.

LLVM (not designed for isolation):

- 2 million lines of code.
- 242 open miscompilation bugs.
- not hardened vs. malicious input.

Cranelift (designed for isolation):

- “only” 200,000 lines of code.
- only 2 sandbox-escape CVEs due to miscompilation so far.
- avoids quadratic-time algorithms.

Tradeoff: performance vs. security.

Even “secure” JIT compilers are complicated and have bugs[3].

[3] https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html

# Classic Software-based Fault Isolation (SFI)

Solves the trusted compiler problem with an arch-specific verifier (Wahbe, 1993).

→ Make machine code the source language — trusted compiler no longer necessary.

Two components:

1. An untrusted compiler that can generate binaries that pass verification.
2. A static verifier to validate the generated binaries.

# Classic Software-based Fault Isolation (SFI)

Solves the trusted compiler problem with an arch-specific verifier (Wahbe, 1993).

→ Make machine code the source language — trusted compiler no longer necessary.

Two components:

1. An untrusted compiler that can generate binaries that pass verification.
2. A static verifier to validate the generated binaries.

**Key**: A verifier is much simpler than a compiler.

→ Also easier to formally verify.

# Classic Software-based Fault Isolation (SFI)

Solves the trusted compiler problem with an arch-specific verifier (Wahbe, 1993).

→ Make machine code the source language — trusted compiler no longer necessary.

Two components:

1. An untrusted compiler that can generate binaries that pass verification.
2. A static verifier to validate the generated binaries.

Overhead and complexity are still problems:

- Prior multi-sandboxing SFI systems have overheads of $> 20\%$.
- No actively developed SFI systems currently exist.

# Lightweight Fault Isolation

This work presents Lightweight Fault Isolation (LFI), an SFI system that:

- Has low runtime overhead ($< 10\%$).
- Supports 10,000+ sandboxes.
- Is available on commodity hardware.

Performance **and** security!

Not just equivalent performance: significantly better than WebAssembly+LLVM.

→ High-performance and secure software-based multi-sandboxing system.

# Lightweight Fault Isolation

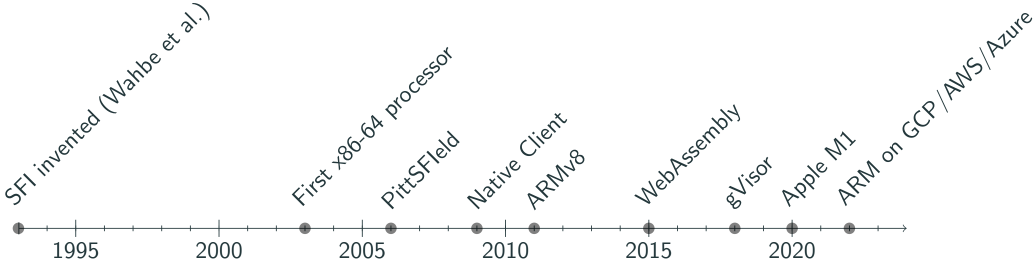
More details:

- Scalable: supports 32K or 64K sandboxes per address space[4].
- Low overhead: 6% runtime and 14% code size overhead on SPEC 2017 subset.
- Simple: no modifications to existing compiler source code.
- Secure: fast and simple static binary verifier.
- Compatible with Spectre mitigations.
- Targets ARM64.

Key insight: the design of the ARM64 ISA makes it amenable to efficient SFI.

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

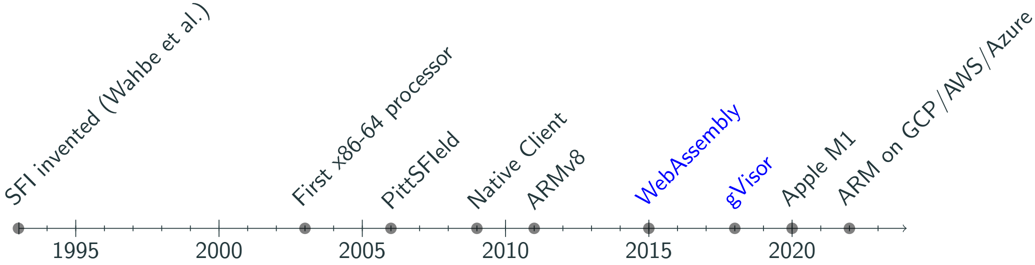
# Why now?



Key points:

- Cloud and serverless computing increasingly demand lightweight isolation.
- ARM64 is starting to see widespread deployment (2020+).

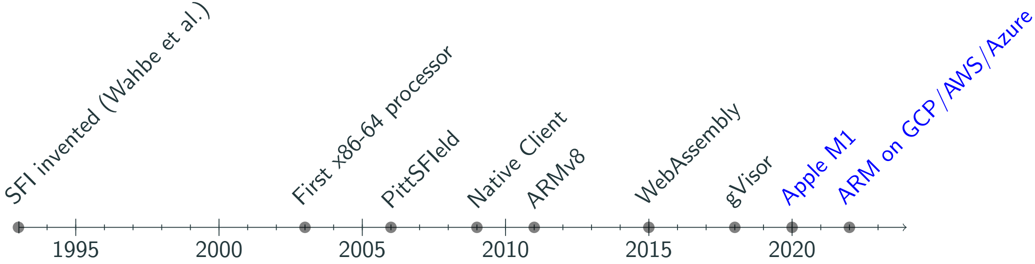
# Why now?



Key points:

- Cloud and serverless computing increasingly demand lightweight isolation.
- ARM64 is starting to see widespread deployment (2020+).

# Why now?



Key points:

- Cloud and serverless computing increasingly demand lightweight isolation.
- ARM64 is starting to see widespread deployment (2020+).

# Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding[5].
- 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
```

[5] ARM32's thumb mode was removed in ARM64.

# Overview of ARM64

Important ARM64 features for SFI:

- **Fixed-width encoding[5].**
- 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
```

[5] ARM32's thumb mode was removed in ARM64.

# Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding[5].
- **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
```

[5] ARM32's thumb mode was removed in ARM64.

# Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding[5].
- 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.**

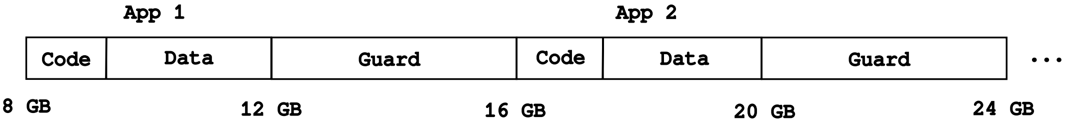
```
<fib>:
a9be53f3  stp   x19, x20, [sp, #-32]!
2a0003f3  mov   w19, w0
52800014  mov   w20, #0x0
f9000bfe  str   x30, [sp, #16]
34000113  cbz   w19, 30 <fib+0x30>
7100067f  cmp   w19, #0x1
540000c0  b.eq  30 <fib+0x30>
51000660  sub   w0, w19, #0x1
51000a73  sub   w19, w19, #0x2
94000000  bl    0 <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
```

[5] ARM32's thumb mode was removed in ARM64.

# 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.

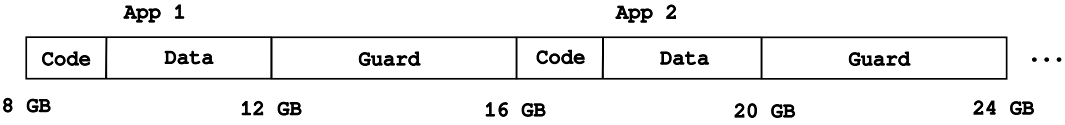


Note: code is statically linked and position-independent.

# 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: Overview

Compiler “plugin” (compiler-independent):

→ Inserts new instructions (needed to demonstrate program safety).

→ Runs after optimization, but before linking.

→ Untrusted.

Static verifier (described afterward):

→ Checks machine code for program safety.

→ Trusted, but simple.

# Basic Implementation: Registers

Special/reserved registers (prevent register allocation with `-ffixed-xN`):

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

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

Only reserved registers may be used to access memory.

→ Enforced by the verifier.

# Basic Implementation: Registers

Special/reserved registers (prevent register allocation with `-ffixed-xN`):

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

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

# Basic Implementation: Guard Instruction

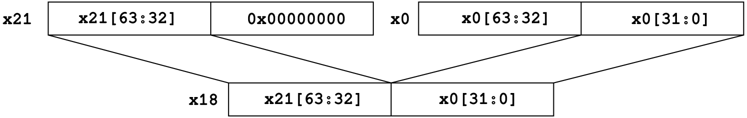
How to safely modify a reserved register?

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

# Basic Implementation: Guard Instruction

How to safely modify a reserved register?

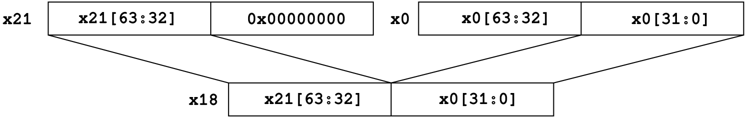
```
mov x18, x0 // unsafe
```



# 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>`br x18` |
| `ldr rt, [xN]` | `add x18, x21, wN, uxtw`<br>`ldr rt, [x18]` |
| `ldr x30, [x18]` | `ldr x30, [x18]`<br>`add x30, x21, w30, uxtw` |

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

→ No bundle alignment or control-flow integrity 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 to multi-instruction sequences.

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 to multi-instruction sequences.

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`<br>`ldr rt, [x21, w24, uxtw]` | 1 |
| `ldr rt, [xN, #i]!` | `add xN, xN, #i`<br>`ldr rt, [x21, wN, uxtw]` | 1 |
| `ldr rt, [xN], #i` | `ldr rt, [x21, wN, uxtw]`<br>`add xN, xN, #i` | 1 |
| `ldr rt, [xN, xM, lsl #i]` | `add w24, wN, wM, lsl #i`<br>`ldr rt, [x21, w24, uxtw]` | 2 |
| `ldr rt, [xN, wM, uxtw #i]` | `add w24, wN, wM, uxtw #i`<br>`ldr rt, [x21, w24, uxtw]` | 2 |
| `ldr rt, [xN, wM, sxtw #i]` | `add w24, wN, wM, sxtw #i`<br>`ldr rt, [x21, w24, uxtw]` | 2 |

# Additional Optimizations

- Guard hoisting: remove redundant guards.
- Stack pointer optimizations.

Can discuss at the end of the talk if there's interest.

# 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`!

# 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`!

```
svc #0
```

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

→ Verifier must ensure `blr` always follows the load.

Benefit: application can select the exact runtime call it wants statically (e.g, fast yield).

Note: additional instructions to save/restore `x30` may be required.

# Implementation Part 1: Assembly Transformation

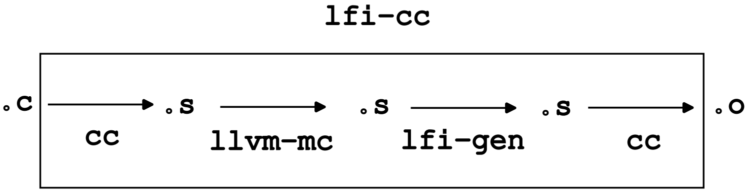
**Problem**: How to implement without modifying/maintaining a compiler toolchain?

# Implementation Part 1: 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 Part 2: Static Verifier

For each instruction, we check the following:

**1**: Must be a legal instruction (no system calls, no unknown/ARMv8.1+ instructions).

**2**: If indirect branch: must target reserved register.

**3**: If memory operation: must target reserved register or use guarded addressing mode.

**4**: If modification to reserved register: must ensure modification maintains invariants:

- `x21` may not be modified.
- `x18`, may only be modified via a guard.
- `x30` may only be modified if immediately followed by a guard, or `blr xN`.
- `sp` may only be modified if immediately followed by a guard, or a stack access (if modification was constant).

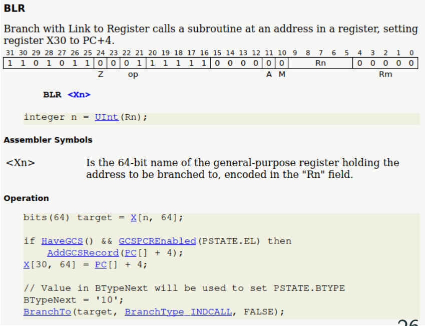
# Implementation Part 2: Static Verifier

Lines of code:

- 290 lines of core logic (manually written).
- 1,600 lines of instruction tables (semi-automatically generated).
- 80,000 lines of disassembler[6] (mostly automatically generated).

Uses the ARM Machine Readable Specification to:

- 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);
```

[6] Provided by Binary Ninja; covers the entire ARMv8.7 ISA.

# Implementation Part 2: Static Verifier

Lines of code:

- 290 lines of core logic (manually written).
- 1,600 lines of instruction tables (semi-automatically generated).
- 80,000 lines of disassembler[6] (mostly automatically generated).

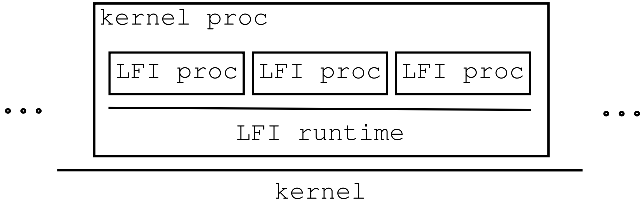
Performance: verifies at 30 MB/s on a Macbook Air.

[6] Provided by Binary Ninja; covers the entire ARMv8.7 ISA.

# Implementation Part 3: Runtime

The runtime is a kernel-like layer between the OS and an LFI process.

→ Enforces safe access to the underlying machine (e.g., file system ops).



Fast switch: switches to a specific sandbox; separate from normal system calls.

→ Clears caller-saved registers instead of saving them.

# Evaluation: Overview

Primary metric: CPU overhead introduced by additional instructions.

Measured on SPEC 2017 benchmarks that compile with our toolchain.

→ C or C++ and compatible with Musl libc.

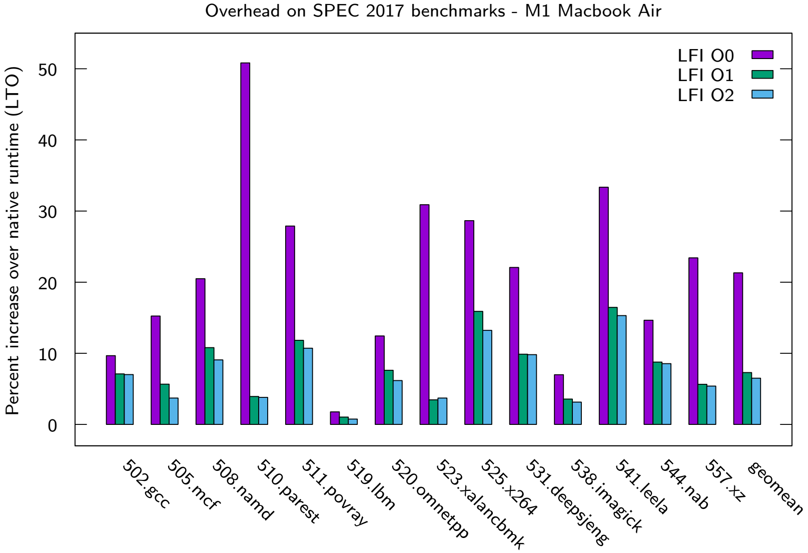
Three LFI optimization levels:

**O0**: No optimizations.

**O1**: Guarded addressing mode enabled.

**O2**: Guard hoisting enabled.

# Evaluation: LFI Overhead



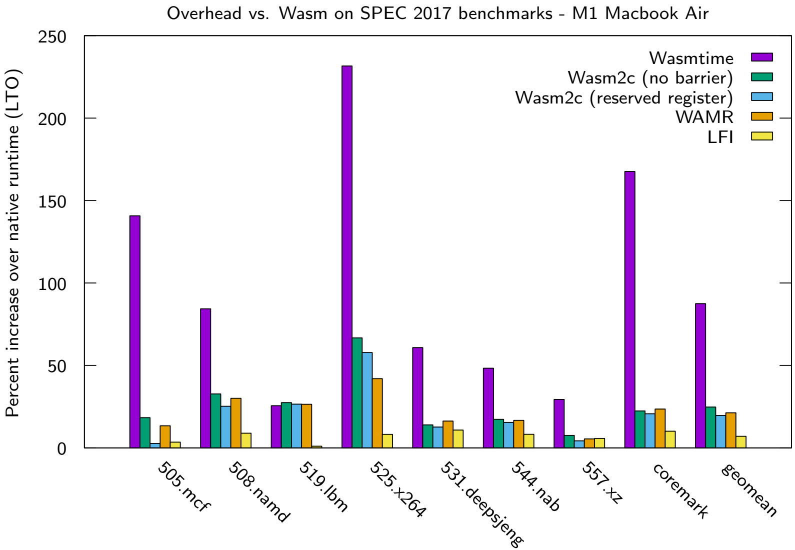
# Evaluation: LFI vs. WebAssembly

WebAssembly engines tested:

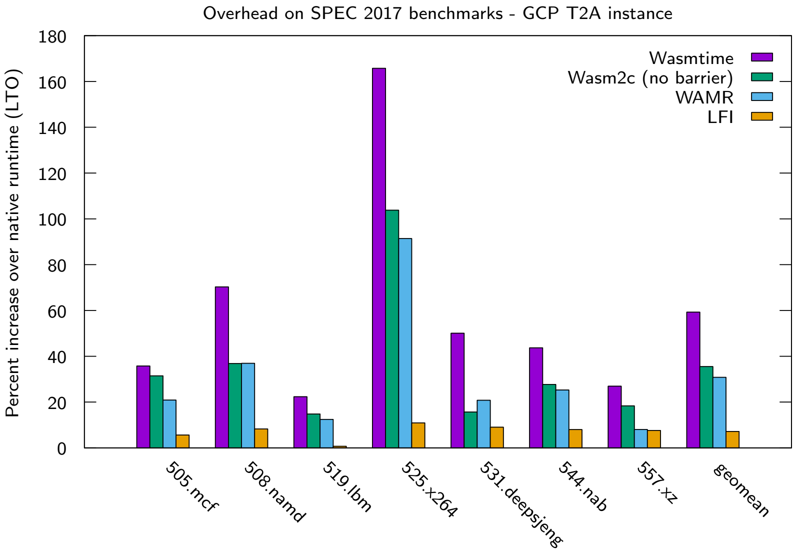
- Wasmtime: WebAssembly JIT compiler using Cranelift.
- Wasm2c: WebAssembly-to-C convertor; C code compiled with Clang.
- WAMR: WebAssembly AOT compiler using LLVM.

Note: restricted to benchmarks that compile with WebAssembly (no exceptions, longjmp, linux-specific headers).

# Evaluation: LFI vs. WebAssembly



# Evaluation: LFI vs. WebAssembly



# Evaluation: Context Switch Microbenchmarks

**Table 1:** GCP T2A VM, 2.8 GHz

| Platform | Syscall (ns) | Ctxsw (ns) |
|---|---|---|
| LFI | 23 | 19 |
| Linux | 162 | 2,227 |
| gVisor | 11,937 | 30,218 |

**Table 2:** Apple M1, 3.2 GHz

| Platform | Syscall (ns) | Ctxsw (ns) |
|---|---|---|
| LFI | 20 | 17 |
| Linux | 128 | 1,214 |

- Linux does not provide an optimized context switch implementation[7].
- gVisor incurs high overhead from the suboptimal Linux switch.
- Software protection can go beyond the limits of current hardware protection.

[7] seL4 does much better with a ~400 cycle switch.

# Discussion: Virtualization

Virtualization could be used as an alternative hardware-based method.

Benefits of virtualization:

- Can run unmodified binaries, including with self-modifying code.
- Provides a large address space for each untrusted program.
- No static verifier.

# Discussion: Virtualization

Hardware virtualization overheads, measured on SPEC 2017 subset:

- Intel VT-x (Core i7 11700k): 3%.
- AMD-V (Ryzen 9 7950X): 15%.
- ARM Virtualization (Cortex-A76): 6%.

Additional problems:

- Nested virtualization incurs higher overheads, or may be entirely unavailable.
- Incurs higher hardware protection switching costs.
- No minimal virtualization-based sandboxing tool currently exists[8] (future work?).

[8] Closest equivalent is the Dune sandbox, as far as I know.

# Spectre Safety

Types of attacks (see Swivel[9] for details):

1. Sandbox breakout: the attacker abuses mispredictions within the sandbox to speculatively access code or data outside the sandbox.
2. Host poisoning: the attacker trains the branch predictor to cause the runtime to execute a Spectre gadget.
3. Cross-sandbox poisoning: the attacker trains the branch predictor to cause another sandbox to execute a Spectre gadget.

[9] Narayan et al., Swivel: Hardening WebAssembly against Spectre. In USENIX Security ’21.

# Spectre Safety

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

→ Speculative sandbox breakout attacks are mitigated.

# 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.

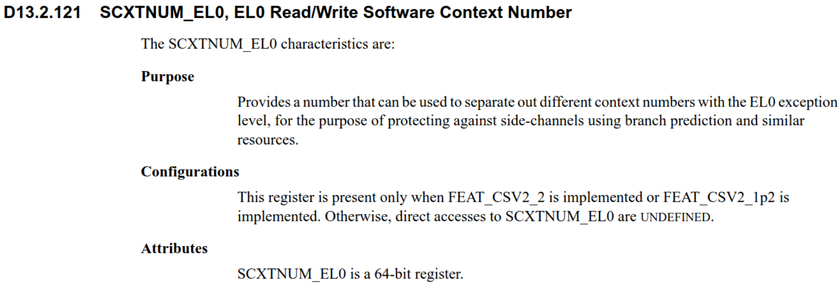
# Spectre Safety

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.

# Side-channel Attacks

An impossible problem to solve in practice?

One aid: remove sources non-determinism — explicit timers and multi-threading (implicit timers).

→ Very few timerless remote side-channel attacks have been published.

Software protection allows greater prevention of issues such as:

1. Hypervisor-based side-channel caused by self-modying code [1].
2. LL/SC timerless side-channel [2].

More investigation needed.

# Thank you!

You can follow further development at:

https://github.com/zyedidia/lfi

Many potential directions:

- Application to other architectures, like x86-64 and RISC-V (extra slides).
- Flexible sandbox sizes: any power of 2, instead of 4GB.
- Determinism: position-oblivious code.
- Portability via dynamic recompilation.
- Lazy verification and hybrid protection techniques.
- Increasing verifier robustness.
- Hardware-based sandboxing using virtualization.
- And more. . .

# 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.

→ 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`<br>`add sp, x21, w24, uxtw` |
| `add sp, sp, #n`<br>. . . (no branches) . . .<br>`ldr rt, [sp, #m]` | No change necessary |
| `str rt, [sp, #n]!` | No change necessary |

# Application to other architectures: x86-64

An efficient implementation is probably possible with Intel CET and segment registers.

CET: shadow call stacks and indirect branch tracking[10].

→ Ensures all indirect branches target instruction boundaries.

→ Verifier will have to check direct branches (slower verification).

Store sandbox base in `%gs`, reserve `%r15`, rewrite loads/stores:

| Original code | Sandboxed equivalent |
|---|---|
| `mov %rxx, (...)` | `lea (...), %r15d`<br>`mov %rxx, %gs:r15` |

[10] Usermode IBT is not currently provided by Linux: showstopper for avoiding alignment constraints.

# Application to other architectures: RISC-V

**Problem 1**: Compressed instructions, and no hardware control-flow protection (yet).

→ Require that compressed instructions only exist as pairs (otherwise decompress).

→ Require that branches target a 4-byte aligned block, possibly via an enforced `and`.

**Problem 2**: More difficult to operate on 32-bit subsets.

→ `Zba` provides `add.uw rd, rs1, rs2` (zero-extends bottom 32 bits of `rs2`).

Store sandbox base in `x21`, reserve `x18`,

| Original code | Sandboxed equivalent |
|---|---|
| `ld xN, n(xM)` | `add.uw x18, x21, xM`<br>`ld xN, n(x18)` |