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

Zachary Yedidia

Stanford University

Untrusted Code

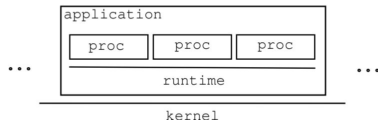
Today's systems increasingly run untrusted code.

- Cloud machines and serverless (VMs, containers, WebAssembly).
- Kernels (eBPF).
- Web browsers (JavaScript, WebAssembly).
- Smart contracts (WebAssembly, EVM).

Applications need lightweight solutions with fast context switch times (single process).

Goal: enforce that untrusted programs

- cannot read/write outside sandbox.
- cannot directly perform system calls.



Software Sandboxing: Language-based Security (LBS)

Native

source language

.c

compile

.elf

run (unsafe)

Software Sandboxing: Language-based Security (LBS)

Native

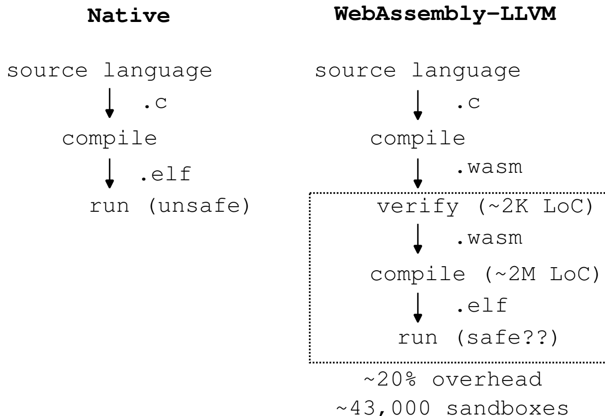
source language
↓ .c
compile
↓ .elf
run (unsafe)

WebAssembly-LLVM

source language
↓ .c
compile
↓ .wasm
verify (~2K LoC)
↓ .wasm
compile (~2M LoC)
↓ .elf
run (safe??)

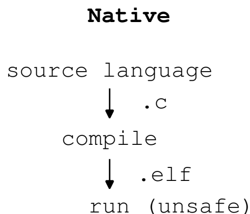
~20% overhead
~43,000 sandboxes

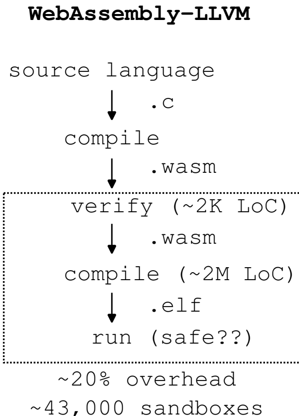
Software Sandboxing: Language-based Security (LBS)

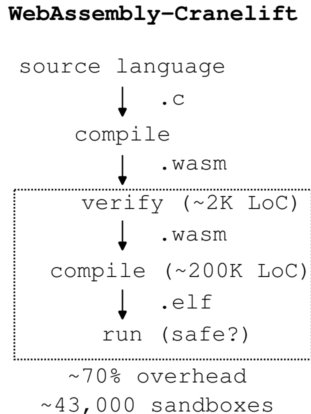


Problem: trusting a language verifier and compiler can be dangerous.

Software Sandboxing: Language-based Security (LBS)







Problem: tradeoff between performance and security.

Are Secure Compilers Secure?

CVE-2021-32629 PUBLISHED　　View JSON

Memory access due to code generation flaw in Cranelift module

Important CVE JSON 5 Information +

Assigner: GitHub M
Published: 2021-05-24 **Updated:** 2021-05-24

Cranelift is an open-source code generator maintained by Bytecode Alliance. It translates a target-independent intermediate representation into executable machine code. There is a bug in 0.73 of the Cranelift x64 backend that can create a scenario that could result in a potential sandbox escape in a Wasm program. This bug was introduced in the new backend on 2020-09-08 and first included in a release on 2020-09-30, but the new backend was not the

Are Secure Compilers Secure?

CVE-2021-32629 PUBLISHED View JSON

Memory access due to code generation flaw in Cranelift module

Important CVE JSON 5 Information +

Assigner: GitHub M
Published: 2021-05-24 **Updated:** 2021-05-24

Cranelift is an open-source code generator maintained
target-independent intermediate representation into exe
in 0.73 of the Cranelift x64 backend that can create a sc
sandbox escape in a Wasm program. This bug was
2020-09-08 and first included in a release on 2020-09-3

CVE-2023-26489 PUBLISHED View JSON

Guest-controlled out-of-bounds read/write on x86_64 in wasmtime

Important CVE JSON 5 Information +

Assigner: GitHub M
Published: 2023-03-08 **Updated:** 2023-03-08

wasmtime is a fast and secure runtime for WebAssembly. In affected versions wasmtime's code generator, Cranelift, has a bug on x86_64 targets where address-mode computation mistakenly would calculate a 35-bit effective address instead of WebAssembly's defined 33-bit effective address. This bug means that, with default codegen settings, a wasm-controlled load/store operation could read/write addresses up to 35 bits away from the base of linear memory. Due to this bug, however, addresses up to `0xffffffff * 8 + 0x7ffffffc = 36507222004 = ~34G` bytes away from the base of linear memory are possible from guest code. This means that the virtual memory 6G away from the base of linear memory up to ~34G away can be read/written

Are Secure Compilers Secure?

CVE-2021-32629 PUBLISHED View JSON

Memory access due to code generation flaw in Cranelift module

Important CVE JSON 5 Information +

Assigner: GitHub M
Published: 2021-05-24 **Updated:** 2021-05-24

Cranelift is an open-source code generator maintained
targe
in 0.7
sandl
2020

CVE-2023-26489 PUBLISHED View JSON

Guest-controlled out-of-bounds read/write on x86_64 in wasmtime

+

y. In affected versions wasmtime's code
ts where address-mode computation
nstead of WebAssembly's defined 33-bit
degen settings, a wasm-controlled load/
s away from the base of linear memory.
3 + 0x7ffffffc = 36507222004 = ~34G`
e from guest code. This means that the
y up to ~34G away can be read/written

CVE-2023-41880 PUBLISHED View JSON

Miscompilation of wasm `i64x2.shr_s` instruction with constant input on x86_64

Important CVE JSON 5 Information +

Assigner: GitHub M
Published: 2023-09-15 **Updated:** 2023-09-15

Wasmtime is a standalone runtime for WebAssembly. Wasmtime versions from 10.0.0 to versions 10.02, 11.0.2, and 12.0.1 contain a miscompilation of the WebAssembly `i64x2.shr_s` instruction on x86_64 platforms when the shift amount is a constant value that is larger than 32. Only x86_64 is affected so all other targets are not affected by this. The miscompilation results in the instruction producing an incorrect result, namely the low 32-bits of the second

Are Secure Compilers Secure?

CVE-2021-32629 PUBLISHED   View JSON

Memory access due to code generation flaw in Cranelift module

Important CVE JSON 5 Information +

Assigner: GitHub M
Published: 2021-05-24 **Updated:** 2021-05-24

Cranelift is an open-source code generator maintained
targe
in 0.7
sandl
2020

CVE-2023-26489 PUBLISHED   View JSON

Guest-controlled out-of-bounds read/write on x86_64 in wasmtime

+

cted versions wasmtime's code
address-mode computation
WebAssembly's defined 33-bit
ttings, a wasm-controlled load/
om the base of linear memory.
ffffc = 36507222004 = ~34G`
uest code. This means that the
-34G away can be read/written

CVE-2023-41880 PUBLISHED   View JSON

Miscompilation of wasm `i64x2.shr_s` instruction with constant input on x86_64

CVE-2021-20320 PUBLISHED   View JSON

Important CVE JSON 5 Information +

Assigner: Redhat
Published: 2022-02-18 **Updated:** 2022-02-18

A flaw was found in s390 eBPF JIT in bpf_jit_insn in arch/s390/net/bpf_jit_comp.c in the Linux kernel. In this flaw, a local attacker with special user privilege can circumvent the verifier and may lead to a confidentiality problem.

Are Secure Compilers Secure?

CVE-2021-32629 PUBLISHED

View JSON

Memory access due to code generation flaw in Cranelift module

Important CVE JSON 5 Information +

Assigner: GitHub M
Published: 2021-05-24 **Updated:** 2021-05-24

Cranelift is an open-source code generator maintained
targe
in 0.7
sand
2020

CVE-2023-26489 PUBLISHED

View JSON

Guest-controlled out-of-bounds read/write on x86_64 in wasmtime

cted versions wasmtime's code
address-mode computation
WebAssembly's defined 33-bit

CVE-2023-41880 PUBLISHED

View JSON

Miscompilation of wasm `i64x2.shr_s` instruction with constant input on x86_64

Important CVE JSON 5 Information +

CVE-2021-20320 PUBLISHED

View JSON

Important CVE JSON 5 Information

Assigner: Redhat
Published: 2022-02-18 **Updated:** 2022-02-18

A flaw was found in s390 eBPF JIT in bpf_jit_insn in arch
kernel. In this flaw, a local attacker with special user priv
may lead to a confidentiality problem.

CVE-2021-3490 PUBLISHED

View JSON

Linux kernel eBPF bitwise ops ALU32 bounds tracking

Important CVE JSON 5 Information +

Assigner: Canonical
Published: 2021-05-11 **Updated:** 2021-09-01

The eBPF ALU32 bounds tracking for bitwise ops (AND, OR and XOR) in the Linux kernel did not properly update 32-bit bounds, which could be turned into out of bounds reads and writes in the Linux kernel and therefore, arbitrary code execution. This issue was fixed via commit 049c4e13714e ("bpf: Fix alu32 const subreg bound tracking on bitwise operations") (v5.13-rc4) and backported to the stable kernels in v5.12.4, v5.11.21, and v5.10.37. The AND/OR issues were introduced by commit 3f50f132d840 ("bpf: Verifier, do explicit ALU32 bounds tracking") (5.7-rc1) and the XOR variant was introduced by 2921c90d4718 ("bpf:Fix a verifier failure with xor") (5.10-rc1).

Are Secure Compilers Secure?

CVE-2021-32629 PUBLISHED View JSON

Memory access due to code generation flaw in Cranelift module

Important CVE JSON 5 Information

Assigner: GitHub M
Published: 2021-05-24 **Updated:** 2021-05-24

Cranelift is an open-source code generator maint
targe
in 0.
sand
2020

CVE-2023-41880 PUBL

Miscompilation of wasm `i64x2.shr_s` instru

CVE-2021-21220 PUBLISHED View JSON

Important CVE JSON 5 Information +

Assigner: Chrome
Published: 2021-04-26 **Updated:** 2023-12-14

Insufficient validation of untrusted input in V8 in Google Chrome prior to 89.0.4389.128 allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page.

CVE-2021-20320 PUBLISHED View JSON

Important CVE JSON 5 Information

Assigner: Redhat
Published: 2022-02-18 **Updated:** 2022-02-18

A flaw was found in s390 eBPF JIT in bpf_jit_insn in arch
kernel. In this flaw, a local attacker with special user priv
may lead to a confidentiality problem.

WebAssembly's defined 33-bit

CVE-2021-3490 PUBLISHED View JSON

Linux kernel eBPF bitwise ops ALU32 bounds tracking

Important CVE JSON 5 Information +

Assigner: Canonical
Published: 2021-05-11 **Updated:** 2021-09-01

The eBPF ALU32 bounds tracking for bitwise ops (AND, OR and XOR) in the Linux kernel did not properly update 32-bit bounds, which could be turned into out of bounds reads and writes in the Linux kernel and therefore, arbitrary code execution. This issue was fixed via commit 049c4e13714e ("bpf: Fix alu32 const subreg bound tracking on bitwise operations") (v5.13-rc4) and backported to the stable kernels in v5.12.4, v5.11.21, and v5.10.37. The AND/OR issues were introduced by commit 3f50f132d840 ("bpf: Verifier, do explicit ALU32 bounds tracking") (5.7-rc1) and the XOR variant was introduced by 2921c90d4718 ("bpf:Fix a verifier failure with xor") (5.10-rc1).

Are Secure Compilers Secure?

CVE-2021-32629 PUBLISHED View JSON

Memory access due to code generation flaw in Cranelift module

Important CVE JSON 5 Information

Assigner: GitHub M
Published: 2021-05-24 **Updated:** 2021-05-24

Cranelift is an open-source code generator maint
targe
in 0.
sand
2020

CVE-2023-41880 PUBL

Miscompilation of wasm `i64x2.shr_s` instru

CVE-2021-21220 PUBLISHED View JSON

Important CVE JSON 5 Information +

Assigner: Chrome
Published: 2021-04-26 **Updated:** 2023-12-14

Insufficient validation of untrusted input in V8 in Google Chrome prior to 89.0.4389.128 allowed a remote attacker to

CVE-2021-20320 PUBLISHED

Important CVE JSON 5 Information

Assigner: Redhat
Published: 2022-02-18 **Updated:** 2022-02-18

A flaw was found in s390 eBPF JIT in bpf_jit_insn in arch kernel. In this flaw, a local attacker with special user priv may lead to a confidentiality problem.

CVE-2021-34

Linux kernel eBPF bitwise o

Important CVE JSON

Assigner: Canonical
Published: 2021-05-11 **U**

The eBPF ALU32 bounds tra properly update 32-bit bou the Linux kernel and ther 049c4e13714e ("bpf: Fix alu and backported to the stab were introduced by commit 3f50f132d840 ("bpf: Verifier, do explicit ALU32 bounds tracking") (5.7-rc1) and the XOR variant was introduced by 2921c90d4718 ("bpf:Fix a verifier failure with xor") (5.10-rc1).

CVE-2020-9802 PUBLISHED View JSON

Important CVE JSON 5 Information +

Assigner: Apple
Published: 2020-06-09 **Updated:** 2020-10-16

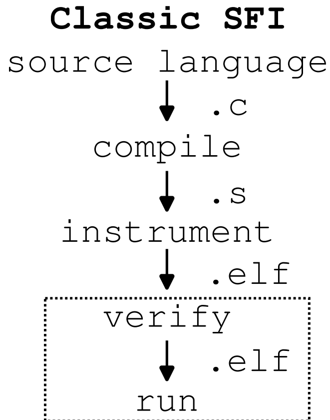
A logic issue was addressed with improved restrictions. This issue is fixed in iOS 13.5 and iPadOS 13.5, tvOS 13.4.5, watchOS 6.2.5, Safari 13.1.1, iTunes 12.10.7 for Windows, iCloud for Windows 11.2, iCloud for Windows 7.19. Processing maliciously crafted web content may lead to arbitrary code execution.

Conclusion: we need better security!

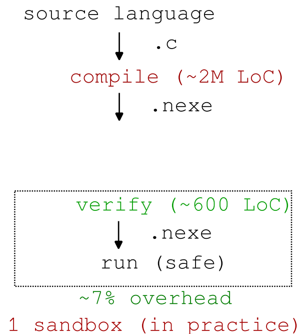
$\rightarrow$ and we can get better performance too.

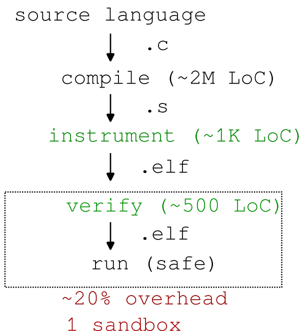
Software Sandboxing: Software Fault Isolation (SFI)

SFI (Wahbe et al., SOSP '93): Verify machine code.

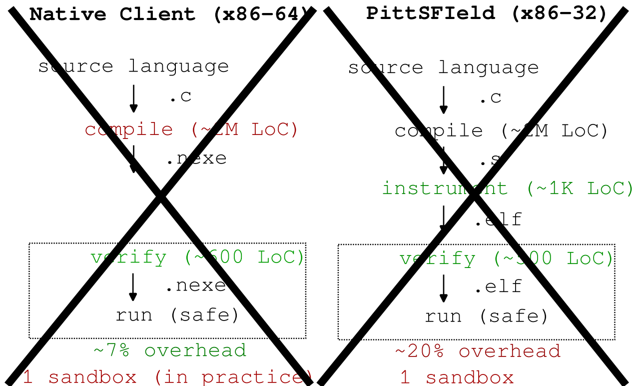


Software Sandboxing: Software Fault Isolation (SFI)

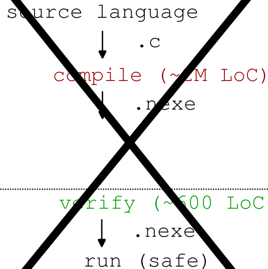


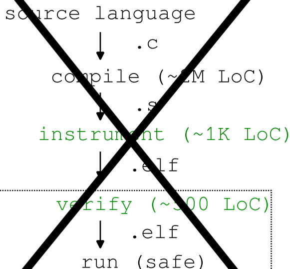


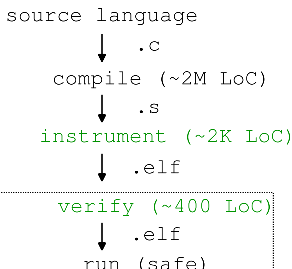
Software Sandboxing: Software Fault Isolation (SFI)



Software Sandboxing: Software Fault Isolation (SFI)

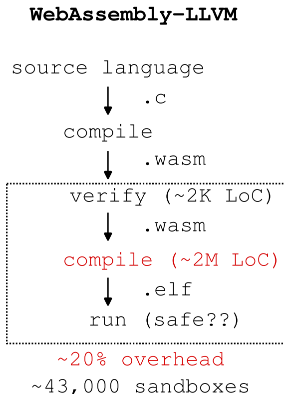


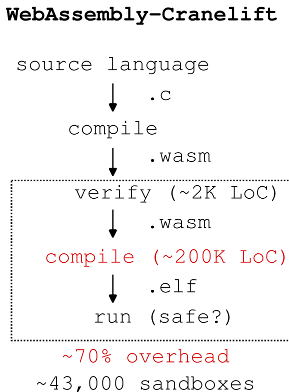


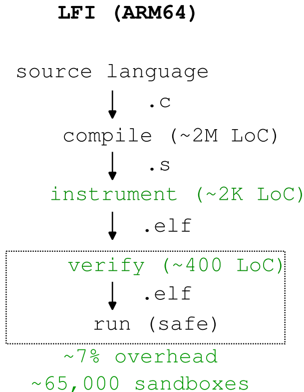


Presenting Lightweight Fault Isolation: **low overhead**, **secure**, **scalable**, **simple**.

Software Sandboxing: Software Fault Isolation (SFI)

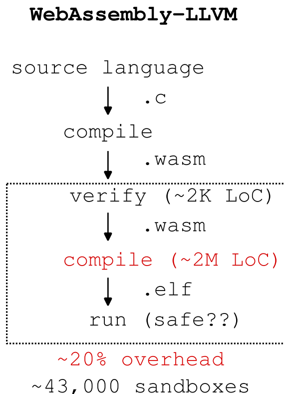


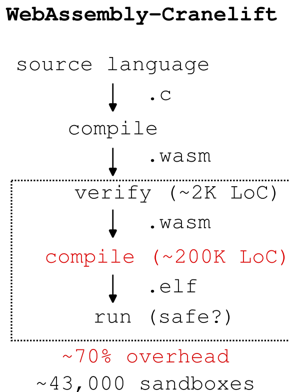


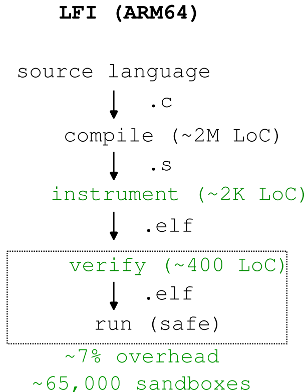


Presenting Lightweight Fault Isolation: **low overhead**, **secure**, **scalable**, **simple**.

Software Sandboxing: Software Fault Isolation (SFI)

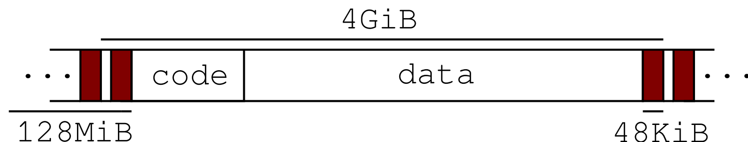






Already using WebAssembly? You can run WebAssembly inside LFI!

LFI Principles



Principles:

- Use 4GiB sandboxes combined with instructions to operate on 32-bit values.
- Works without modification to existing compilers.

  → just reserve a few registers (`x18` and `x21`) when compiling.

  → can work with any compiler that produces GNU assembly.
- Every address in the sandbox is a valid branch target (no aligned bundles).

  → helps simplicity, code size, running time, and Spectre-safety.

Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding: misalignment traps.
  → Consistent disassembly without aligned bundles.
- 32 64-bit registers (`x0`-`x30`, `sp`).
- Stack pointer register (`sp`).
- Dedicated return address register (`x30`).
  → Easy to reserve registers.
- 32-bit register subsets (`w0`-`w30`, `wsp`).
- A 32-bit addressing mode.
  → Fast operations for 4GiB sandboxes.

Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding: misalignment traps.
  $\rightarrow$ Consistent disassembly without aligned bundles.
- 32 64-bit registers (`x0`-`x30`, `sp`).
- Stack pointer register (`sp`).
- Dedicated return address register (`x30`).
  $\rightarrow$ Easy to reserve registers.
- 32-bit register subsets (`w0`-`w30`, `wsp`).
- A 32-bit addressing mode.
  $\rightarrow$ Fast operations for 4GiB sandboxes.

Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding: misalignment traps.
  $\rightarrow$ Consistent disassembly without aligned bundles.
- 32 64-bit registers (`x0`-`x30`, `sp`).
- Stack pointer register (`sp`).
- Dedicated return address register (`x30`).
  $\rightarrow$ Easy to reserve registers.
- 32-bit register subsets (`w0`-`w30`, `wsp`).
- A 32-bit addressing mode.
  $\rightarrow$ Fast operations for 4GiB sandboxes.

Overview of ARM64

Important ARM64 features for SFI:

- Fixed-width encoding: misalignment traps.
  → Consistent disassembly without aligned bundles.
- 32 64-bit registers (`x0`-`x30`, `sp`).
- Stack pointer register (`sp`).
- Dedicated return address register (`x30`).
  → Easy to reserve registers.
- 32-bit register subsets (`w0`-`w30`, `wsp`).
- A 32-bit addressing mode.
  → Fast operations for 4GiB sandboxes.

Basic Implementation: Overview

LFI (ARM64)

source language

↓ .c

compile (~2M LoC)

↓ .s

instrument (~2K LoC)

↓ .elf

verify (~400 LoC)

↓ .elf

run (safe)

~7% overhead

~65,000 sandboxes

Sandbox Instrumentation and Verification

```
add x0, x1, x2    // safe
b foo             // safe
svc #0 (syscall) // unsafe
ldr x1, [x0]      // unsafe
mov x18, x0       // unsafe
```

Unsafe instruction → rejected by verifier.

Sandbox Instrumentation and Verification

```
add x0, x1, x2      // safe
b foo               // safe
svc #0 (syscall)    // unsafe
ldr x1, [x0]        // unsafe
mov x18, x0         // unsafe
```

Unsafe instruction → rejected by verifier.

Special/reserved register (same idea from the '93 SFI project):

- `x18`: always contains a valid sandbox address.

Sandbox Instrumentation and Verification

```
add x0, x1, x2    // safe
b foo             // safe
svc #0 (syscall)  // unsafe
ldr x1, [x0]      // unsafe
mov x18, x0       // unsafe
```

Unsafe instruction → rejected by verifier.

Special/reserved register (same idea from the '93 SFI project):

- `x18`: always contains a valid sandbox address.

```
ldr x1, [x18]  // safe
```

Basic Implementation: Guard Instruction

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

How to safely modify `x18`?

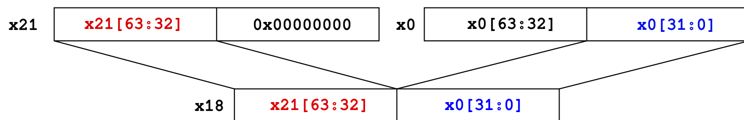
- `x21`: sandbox base address (aligned to 4GiB).

Basic Implementation: Guard Instruction

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

How to safely modify `x18`?

- `x21`: sandbox base address (aligned to 4GiB).

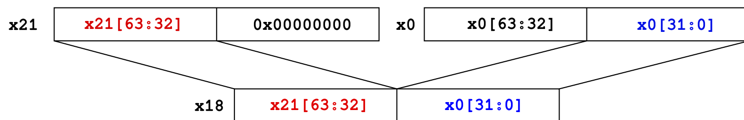


Basic Implementation: Guard Instruction

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

How to safely modify `x18`?

- `x21`: sandbox base address (aligned to 4GiB).



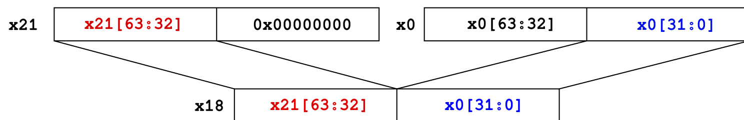

```
add x18, x21, w0, uxtw // safe
```

Basic Implementation: Guard Instruction

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

How to safely modify x18?

- x21: sandbox base address (aligned to 4GiB).




```
add x18, x21, w0, uxtw // safe
ldr x1, [x18]          // safe
```

If x0 contained a valid address, the add is a mov.

Otherwise, sandbox has non-escaping undefined behavior (pointer overflow).

Sandbox Instrumentation

Original	Sandboxed
`ldr x1, [x0]`	`add x18, x21, w0, uxtw` `ldr x1, [x18]`
`br x0`	`add x18, x21, w0, uxtw` `br x18`

Instrumenter performs transformations; verifier is convinced of their safety.

Sandbox Instrumentation

Original	Sandboxed
`ldr x1, [x0]`	`add x18, x21, w0, uxtw` `ldr x1, [x18]`
`br x0`	`add x18, x21, w0, uxtw` `br x18`

Instrumenter performs transformations; verifier is convinced of their safety.

Same invariant for `sp` and `x30`.

```
ret x30           // safe
ldr x1, [sp, #8]  // safe
```

`ldr x30, [sp]`	`ldr x30, [sp]` `add x30, x21, w30, uxtw`

Zero-instruction Guard

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

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

Zero-instruction Guard

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

Original code	Sandboxed equivalent	Cycles of overhead
`ldr rt, [xN]`	`ldr rt, [x21, wN, uxtw]`	0
`ldr rt, [xN, #i]`	`add w22, wN, #i` `ldr rt, [x21, w22, 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

(other addressing modes omitted for brevity)

Evaluation: Overview

Primary metric: CPU overhead introduced by additional instructions.

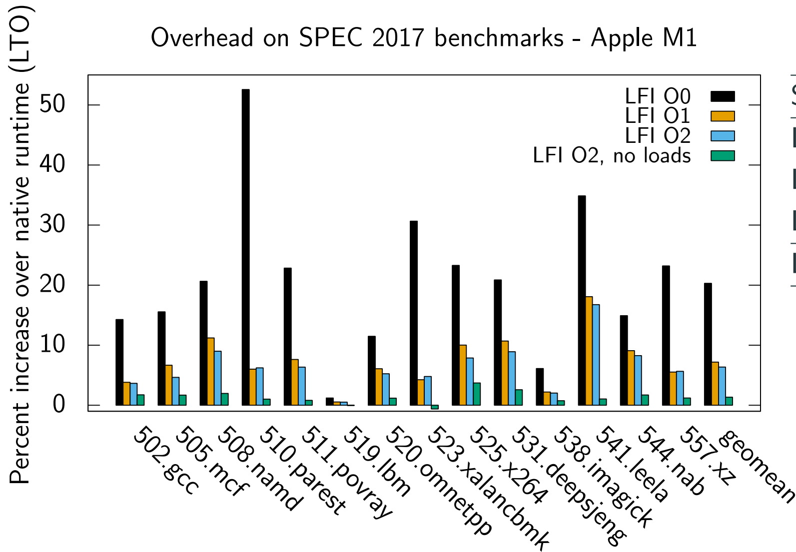
Measured on SPEC 2017 benchmarks that compile with our toolchain.

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

→ Apple M1 running Asahi Linux.

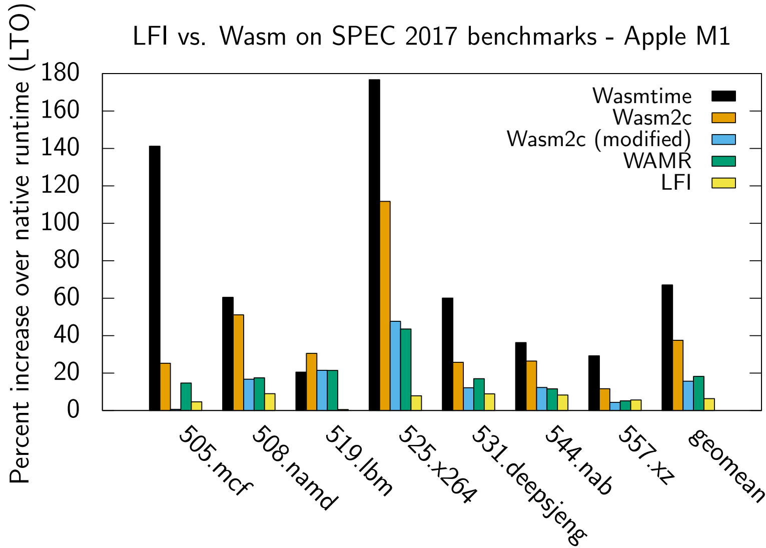
1. Comparing the effects of LFI optimizations.
2. Comparing LFI vs. AOT WebAssembly compilers that use LLVM and Cranelift.
   - → WAMR: LLVM.
   - → Wasm2c: LLVM.
   - → Wasmtime: Cranelift.

Evaluation: LFI Overhead



System	Geomean
LFI O0	20.3%
LFI O1	7.2%
LFI O2	6.4%
LFI, no loads	1.3%

Evaluation: LFI vs. WebAssembly



System	Geomean
Wasmtime	67.1%
Wasm2c[a]	37.5%
Wasm2c*[b]	15.7%
WAMR*	18.2%
LFI	6.4%

*May optimize out loads.

[a] unmodified
[b] with my modifications

Evaluation: Context Switch Microbenchmarks

Table 1: GCP T2A VM, 2.8 GHz

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

Table 2: Apple M1, 3.2 GHz

Platform	Syscall (ns)	Ctxsw (ns)
LFI	22	48
Linux	129	1,504
gVisor	not	supported

- Linux does not provide an optimized context switch implementation[1].
- gVisor incurs high overhead from the suboptimal Linux switch.
- Software protection can go beyond the limits of current hardware protection.

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

Thank you!

You can follow further development at:

https://github.com/zyedidia/lfi

See the paper for more details about optimizations, the verifier, the runtime, Spectre, proposed designs for other architectures (x86-64 and RISC-V), and more!

Questions?

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 w22, wsp, #n` `add sp, x21, w22, uxtw`

`add sp, sp, #n`
. . . (no branches) . . .
`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[2].

→ 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` `mov %rxx, %gs:r15`

[2]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` `ld xN, n(x18)`

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.

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: 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: 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: 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.