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

Zachary Yedidia
Stanford University

## Abstract

Software-based fault isolation (SFI) is a longstanding technique that allows isolation of one or more processes from each other with minimal or no use of hardware protection mechanisms. The demand for SFI systems has been increasing due to the advent of cloud and serverless computing, which require systems to run untrusted code with low latency and low context switch times. SFI systems must optimize for a combination of performance, trusted code base (TCB) size, scalability, and implementation complexity. With the rise of ARM64 in both cloud and personal computers, we revisit classic SFI in the context of ARM64 and present a new multi-sandbox SFI scheme that is practical to implement, efficient, and maintains a small TCB. Our technique, called Lightweight Fault Isolation (LFI), supports tens of thousands of 4GiB sandboxes in a single address space and does full software isolation of loads, stores, and jumps with a runtime overhead of 7% on the compatible subset of the SPEC 2017 benchmark suite. In addition to providing low runtime and code size overheads compared to existing multi-sandbox systems, LFI is implemented independently of existing compiler toolchains, has a small static verifier to reduce TCB size, is hardened against basic Spectre attacks, and has broad software support, including for language mechanisms like exceptions and ISA features such as SIMD.

**ACM Reference Format:**
Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640408

## 1 Introduction

Process isolation is a fundamental part of computer systems security. Most systems make use of hardware protection for isolation. While hardware protection is flexible and can be used to isolate arbitrary code, changing protection domains is expensive. For example, a context switch between two Linux processes on modern high-performance processors can cost thousands of cycles — orders of magnitude more than the time needed to save/restore register state. Even optimized implementations like those in microkernels cost hundreds of cycles.

These costs can be significant in cloud settings, which must run thousands of short-lived untrusted programs with low latency [37, 54], and in settings such as microkernels with frequent IPC between isolated processes [22].

In addition, hardware protection is often unavailable to userspace because it is reserved for the kernel. Instead, user applications typically use one of three possible approaches for sandboxing: virtualization, containerization, or software sandboxing. Software sandboxing systems use either classic software-based fault isolation (SFI) [58, 61], or language-based isolation [20, 30]. While virtualization and containerization incur minimal CPU overhead, the cost of context switches and system calls are high. Conversely, software sandboxing allows multiple isolation domains within the same address space, so context switches are very fast, but existing multi-sandbox software systems incur overheads of 20% or more on typical CPU-bound benchmarks [15, 40, 61]. As a result, some SFI systems only sandbox stores and jumps — not loads — for more reasonable overheads of 5-10% [29, 40, 61], or use hybrid hardware-software mechanisms [17, 29, 59, 66]. Other systems, such as Google Native Client (NaCl) [52], achieve low overheads but significantly restrict scalability (number of sandboxes) to do so.

This work presents Lightweight Fault Isolation (LFI): the first SFI system that supports tens of thousands of sandboxes in a single address space (around 65,000 by default) while also providing full software-based isolation of loads, stores, and jumps with a low overhead of 7%, as measured on a subset of SPEC 2017. LFI combines fast context switches with low CPU overheads that are comparable with hardware-assisted virtualization. LFI also benefits from the use of a machine code verifier, which improves the system's security by reducing the size of the trusted code base (TCB).

Past difficulties with SFI have led to an industry move away from machine code verification approaches. Across various projects, these difficulties with classic SFI included the need to modify compiler backends and assemblers, the irregularity of CISC ISAs like x86, and the lack of complete and up-to-date machine-readable ISA specifications.

*ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0385-0/24/04.
https://doi.org/10.1145/3620665.3640408

Instead, language-based approaches are now typically used instead of SFI. One example is WebAssembly [20]. While WebAssembly was initially designed for sandboxing in web browsers, it also powers low-latency edge computing platforms from Fastly [23], and Cloudflare [60], and is the industry standard for software sandboxing. In a language-based sandbox, an untrusted program is accepted in the form of a safe language. The language's type checker validates the program, and then a compiler transforms it into machine code. If the type checker and compiler are sound then the machine code is safe to run.

However, since compilers are complex and difficult to implement correctly, this approach must grapple with a tradeoff between performance and security. High-performance implementations use LLVM, which is not designed to be secure against malicious inputs: it is over 2 million lines of code, with hundreds of known miscompilation bugs, and uses quadratic-time algorithms that can result in slow compilation (problematic for a secure JIT compiler). Instead, a smaller compiler backend such as Cranelift can be used — it is 200,000 lines of code, uses linear-time algorithms, and has no currently known security-critical miscompilation bugs. However, Cranelift generates code that is significantly slower than LLVM-generated code.

In constrast, by verifying the machine code after compilation in a single linear pass, SFI can provide high performance while maintaining a TCB that is smaller than those of language-based approaches.

Recent changes in the software industry make classic SFI more practical. RISC architectures, such as ARM and RISC-V, have been increasing in popularity and are more amenable to classic SFI. ARM64 in particular has recently started to see wide adoption, with Apple switching all desktop products to use ARM64, and Amazon deploying Graviton 3 widely, providing competitive performance to x86 at a cheaper cost [32]. Increasingly large virtual address spaces [4, 26] also make it practical to support many sandboxes in a single address space.

With these changes in mind, we have developed LFI, the first SFI scheme for the ARM64 architecture, and leverage several architecture-specific optimizations to achieve average runtime and code size overheads of 6% and 13% respectively on SPEC 2017 benchmarks [13]. Our results demonstrate a significant improvement over WebAssembly, the primary available software sandboxing approach with similar capabilities. The scheme supports up to $2^{16}$ sandboxes in the 48-bit usermode address space and performs full isolation via software guards, allowing for very fast context switches between isolation domains. We implement LFI without modifying a compiler toolchain by operating on assembly text produced by off-the-shelf compilers. This keeps our implementation simple and independent from massive software projects.

## 2 ARM64 Overview

The ARM64 ISA [5], also called AArch64, is a 64-bit ISA designed over the last two decades and released as part of ARMv8. Recently, it has begun to reach mainstream adoption: all Apple products from mobile phones to high-end desktops now use ARM processors, and all major cloud providers support ARM instances that are often cheaper than their x86 counterparts and competitively performant. In this section we provide a brief introduction to ARM64 and discuss details about the ISA that are important to the SFI scheme we use. We only discuss the base ARMv8.0-A AArch64 ISA.

Instructions in ARM64 are all 4 bytes — there is no compressed encoding. Each instruction performs operations on registers. There are 31 general-purpose 64-bit registers (numbered `x0` to `x30`), a zero register (`xzr`), and a stack pointer (`sp`). ARM64 is a load-store architecture, meaning that instructions either perform ALU operations between registers, or move values from registers to memory or vice versa. ALU operations may operate on the 64-bit registers, or the bottom 32 bits of each corresponding 64-bit register (referred to as `w0`-`w30`, `wzr`, `wsp`). When writing to a register with the 32-bit name, the bottom 32 bits are written and the top 32 bits are zeroed.

Memory operations in ARM64 center around the `ldr` (load into register) and `str` (store from register) instructions. Several other load/store instructions exist, for example for loading/storing pairs of registers, or for implementing atomic operations. The memory instructions are of special importance to SFI, because these are instructions that could access data outside of a sandbox. Loads and stores may use several different addressing modes, used to determine the address at which to load or store. These addressing modes are listed in Table 1. The more complex addressing modes are only available to basic load/store instructions. It is important to note that immediates in addressing modes are encoded with no more than 15 bits, meaning that such immediates cannot exceed $2^{15}$ bytes (32KiB).

The base ARM64 ISA also supports floating point and SIMD instructions. Floating point and SIMD instructions share a register file of 32 128-bit registers (numbered `v0` to `v31`). There are SIMD load/store instructions, but they use the standard addressing modes and integer registers for address generation. Vector scatter/gather instructions, which only exist in ARM64's SVE extension, are not supported by LFI and are disallowed by the static verifier.

ARM64 has three indirect branch instructions, listed in Table 2. These instructions are also important for SFI since an indirect branch could be used to begin executing code outside a sandbox. Together with loads/stores, these instructions comprise all instructions that could be used to escape from a sandbox. Due to encoding, direct branches can only target locations that are at most 128MiB away.

| 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` |

**Table 1.** Addressing modes for basic load/store instructions. Specialized load/store instructions only support subsets of these addressing modes. Due to encoding, immediates cannot exceed 32KiB in size.

| Instruction | Effect |
|---|---|
| `br xN` | `PC = xN` |
| `blr xN` | `x30 = PC + 4; PC = xN` |
| `ret xN` | `PC = xN` |

**Table 2.** Indirect branch instructions.

## 3 Basic SFI Scheme

A fundamental part of an SFI system is the "guard" instruction, which guarantees that addresses are within the sandbox. One formulation of the ARM64 add instruction can be used to construct a guard:

```
add xA, xB, wC, uxtw
```

The uxtw modifier specifies that the wC 32-bit register should be zero-extended to 64 bits before it is added to xB, and the result is then stored in xA. If xB's bottom 32 bits are all zeroes, this has the effect of combining xB's top 32 bits with xC's bottom 32 bits, while ignoring the existing top 32 bits in xC. This is particularly important for SFI, since it allows us to take a value in xC and directly replace its top 32 bits with a constant stored in the top 32 bits of xB.

We can store each sandbox in an aligned 4GiB region, and put the base address of the region in xB. Since $2^{32}$ is 4GiB and the base address is aligned to 4GiB, xB will contain all zeroes in its bottom 32 bits. Then, for some address stored in xC, no matter what it contains, the add instruction from above will force xC's top 32 bits to be equal to the top 32 bits of the base address — forcing it to be an address within the bounds of the 4GiB sandbox. The resulting "fixed" address will be stored in xA. If xC already contained an address within the sandbox, the instruction will just perform a move from xC to xA, but if not the instruction will generate an in-bounds address in xA. This forms the basic guard instruction for our SFI implementation because it always produces a valid address.

Next, we make use of reserved registers for sandbox integrity. LFI reserves five general-purpose registers for the system:

- x21: contains the base address of the sandbox region.
- x18: always contains a valid sandbox address.
- x22: always contains a 32-bit value.
- x23: always contains a valid sandbox address.
- x24: always contains a valid sandbox address.

Only the first two of these registers must be reserved to implement the sandbox. The remaining three are used for optimizations because we found the impact of reserving up to five registers to be minimal. The particular choices of reserved registers are not especially important, but we try to reserve roughly equal numbers of callee- and caller-saved registers.

Some special registers also have invariants:

- x30 (return address/link register): always contains a valid jump target within the sandbox.
- sp: always contains a valid address within the sandbox.

These reserved registers allow us to perform safe memory accesses. For example, since x18 is guaranteed to always contain a valid address within the sandbox, loading from the address it contains is always safe.

The add guard instruction can then be used in tandem with a reserved register to transform an arbitrary unsafe instruction (memory access or indirect branch) into a safe one. For example, a load such as `ldr x0, [x1]` becomes:

```
add x18, x21, w1, uxtw
ldr x0, [x18]
```

The first instruction performs a safe move from x1 into x18, preserving the invariant that x18 always contains a valid address no matter what x1 contains. Since x18 is a reserved register that must always contain an address that is within the sandbox bounds, it is safe to load from its value. It is safe to execute both instructions separately (a direct jump to the load would be safe), but the combination of the two preserves the semantics of the original `ldr x0, [x1]`.

We treat the sandbox as one 4GiB region without using SFI to isolate code and data segments and instead use hardware protection to prevent writes to the program code and execution of program data, following the same approach as NaCl [66]. The text segment is marked as read/execute-only, and other segments are marked as non-executable. Critically, these hardware protections only need to be modified when the system is initialized. During execution, we can safely context switch between processes or the runtime without modifying any page protections. Sandboxes are placed immediately next to each other to maximize the number of sandboxes that can be supported in a single virtual address space. In order to allow addressing modes that use small immediates, we also reserve 80KiB[1] guard regions (unmapped pages) at the beginning and end of the sandbox. This means the address generated by `[sp, #i]` is guaranteed to be safe

[1]This size is the smallest multiple of 16KiB — the page size on Apple ARM64 machines — that is greater than $2^{16} + 2^{10}$, allowing a stack pointer optimization discussed in Section 4.2.

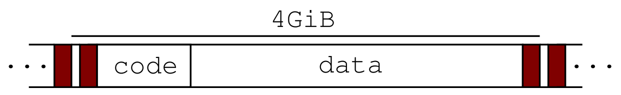


**Figure 1.** The layout of an LFI sandbox (not to scale). The base address must be aligned to a 4GiB boundary. Guard regions of size 80KiB are shown in red. The code must be at least 128MiB away from any executable region in neighboring sandboxes. One page is reserved between guard regions for the runtime call table, discussed in Section 4.4.

since it will either point inside the sandbox or inside a neighboring sandbox's guard region (`#i` is limited to 16 bits[2] by the encoding, and `sp` is guaranteed to be within the sandbox). To prevent jumps from a sandbox into its neighbor, a sandbox's code region must be at least 128MiB away from any executable regions in its neighbors. This is satisfied simply by not placing executable code in the last 128MiB of a sandbox. The sandbox layout is shown in Figure 1.

The total number of sandboxes supported by the system depends on the size of the virtual address space. Typical ARM64 machines support a 49-bit virtual address space, with half reserved for the kernel, and half given to userspace. In the 48-bit userspace region, LFI can support up to 64Ki sandboxes[3]. With access to the kernel address space, either via virtualization or via bare-metal control, LFI can support up to 128Ki sandboxes.

When a sandbox is initialized, a static verifier reads the text segment and ensures that the program follows all invariants necessary to maintain isolation, such as the reserved register invariants. For example, it will reject any instruction that writes to `x18` without using the invariant-preserving guard discussed earlier. The static verifier is discussed in more detail in Section 5.2.

## 4 Optimizations

The `add` guard discussed in the previous section is an effective instruction for SFI, but it executes with 2-cycle latency and half-throughput on both Apple and Arm CPU designs. Our goal for optimizing the SFI scheme is thus to decrease the guard overhead down to 1- or 0-cycle latency. We refer to microarchitectural documentation [8] and analysis [27] for instruction latency and throughput information.

### 4.1 Zero-instruction Guards

In the basic SFI scheme, load/store instructions must use a reserved register that is first loaded via the guard instruction. However, for typical load/store instructions we can take advantage of ARM64's addressing modes to perform the guard directly in the memory access instruction. In particular, the following load is guaranteed to be safe to execute, thanks to the 32-bit addressing mode:

```
ldr rt, [x21, wN, uxtw]
```

This performs the same operation as a load of `xN` in an unsandboxed program. The guard becomes embedded in the memory operation itself, meaning that basic loads/stores (without an offset) can be transformed into safe equivalents at no cost: microarchitectural documentation shows that both forms have equivalent performance.

In addition, sandboxing the more complex addressing modes can now be done more efficiently than with the original two-cycle guard. For example, while the `[xN, #i]` addressing mode cannot be transformed into a safe load at no cost, the use of the 32-bit addressing mode reduces the latency of its sandboxed equivalent by one cycle. All transformations are shown in Table 3.

| Original code | Sandboxed equivalent |
|---|---|
| `ldr rt, [xN]` | `ldr rt, [x21, wN, uxtw]` |
| `ldr rt, [xN, #i]` | `add w22, wN, #i`<br>`ldr rt, [x21, w22, uxtw]` |
| `ldr rt, [xN, #i]!` | `add xN, xN, #i`<br>`ldr rt, [x21, wN, uxtw]` |
| `ldr rt, [xN], #i` | `ldr rt, [x21, wN, uxtw]`<br>`add xN, xN, #i` |
| `ldr rt, [xN, xM, lsl #i]` | `add w22, wN, wM, lsl #i`<br>`ldr rt, [x21, w22, uxtw]` |
| `ldr rt, [xN, wM, uxtw #i]` | `add w22, wN, wM, uxtw #i`<br>`ldr rt, [x21, w22, uxtw]` |
| `ldr rt, [xN, wM, sxtw #i]` | `add w22, wN, wM, sxtw #i`<br>`ldr rt, [x21, w22, uxtw]` |

**Table 3.** Transformations for load operations using the guarded addressing mode, which allows us to optimize guard overhead down to 1 cycle or less. Most of these transformations each only impose 1 cycle of additional overhead, and the base transformation imposes 0 cycles of additional overhead. The same transformations can be re-used for store instructions.

Some specialized load/store instructions do not have access to the guard-form addressing mode. Examples include multi-register operations (`ldp`/`stp`), and atomic instructions. These instructions must be guarded using the basic technique via a reserved register and the `add` guard.

### 4.2 Stack Pointer and Return Address Isolation

Like most SFI systems, we make optimizations in order to efficiently handle stack accesses and function returns. Both the stack pointer and the link register (containing the return address) are assumed to always contain pointers, and rather than introduce guards when those pointers are dereferenced, we perform the guards when the registers are modified.

Guards must only be inserted when the link register is loaded from memory (often at the end of a function). As a

[2]Memory accesses targeting 128-bit SIMD registers allow a 16-bit offset. Earlier versions of this paper used 48KiB guard pages rather than allowing such accesses.
[3]One sandbox region may need to be dedicated to the runtime.

result, functions that do not save/restore the link register do not need any additional guards.

On ARM64, the stack pointer is a special register and cannot be used in all the same cases as a general-purpose register. In particular, it cannot be used as an operand in the zero-extending add instruction that we are using as a guard. This means that we have to expand our stack pointer guard into a two-instruction sequence:

```
mov w22, wsp
add sp, x21, x22
```

This sequence uses the invariant that x22 must always have 32 zeroes in its top bits. As a result, we can use a normal add instruction as the guard, which saves one cycle. This is a specific case of a "pre-extension" optimization that we also implemented more generally: it attempts to avoid the need for zero-extending add instructions by performing the zero-extension as part of an earlier instruction, but after benchmarking we found little benefit in general.

Since the stack pointer always contains a valid address, we can completely drop guards for basic register-immediate addressing modes. The more complex register-register addressing modes still require transformation into basic equivalents, but uses of such addressing modes with the stack pointer are rare. In general, the stack pointer must always have guards inserted when it is modified, but we do apply optimizations that allow us to drop these guards sometimes, as discussed below.

***Pre/post-indexed modification.*** When the stack pointer is modified as part of a load or store instruction via the post/pre-index addressing mode, a guard is not necessary as the stack pointer can be known to be valid. In the pre-index case, the memory access would cause a trap if sp exited the bounds of the sandbox into a guard page, so after the memory access completes, we are certain sp contains a pointer within the sandbox. In the post-index case, for the same reason the stack pointer was guaranteed to be within the sandbox before the modification took place (in the post-index case, sp is modified after the memory access). After the modification, sp must be at most $2^{10}$ bytes away from the sandbox (pre/post-index immediates at most 10 bits), which could not be past a guard region. If sp is subsequently modified by an arithmetic instruction, a guard will be introduced. Otherwise, if sp is subsequently accessed it will be within the sandbox or a guard page. A second pre-index mode can only move the stack pointer by another $2^{10}$ bytes — still well within the guard region — before performing an access that will cause a trap. A normal immediate mode can index at most $2^{15}$ bytes away, and $2^{15} + 2^{10}$ is also still within the guard region.

***Later access within the same basic block.*** If the stack pointer is modified by an add/sub-immediate instruction with a sufficiently small immediate (less than $2^{10}$), but is later accessed by a memory operation in the same basic block (i.e., without a branch in between), the guard can be safely omitted. This is a classic SFI optimization that is safe since the immediate is limited to a size smaller than that of the guard page. The stack pointer may move outside the sandbox into a guard page, but after the modification the machine is guaranteed to execute the memory operation that accesses sp, causing a trap if it moved into a guard page.

```
str x0, [x1, #8]
str x0, [x1, #16]
str x0, [x1, #24]
str x0, [x1, #32]
```

**(a)** Before LFI

```
add x24, x21, w1, uxtw
str x0, [x24, #8]
str x0, [x24, #16]
str x0, [x24, #24]
str x0, [x24, #32]
```

**(b)** After LFI

**Figure 2.** Example demonstrating the effect of redundant guard elimination. Instead of using a guard for each store instruction, a single guard can be used for all four. Since the hoisting register is reserved, the optimization is resistant to jumps that skip the guard.

## 4.3 Redundant Guard Elimination without CFI

Programs often perform several loads/stores in a row, each offset from the same base register. This pattern can be common when accessing array or struct elements. We have an optimization that eliminates these redundant guards by performing an initial guard, storing the result in a reserved "hoisting" register, and subsequently performing all accesses offset from the hoisting register. An example of this optimization is shown in Figure 2. This optimization can help both runtime and code size.

This optimization is inspired by prior work [68], but unlike prior work, our implementation does not rely on the enforcement of control-flow integrity (CFI) via additional mechanisms. Since ARM64 is a register-rich ISA, we are able to apply this optimization without needing strong CFI guarantees by using an additional reserved register. In addition, using a reserved register keeps the verifier simple because it only needs to check that the reserved register is modified using a guard instruction and to allow loads/stores that use the reserved register — it does not need knowledge of the hoisting optimization in order to verify its correctness. We reserve two registers (x23 and x24) for the purpose of redundant guard elimination. The second register makes it possible to hoist two sets of redundant guards in the same basic block.

## 4.4 Runtime Calls without Trampolines

The sandbox must be able to make calls to the runtime so that it can perform operations (such as system calls) on behalf of the sandbox. One solution would be to place the runtime call entrypoint address in a reserved register and allow indirect branches to that register, but this would involve reserving an additional register exclusively for this use. Instead, we take advantage of the fact that the reserved register x21 already points to the base of the sandbox. We can place a table of runtime call addresses in the first page of the sandbox, and mark it as read-only. It is then followed by the guard region and the program code. Loads offset from x21 are permitted to write to register x30, as long as the following instruction is blr x30, which resets x30 to an address within the sandbox. Since this table is located before the guard region, it can be read by the neighboring sandbox, and must therefore not contain any sandbox-specific secrets.

Thus a runtime call looks like the following instruction sequence[4].

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

This also has the benefit that the program can statically load the runtime call function pointer it wants and jump to it directly, without needing to use a register to identify the desired runtime call, and without incurring additional overhead in the runtime to dynamically determine which call was requested. This optimization is used as part of our fast direct yield runtime call, discussed in Section 5.3. Unused entries in the table point to an unmapped page, and the verifier is also used to ensure the value of n is valid.

## 5 Implementation

A classic SFI implementation requires three parts: the compiler that generates programs that pass verification, the static verifier, and the runtime. In this section we discuss the implementation of each of these components.

### 5.1 Compiler

Many prior SFI systems that work with GCC or LLVM have been implemented as modifications to the corresponding compiler toolchain. For example, the latest versions of NaCl are implemented by several thousand lines of modifications to LLVM, and other SFI systems often similarly choose to modify LLVM. By contrast, our system is not fundamentally tied to a particular compiler toolchain, and is implemented independently.

The implementation consumes assembly source code files generated by the compiler, and produces assembly source code files with SFI guards inserted to be then passed to the assembler. We can accept assembly source from LLVM, GCC, or any other compiler toolchain that can produce GNU assembly text. Our compiler tool wraps Clang to automatically output an intermediate .s file that is then transformed by the LFI tool and then fed to the assembler. Clang is invoked with -ffixed-reg flags to prevent it from using reserved registers. To simplify implementation even more, we first pass the assembly file through llvm-mc, a tool that preprocesses assembly files and resolves local labels, assembler macros, assembler expressions, and other features of the GNU assembly format. Our optimizing assembly transformation pass is implemented in roughly 1,500 lines of code, including code for parsing and emission.

There are two components to a compiler toolchain: the compiler itself, and the runtime libraries needed for the generated program to function properly. For example, a hosted C program may be compiled with Clang, and must be able to link with a libc implementation and either libgcc or compiler-rt (i.e., a library that implements compiler intrinsics). While our tool accepts assembly output from any compiler toolchain, the toolchain must still use SFI-instrumented versions of the runtime libraries. We currently provide instrumented versions of the LLVM/musl C/C++ toolchain: musl-libc [44], compiler-rt, libc++, libc++abi, and libunwind [36]. In the future we also hope to provide an LFI-compatible GNU toolchain, as well as toolchains for other languages.

***Difficulties.*** ARM64 contains local branch instructions (tbz/tbnz) that can only jump a relative distance up to 32KiB in either direction. It is possible that in an especially large function, inserting SFI instructions can cause one of these branches to target a destination that is out of range. We solve this by conservatively estimating the distance between local branches and their targets, and replacing the branch with a two-instruction sequence that can target further destinations if the estimate indicates that this is necessary.

Another issue we encountered involved LLVM's "AArch64 compressed jump table" pass [35]. This pass attempts to compress jump table offsets from 32-bit words into shorts or bytes by checking if the offset between the jump base and jump target is small enough to fit into a smaller storage unit. Since we add guards after this pass, the offset may change and the storage unit may become too small to store the offset. We solve this issue by disabling the AArch64 compressed jump table pass.

***Why not implement in LLVM directly?*** Most prior SFI implementations choose to modify LLVM directly. We determined that this was less practical than parsing GNU assembly because it ties the implementation to a particular compiler toolchain, and usually even a particular version of LLVM. Maintenance would be more difficult since LLVM moves quickly and makes major changes across versions, and development would be more complex because the project would depend on a megasystem with millions of lines of code. We have seen several projects become stuck on old versions of LLVM due to the maintenance burden of keeping up-to-date.

In addition, the benefit of modifying LLVM directly is minimal. Most of the guards must use architecture-specific instructions, and it is useful to be able to instrument assembly code files since some projects contain .s files as source code. Such files do not get translated to LLVM IR, or even

[4]Saving and restoring x30 may additionally be necessary. Our current implementation conservatively always saves/restores x30 when inserting this sequence.

LLVM MIR — `.s` files are directly converted to `MCInstr` structures, without even the possibility of using a backend pass. The modifications must therefore be made in the assembler, which is not designed to support passes or store CFG information.

## 5.2 Static Verifier

The static verifier is a small program that reads the text segment of a binary that the user wishes to execute and performs a single linear pass to verify that the machine code is well-behaved. Specifically, it ensures the following properties:

1. Loads, stores, and indirect branches may only target reserved registers (guaranteed to always hold valid sandbox addresses), or use safe addressing modes (where the addressing mode guarantees the resulting address is valid).
2. Reserved registers are only modified in safe ways: `x21` is never modified, and registers such as `x18` and `x30` are only modified using the `add x18, x21, wN, uxtw` instruction, which guarantees that the result stored in `x18` is a safe address.
3. No unsafe instructions are used (e.g, a system call instruction). We only allow instructions from a premade list of safe ARMv8.0 instructions.

The verifier is vital because it keeps the trusted code base small. Unlike most language-based sandboxing systems, the compiler that produced the program does not need to be trusted. This is because all security properties are checked directly on the machine code, after the compiler has finished running. By contrast, WebAssembly implementations that use LLVM assume that the generated code is safe. As a result, they must trust that all LLVM passes used during compilation perform correct transformations. Using a static verifier provides a major reduction in TCB size, especially when the trusted compiler being removed is LLVM. While it is possible to use a static verifier with the Cranelift WebAssembly backend [28], such a verifier is more complex, slower, and directly tied to Cranelift. By contrast, the LFI verifier is fast and can verify binaries at a rate of around 34 MB/s on a Macbook Air (all SPEC 2017 binaries verify in under 0.3 seconds per binary). For additional comparison, the WABT WebAssembly validator, which type-checks WebAssembly bytecode (a required validation step), runs at 3 MB/s on the same machine.

The core of the verifier is implemented in 300 lines of Rust. It additionally makes use of the Binary Ninja ARM64 disassembler [31], an ELF reader, and a list of instruction definitions, which are not counted as part of the core. Despite the small core, the external dependencies are larger than we would like, and we are pursuing improvements to the verifier to reduce its external dependencies.

One major aid to the construction of the static verifier is Arm's Machine Readable Specification (MRS) [9], which provides a complete formal definition of the ARM64 ISA. The MRS is used to auto-generate most of the disassembler, and to generate instruction definitions. These definitions list instructions that can perform indirect branches, modify registers, or access memory — knowing such instructions is of vital importance to the verifier.

## 5.3 Runtime

The runtime is a single process that manages sandboxes as they run and provides mediated access to the underlying host machine. ELF executables are verified and then loaded into appropriate 4GiB slots in the address space. The runtime also marks the first page of the sandbox as read-only and places the runtime call table there, so that the application may safely call the runtime without the need for a trampoline.

The runtime supports a set of basic runtime calls such as `open`, `read`, `write`, `fork`, `wait`, `pipe`, `mmap`, and more. This effectively implements a small Unix-like operating system within a single Linux process. When a sandbox makes a runtime call such as `open`, the runtime first checks the arguments for correctness. For example, the runtime can disallow all access to certain directories. Runtime calls that perform file access will often end up making a system call to Linux, while runtime calls that perform process management, such as `fork`, `wait`, or `yield`, are handled internally. Internal runtime calls can speed up applications significantly since there is no need for a mode switch or pagetable switch to handle them.

The runtime also supports a special optimized yield for performing an inter-process call. This yield implementation directly invokes another sandbox by saving and restoring only the callee-saved registers, and can be used to implement microkernel-like IPC. Since no hardware context switch is required, the inter-process call is extremely fast, costing roughly 50 cycles.

The runtime uses signals for preemption. An alarm signal is repeatedly generated using a timer configured with `setitimer`. When the signal arrives, the runtime uses its scheduler to decide which sandbox to run next.

One useful consequence of performing SFI using guards that force pointers back into the sandbox is that it becomes possible to implement `fork` in a single address space. Since the top 32 bits of all pointers are reset before any memory access, pointers can be constructed as 32-bit offsets within a 4GiB region that could be located anywhere. This can be achieved by zeroing the top 32 bits of addresses constructed with `adr`/`adrp`, read from reserved registers, or returned from the runtime. When a fork occurs, the child is placed at a new base address, which will be added to its pointers when they are accessed. Implementing fork as copy-on-write is also possible through the use of Linux's `memfd` API to map the same memory at multiple places in the address space.

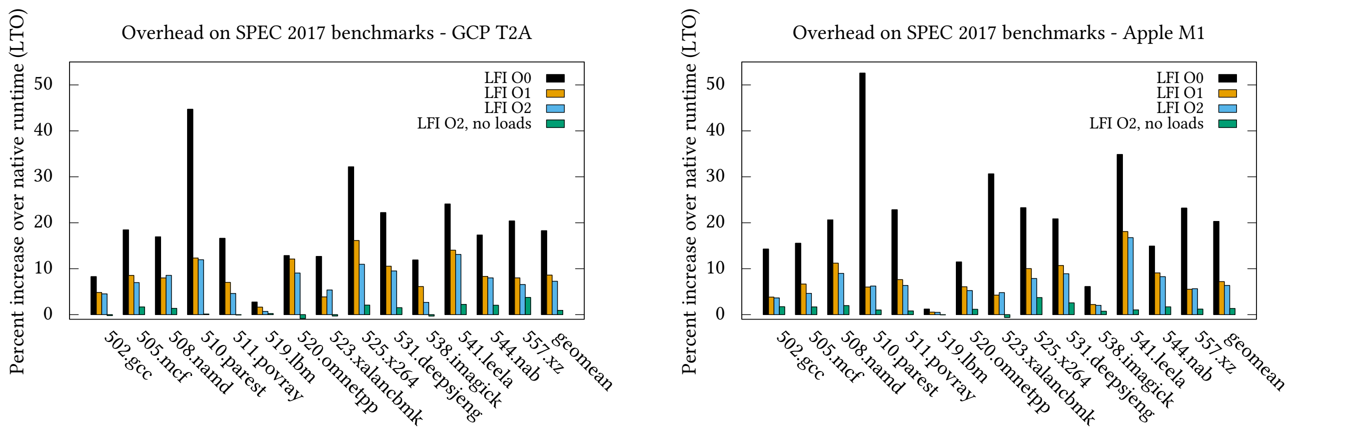


**Figure 3.** The effects of the various optimizations we apply. The data bars are shown in the same order as the legend.

## 6 Evaluation

We evaluate LFI using the SPEC CPU2017 benchmark suite [13]. SPEC 2017 is the most recent iteration of the industry standard SPEC CPU benchmark suite, comprising programs written in C, C++ and Fortran. SPEC 2017 has two categories: SPECspeed, which requires up to 16GB of RAM, and SPECrate, which requires up to 1-2GB per copy, and may run multiple copies of a benchmark in parallel. Since our sandboxes are limited to 4GiB in size, we only use single-copy SPECrate benchmarks, which are typically just the same as SPECspeed programs but with smaller inputs. We also limit the benchmarks to C/C++ programs that compile with musl-libc, resulting in 14 supported benchmarks[5]. We evaluate on two machines: an Apple M1 Macbook Air (3.2 GHz) running Asahi Linux 6.3.0 and LLVM 15.0.7, and a Google Cloud Platform (GCP) T2A instance (3.0 GHz) running Linux 6.1.0 and LLVM 15.0.7. The M1 represents close to the best performance available for ARM64 machines, and the GCP machine represents a virtualized environment on a server-class machine.

Our goal in evaluation is to determine the runtime and code size overhead of the additional SFI checks, and how these overheads compare with WebAssembly (the only other software sandboxing system that supports ARM64) as well as hardware-based isolation. We are also interested in the individual effects of the various optimizations we apply.

### 6.1 Runtime Overhead

We would like to determine how much runtime overhead is introduced by the additional guard instructions and transformations used by LFI. We do this by compiling both with the guards (LFI) and without (native). Since LFI can speed up system calls, we run the native version within the LFI environment so that it also benefits from accelerated system calls. In our testing, this made a difference on the `502.gcc_r` benchmark, where LFI was faster than native code running directly under Linux because the benchmark was frequently allocating and de-allocating large regions of memory with `mmap`, and incurring overhead from frequent system calls. Running native code within the LFI runtime removes this advantage that LFI has over standard native code.

***Optimization effects.*** Since our assembly transformer applies various optimizations, we would like to analyze the effect of each optimization on overall performance. We split the optimizations into multiple levels:

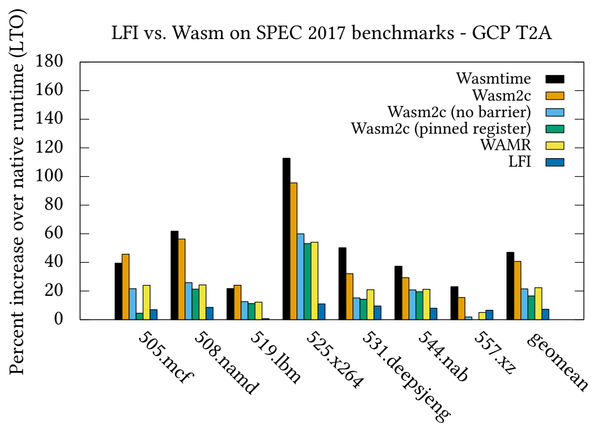
- `O0`: only uses the basic two-cycle guard instruction. All optimizations are disabled except ones that apply to the stack pointer.
- `O1`: enables use of the zero-instruction guard, and address mode rewriting to take advantage of it.
- `O2`: enables redundant guard elimination.
- `O2, no loads`: does not perform sandboxing for loads.

The performance of each optimization level is shown in Figure 3. The jump between `O0` and `O1` is especially large. This shows the importance of optimizing the guard instruction down from two cycles to either one or zero cycles for common load/store instructions. We face diminishing returns beyond this primary optimization. Redundant guard elimination provides about a 1.5% overhead reduction (and the code size reduction is also useful).

With all optimizations enabled (LFI O2) the geometric mean is 6.4% on the M1 and 7.3% on the T2A. On the M1, the worst benchmark is `leela_r`, with a runtime overhead of 17%, and it is the only benchmark with over 10% overhead.

Figure 3 also shows the performance of pure "fault isolation," where loads are not isolated. This allows sandboxes to read, but not modify, each other's data. This form of isolation is only suitable for certain use-cases, such as software compartmentalization, but reduces overhead to around 1%.

[5]`perlbmk` and `blender` rely on Glibc-specific features.



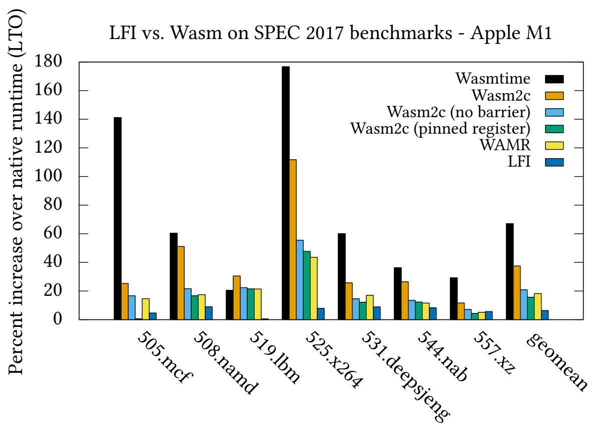


**Figure 4.** Shows the performance overheads of LFI and various WebAssembly engines against native code compiled with LTO. On average, LFI has less than half the overhead of Wasm, and significantly outperforms Wasm on several benchmarks. All systems used ahead-of-time compilation.

## 6.2 Comparison with WebAssembly

Next we compare with several WebAssembly engines by using their WASI support. WebAssembly/WASI has limited feature support: it does not support C++ exceptions, setjmp/-longjmp, or certain system calls needed by some benchmarks. Two benchmarks (`nab` and `x264`) also needed to be slightly modified in order to compile with WebAssembly[6]. Overall, these limitations restrict benchmarking to 7 of the 14 benchmarks. For comparison, we select the two most performant WebAssembly compilers:

- WAMR (also called `iwasm`) [2]: a WebAssembly compiler and engine built on LLVM. It supports an ahead-of-time (AOT) compiler that uses LLVM to transform Wasm directly to machine code, which we use for benchmarking.
- Wasm2c [10]: a compiler that produces C from WebAssembly input. An optimizing C compiler (Clang in this case) can then be used to emit machine code. We use the UVWASI library [50] with Wasm2c to produce native binaries that can make system calls.

We use the most recent versions of WAMR and Wasm2c at the time of writing: 1.2.2 and 1.0.33, respectively.

Other WebAssembly compilers and runtimes exist but either do not perform full sandboxing, or do not outperform WAMR/Wasm2c. In general, we find that Wasm2c and WAMR currently represent the most efficient sandboxing Wasm compilers/runtimes, in agreement with prior work [15]. Notably, no WebAssembly compiler that uses LLVM also supports a native code verifier, so LLVM is free to perform as many optimizations as it can without any constraint that the generated code be verifiable.

[6]These modifications were minimal and involved marking `errno` as thread-local, and fixing two conflicting function signatures.

We also benchmarked Wasmtime [1] 16.0.0, even though it uses Cranelift rather than LLVM, because it is one of the most widely used WebAssembly engines and serves as a baseline. Benchmarking was done with Wasmtime's AOT compiler rather than its JIT compiler.

All WebAssembly engines were given WebAssembly bytecode generated by Clang and further optimized by the `wasm-opt` tool, and we enabled the SIMD128 extension for all benchmarks. All engines were also configured to omit bounds checks and use guard pages for protection instead.

***Optimizations to Wasm2c.*** We made two modifications to Wasm2c to improve its performance. First, Wasm2c's default configuration has a lot of overhead because it attempts to strictly adhere to the WebAssembly specification, which requires that out-of-bounds accesses must cause a trap. As a result, Wasm2c includes a compiler barrier to prevent LLVM from optimizing out loads and stores from WebAssembly memories [63]. Since this barrier can have a significant effect on performance, we removed it. WAMR has chosen a different default: it does not include such a barrier, and has no option to enable one [21]. As a result, both WAMR and our modified version of Wasm2c are non-conforming WebAssembly implementations, and do not pass tests that check for adherence to WebAssembly's deterministic trap semantics.

Second, Wasm2c stores the current module's heap base in a struct, and passes a pointer to this struct to all functions. When a memory access is made, the 32-bit pointer is offset into the heap base, which must be loaded from the struct. This introduces an additional load instruction for every memory access (except when LLVM can perform hoisting). Instead, the heap base could be pinned in a reserved register

| 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% |

**Table 4.** Geometric mean of the overheads over native code (LTO) for various sandboxing systems on the benchmarks from Figure 4.

to avoid these loads, which we implement to increase the performance of Wasm2c[7].

***Results.*** We compare WebAssembly against link-time optimized (LTO) versions of native code and LFI (providing full isolation). This is because ahead-of-time WebAssembly compilation intrinsically has the benefit of LTO by running an optimizing compiler that has a full view of the entire compilation unit. Figure 4 compares WebAssembly overheads to LFI on 7 SPEC 2017 benchmarks. We find that the Wasm runtimes reach geomean overheads of around 15% at best, while LFI has 6-7% overhead on the same set of benchmarks. Geomean overheads are shown in Table 4.

The difference in overhead between LFI and WebAssembly may partially be explained by the increased number of compilation steps for WebAssembly. The compiler first targets the safe Wasm IR, then possibly C, and finally machine code. These additional steps make it more difficult for the compiler to make correct decisions about function inlining and auto-vectorization.

Another factor in play is the additional checks that Wasm performs for indirect function calls. Wasm must ensure that the function being called is valid and has the correct type signature, while LFI just verifies that the target address is within the sandbox.

## 6.3 Code Size Overhead

Code size overhead is generally not as large a concern as runtime overhead, but can be important when transferring binaries over a network or when disk space is scarce.

Since our SFI scheme does not use any alignment constraints, it does not introduce additional padding. The lack of padding, zero-instruction guard, and redundant guard elimination optimizations result in quite modest code size increases. We calculated the percent increase in code size for both the text segment and the overall binary for the SPEC 2017 subset supported by LFI, and observed a geomean text segment increase of 12.9% and overall binary size increase of 8.3%. By contrast, for the subset of benchmarks supported by WebAssembly, we found that WAMR incurs a geomean overall binary size increase of 22%.

[7]Our implementation of reserved heap registers in Wasm2c does not fully handle cross-module indirect calls. This may cause Wasm2c to handle indirect calls faster than it would in a fully conforming implementation.

## 6.4 Comparison with Hardware-based Approaches

An alternative to software sandboxing for usermode isolation is to use hardware-based techniques such as virtualization and containerization. These techniques generally have lower CPU overheads thanks to dedicated hardware support, but suffer from worse context switch performance. Virtualization makes it possible to run guest kernel code in userspace, with another layer of pagetables for isolation between processes running within the guest kernel. This incurs modest CPU overhead, as shown in Figure 5, because the cost of a TLB miss is doubled due to the additional pagetable levels. An additional weakness of virtualization is that it is already used universally by cloud providers, meaning that nested virtualization would be required when running in such an environment. This may cause additional performance overhead, or may be entirely unusable: for example, nested virtualization is not supported on the T2A instance.

Containerization is a different approach that makes use of Linux APIs to allow one process to control and handle the system calls made by another process in a secure way. While there is no CPU overhead for this approach, the interprocess communication between the sandboxed process and the handler suffers from the high context switch costs that exist on Linux.

We have implemented microbenchmarks to compare LFI's context switch performance with hardware-based approaches, shown in Table 5. We compare LFI both with Linux, representing typical pagetable-based hardware protection, and gVisor [18], representing the containerization approach. Our benchmarks with gVisor use its recently released `systrap` platform (August 2023), which is significantly more performant than the older `ptrace` platform. Unfortunately gVisor does not support systems with page sizes other than 4KiB, making it unsupported on Asahi Linux, which uses 16KiB pages, so we can only include it in the T2A results.

The `syscall` benchmark measures the time needed to perform a system call that performs no work (i.e., `getpid`). Since LFI does not require a hardware mode switch to transition from untrusted code into the runtime, the `syscall` benchmark outperforms Linux (standard hardware protection) by a factor of 6. Furthermore, since gVisor must handle system calls via communication between multiple Linux processes, it must perform multiple context switches just to handle a system call.

The `pipe` benchmark aims to measure the overhead of switching between two isolation domains via a system call. The benchmark sets up two pipes between two processes that repeatedly pass one byte of data back and forth. Since the data passed in the pipe is minimal, the majority of time is spent switching between processes.

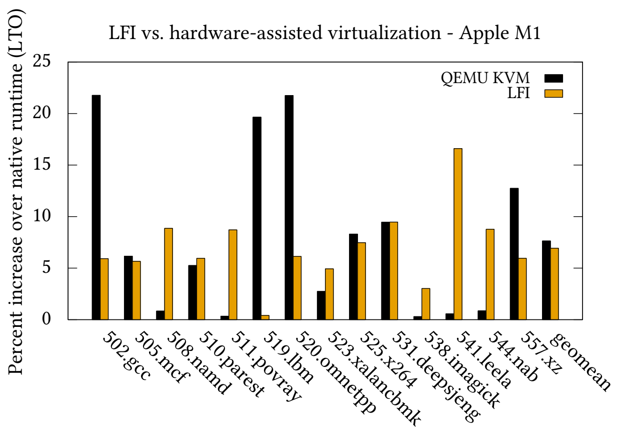


**Figure 5.** Comparison of the overheads of LFI and KVM (using a Ubuntu 20.04 virtual machine) on an Apple M1.

| Benchmark | LFI | Linux |
|---|---|---|
| `syscall` | 22ns | 129ns |
| `pipe` | 46ns | 1504ns |
| `yield` | 17ns | - |

**(a)** Apple M1 (3.2 GHz).

| LFI | Linux | gVisor |
|---|---|---|
| 26ns | 160ns | 12019ns |
| 48ns | 2494ns | 22899ns |
| 18ns | - | - |

**(b)** GCP T2A instance (3.0 GHz).

**Table 5.** Microbenchmarks for measuring the overhead of changing isolation domains.

The `yield` benchmark measures the time to make a cross-sandbox function call, by changing the current isolation domain and directly invoking another sandbox. This is similar to a microkernel-style IPC call. Linux and gVisor do not have efficient mechanisms for this, but existing research on microkernels suggests that the hardware-protection performance limit is around 400 cycles [22, 53], which would imply that IPC with LFI could be substantially faster than even highly optimized existing microkernel IPC.

With currently available sandboxing techniques, there appears to be a tradeoff between average CPU overhead and context switch performance. LFI demonstrates that state-of-the-art software sandboxing can have CPU overheads much closer to virtualization, while still maintaining its advantage in context switch speed. We believe the performance could be further improved with direct hardware support, similar to that of virtualization, and provide some design recommendations in Section 7.3.

## 7 Discussion

In the last ten years there have been a number of developments related to SFI that are worth discussing in the context of LFI. Speculation exploits have been discovered, and can be particularly dangerous for SFI systems, which run mutually untrusting code in the same address space. New ISAs and ISA extensions are being developed, such as RISC-V, hardware control-flow enforcement, and Intel APX [62], which change how SFI may be implemented for various architectures. Finally, there have been proposals for hardware support for SFI, and we give our recommendations for the design of hardware with SFI in mind.

### 7.1 Side-channel Attacks

***Spectre.*** When a processor executes instructions speculatively, it flushes architectural state if those instructions turned out to be along an invalid path, but does not always flush microarchitectural state. Spectre attacks use mispredicted branches to speculatively execute instructions that then modify microarchitectural state to reveal sensitive information via side-channels. Following prior work [46], we are concerned with three types of Spectre attacks:

- Sandbox breakout: the sandbox uses mispredictions within itself to access data from outside the sandbox.
- Cross-sandbox poisoning: a sandbox trains the branch predictor so that it mispredicts a branch in another sandbox, causing it to speculatively execute a Spectre gadget that wouldn't otherwise exist. This gadget can then be used to extract sensitive information from the other sandbox.
- Host poisoning: the same attack as cross-sandbox poisoning, but applied to the host runtime instead of another sandbox.

Sandbox breakout attacks work by using speculation to circumvent CFI guarantees. This is effective against WebAssembly, which requires CFI and enforces it via the trusted compiler and some dynamic checks for indirect calls. By contrast, LFI does not rely on any fine-grained CFI guarantees: jumping anywhere in the sandbox — speculatively or not — is legal, and LFI enforces that all jumps remain in the sandbox with simple data-flow-based guards. As a result, LFI mitigates sandbox breakout attacks by construction.

The remaining two poisoning attacks are more difficult to use, but also more difficult to mitigate. The straightforward solution is to use Arm's `CSV2_2` extension for Spectre mitigation [6], which introduces software context numbers via the system register `SCXTNUM_EL0`. The hardware enforces that a program running within one software context cannot be influenced by a program running in another software context through the use of side-channels using "branch prediction and similar resources." [7] The runtime and each sandbox would be assigned a different software context number, preventing them from sharing speculation resources. This extension is present in Arm's latest processors starting with the Cortex-X2 generation. This approach does require modifying `SCXTNUM_EL0` when entering and leaving the runtime, which will likely have some cost[8].

[8]Linux does not yet enable this feature and hardware with support for this feature is not yet widely available, so we were not easily able to measure the cost of changing the software context.

***General side-channels.*** With hardware support for software context numbers, LFI processes are susceptible to side-channel attacks in mostly the same ways as processes that use hardware protection. However, software protection can be additionally helpful for preventing side-channel attacks in certain cases. Since the system already uses a static verifier, disallowing exploitable instructions when vulnerabilities are discovered becomes easy. For example, recent work has shown that the ARM64 LL/SC instructions can be used to perform a timerless side-channel attack on M1 Macs [67]. This kind of exploit is traditionally difficult to mitigate without hardware modifications, but with software protection those instructions can be simply disallowed by the verifier.

### 7.2 Application to Other Architectures

LFI primarily takes advantage of fixed-width encodings and 32-bit support within the ISA to provide an SFI scheme that is highly efficient while supporting tens of thousands of isolation domains. While our current implementation is for ARM64, we believe there are similarly efficient designs for both x86-64 and RISC-V, and present the initial designs we would use to port LFI to those architectures.

***x86-64.*** The x86-64 ISA is a traditionally difficult target for SFI since it has many instructions that can access memory and has a highly variable-length encoding. Native Client supported x86-64 via 32-byte-aligned chunks, and also made use of 32-bit support to accelerate SFI (but reserved much more virtual memory for a single sandbox than we would like). One of our goals is to avoid alignment constraints, and with the release and deployment of Intel CET [25], this would be possible by using hardware-enforced CFI. Hardware-enforced control flow would preclude the need for any alignment constraints, making it possible to reuse nearly the same design as for ARM64. The scheme for x86-64 would reserve one register (e.g., `%r15`) and place the sandbox base in a segment register (`%gs`). Memory operations can then be safely rewritten as 32-bit offsets from `%gs`, with the syntax `%gs:r15d`. We expect this would have comparable overhead to our ARM64 implementation. One implementation difficulty is that Linux currently does not support indirect branch tracking (a subset of Intel CET) in usermode, which is necessary to avoid relying on alignment constraints.

***RISC-V.*** The main difficulty in applying LFI to RISC-V is in including support for compressed instructions, since RISC-V does not currently have support for hardware-enforced CFI. If we enforce a minimal alignment constraint that every jump target must be 4-byte aligned, then we can uncompress instructions as necessary to meet this constraint, and in many cases still make use of compressed instructions (if two compressed instructions are side-by-side). This small alignment constraint prevents the ability to jump into the middle of 4-byte instructions, and we do not expect it to introduce much code size overhead. The `add.uw` instruction from the `Zba` extension can be used to perform an efficient guard, much like the ARM64 implementation. However, RISC-V does not have any register-register addressing modes, but if processor designs perform instruction fusion between common addressing instructions and memory operations, performance might still be comparable to the ARM64 implementation.

### 7.3 Recommendations for Hardware Design

We believe it is possible to design hardware with fast fault isolation in mind and believe this would look similar to past hardware segmentation implementations. Since a key factor is that changing protection domains must be fast, the design might still make use of a static verifier rather than privilege levels to prevent the execution of unsafe instructions. When sandboxing is active, the hardware would automatically perform masking for loads, stores, and indirect jumps as part of the instructions themselves. It would use a protection domain base register to directly set the top bits of the pointer before making an access.

If the architecture supports a variable-width instruction encoding, it is imperative to provide strong support for hardware-enforced CFI.

## 8 Related Work

***Software-based fault isolation.*** Our approach was heavily informed by the original SFI work by Wahbe et al. targeting MIPS and Alpha — particularly their use of reserved registers for safe sandboxing, and optimizations related to guard pages and stack references [61]. Due to somewhat expensive guard sequences (two bitwise operations), the SFI system presented by Wahbe et al. had higher average overheads (20% on SPEC 92). As x86 became a popular ISA it became a target for subsequent SFI projects [16, 39, 40, 55]. Pure software approaches continued to have high average overhead, though some systems used x86 segmentation (now discontinued) to improve sandboxing overhead [17, 66]. Many uses of SFI only isolate stores and jumps, such as for improving kernel reliability by separating components of the kernel [14, 38].

Google Native Client (NaCl) was a major project that initially applied SFI to x86 using segments [66], and then expanded to ARM32 and x86-64 using pure software techniques [52]. NaCl maintained low overhead across all of its target architectures (roughly 5-7% on SPEC 2000) by optimizing for the single-sandbox case. On ARM32, NaCl made use of single-instruction bitwise guards that limit the number of sandboxes to below 32. On x86-64 NaCl reserved 44GiB of virtual memory per sandbox, limiting the system to less than 3,000 sandboxes. Additionally, in practice, the NaCl implementation never supported more than 1 sandbox. Like LFI, NaCl x86-64 took advantage of instructions that operate on 32-bit subsets of 64-bit registers to accelerate SFI, and hence also limited the sandbox's memory to 4GiB in size. However, NaCl did not make use of any addressing modes that allow the SFI guard to be performed as part of the address calculation. NaCl also required alignment constraints rather than reserved registers for sandbox integrity, which caused code size increases due to padding (average code size overheads of 20% on ARM32 and 62% on x86-64).

Like many prior SFI systems, LFI uses a static verifier, inspired by the general concept of proof-carrying code [49]. Several projects have investigated increasing verifier security via formal verification and/or size reduction [28, 42, 69].

***Language-based isolation.*** There is a long history of using programming languages to provide security. One of the first systems to do so was Pilot [51], an operating system which required all code to be written in Mesa, a type-safe language. Similarly, the Singularity operating system was entirely implemented in a safe language called Sing#, and relied on the language and its runtime to provide isolation rather than any hardware mechanisms [24]. The SPIN operating system used Modula-3 to allow kernel extensions to be downloaded into the kernel [11]. More recently, some projects propose to use Rust in this domain, since it provides memory safety without the use of a garbage collector [12, 33, 48].

In order to support a wide variety of frontend languages, it is also possible to design a safe language meant to be used as an intermediate representation. Examples of this include WebAssembly [20] and Java Bytecode [19]. WebAssembly compilers insert dynamic checks for memory accesses and indirect branches, and use a runtime to implement system calls. Since WebAssembly programs are 32-bit, memories are limited to 4GiB. Expensive bounds checks can be omitted by reserving at least 6GiB of virtual memory per linear memory, since Wasm allows a 32-bit displacement from the 32-bit address for loads and stores.

Language-based isolation typically suffers from using a trusted compiler, but typed assembly language can aid to reduce TCB size by incorporating the language-based safety features into a pseudo-assembly language that can be efficiently compiled to machine code without requiring further optimizations [41, 43, 65]. Another approach is to adapt SFI-style verifiers to language-based systems, as done by VeriWasm [28]. However, this approach involves coordination between the verifier and the compiler's optimizer, which can result in a more complex and slower verifier, and has not been achieved with an advanced compiler such as LLVM.

***Hardware-based isolation.*** The typical mechanism for the isolation of untrusted code is hardware protection via the operating system. Operating systems make use of pagetables to give each process a separate virtual address space, and mark the kernel as inaccessible to user code within that address space. In usermode, hardware protection can still be used in the form of virtualization or containerization, as discussed in Section 6.4. However, all of these approaches suffer from the high cost of changing hardware contexts, caused by privilege mode and pagetable switches, leading to the research of several alternate mechanisms [3, 34, 56]. Microkernels in particular suffer from high context switch costs, since they make use of frequent inter-process communication, and projects like L4 [22] have spent considerable effort optimizing for this case.

Given the overhead costs of software-based fault isolation, many prior projects have made use of alternative hardware protection mechanisms or designed their own. Several sandboxing systems for x86 made use of segmentation [17, 66], and more recently, systems for x86-64 have made use of Intel MPK [57, 59] or MPX [29]. Similarly, on ARM32, ARMlock made use of domains for fault isolation [70]. Approaches like MPK and ARM domains typically only allow for a small number of protection domains, or have to resort to page-based isolation to handle many domains. Some projects have gone further, proposing custom hardware extensions for fault isolation [47] and capability-based hardware [45, 64]. These approaches are typically more suited to fault isolation, but have not been widely deployed.

## 9 Conclusion

Lightweight sandboxes for running untrusted code are increasingly important, and software-based fault isolation is an effective way to provide lightweight isolation. Even though classic SFI is an old technique, it lost traction due to lack of efficient support for many sandboxes, as well as difficulties in applying it to complicated and 32-bit ISAs, performance/code size overheads, and the complexity of modifying massive compiler toolchains. This work shows that many of these issues can be resolved with new designs on available hardware. We present LFI, an optimized SFI implementation for ARM64, and implement it outside of a particular compiler toolchain, massively reducing engineering effort and avoiding the need to develop within a megasystem. We show that LFI has lower overheads than WebAssembly, a modern language-based sandboxing system, while supporting at least 1.5× more bounds-check-free sandboxes. On top of these performance gains, LFI benefits from a small static verifier to reduce TCB size, is hardened against basic Spectre attacks, and has broad support for language mechanisms such as exceptions, and ISA features such as SIMD.

## Acknowledgements

I'd like to thank my shepherd Amit Levy and the anonymous reviewers for their valuable feedback. I'm also grateful to David Mazières, Matthew Sotoudeh, Akshay Srivatsan, and Geoffrey Ramseyer for insightful discussions and for providing comments on drafts of the paper. This material is based upon work supported by Stanford FDCI and by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-2146755.

# A Artifact Appendix

## A.1 Abstract

This artifact demonstrates how to run the benchmarks performed in Section 6. Users must have a Linux ARM64 machine (running inside a VM is fine) — in the paper, we used a GCP T2A instance and an Apple M1 running Asahi Linux. Users must also have access to the SPEC 2017 benchmark suite, which we do not provide due to licensing. Alternatively, our artifact is set up to run CoreMark, which is openly accessible, but is not used in the paper's evaluation.

The instructions for this artifact are also available on GitHub at `zyedidia/lfi-artifact`.

The source code for LFI is available on GitHub at `zyedidia/lfi`.

## A.2 Artifact check-list (meta-information)

- **Program: SPEC 2017, not included.**
- **Compilation: LLVM, Clang, LLD, Wasmtime, WAMR, Wasm2c. All included.**
- **Binary: A pre-built Podman container is included, containing all necessary binaries except the SPEC 2017 benchmarks.**
- **Run-time environment: Linux 5+ with Podman.**
- **Hardware: ARM64 machine**
- **Metrics: Execution time, relative percent overhead.**
- **Output: Bar graphs, overhead numbers.**
- **Experiments: A pre-built container is included along with step-by-step instructions.**
- **How much disk space required (approximately)?: 20GB.**
- **How much time is needed to prepare workflow (approximately)?: 30 minutes.**
- **How much time is needed to complete experiments (approximately)?: 8 hours (on an M1 Mac).**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: MPL 2.0.**
- **Archived (provide DOI)?: 10.5281/zenodo.10694909**

## A.3 Description

### A.3.1 How to access.

The artifact can be downloaded from https://github.com/zyedidia/lfi-artifact/releases/tag/pre-built.

### A.3.2 Hardware dependencies.

ARM64 machine.

### A.3.3 Software dependencies.

Linux and Podman.

### A.3.4 Data sets.

The SPEC 2017 benchmark suite.

## A.4 Installation

### A.4.1 Install Podman.

```
sudo apt install podman
```

### A.4.2 Download and import the Podman container.

Download the file `lfi.tar.xz` from the releases page at `zyedidia/lfi-artifact`. Then import it into Podman:

```
podman import lfi.tar.xz lfi
```

### A.4.3 Download and install SPEC 2017.

Please install to `/home/$USER/cpu2017`.

```
sudo mount -t iso9660 -o ro,exec,loop \
    cpu2017.iso /mnt
cd /mnt
./install.sh
```

### A.4.4 Enter the Podman container.

```
podman run -v ~/cpu2017:/home/lfi/cpu2017:U \
  -it --user lfi --workdir /home/lfi --name lfi \
  --security-opt=seccomp=unconfined lfi /bin/bash
```

### A.4.5 Set up the benchmark.

```
./setup.sh
```

You will not need to re-run this command again, even if you restart the container, since this sets up the shared `cpu2017` directory.

### A.4.6 Run the basic test.

```
./fast-run-and-report.sh
```

This should take about 2 minutes to run and should produce plots in the `$HOME/cpu2017/stats` directory. This directory is accessible from both the host and from within the container. You should see a `wasm.png` and `opts.png` file containing plots for the `mcf_r` benchmark in test mode.

## A.5 Experiment workflow

Programs are compiled using the `lfi-clang` compiler located in `/opt/lfi/toolchain/bin`. Then they can be verified using the `lfi-verify` program and run with the `lfi-run` tool. See the `examples` directory for a simple hello world program.

The `bench-*.sh` scripts invoke the SPEC runner tool `runcpu` with the appropriate options and commands to run the benchmarks inside sandboxing systems. The files in `cpu2017/config` store the configurations for how benchmark files should be compiled. Benchmarks are run by our modified `specinvoke` tool, that wraps the default commands using a loader tool, such as `lfi-run` or `wasmtime`, defined by the `LOADER` environment variable.

SPEC places raw results in the `cpu2017/result` folder. Our `specstats` program then parses those results and calculates overheads and geometric means (see the `stats.sh` script which invokes this tool).

## A.6 Evaluation and expected results

### A.6.1 SPEC 2017.

From inside the container, run

```
./run-and-report.sh
```

This should take approximately 8 hours on an M1 Mac and will generate the graphs seen in Figures 3 (`opts.png`) and 4 (`wasm.png`). You should also be able to view the geomeans in each CSV file, as listed in Table 4 (which is just a summarization of the figures).

Expected results would be no more than 2% deviation on the same exact hardware setup, and no more than 5% deviation on a similar hardware setup.

#### A.6.2 Microbenchmarks.

To reproduce the microbenchmarks from Table 5 (without gVisor), run the following

```
cd microbenchmarks
./run-linux.sh
./run-lfi.sh
```

For the `run-linux.sh` script you may see a benefit to process pinning using `taskset`. We included a comment in the script, but we don't automatically do it since you should select a P-core (if your machine has P-cores vs E-cores). For example, processor 5 is a P-core on an M1 chip (processor 0 is not). We used taskset during the evaluation, so if you don't you might see worse performance for Linux than we reported.

Running gVisor is optional. If your setup can support gVisor (4K pages), you can run the benchmarks with gVisor as well. Unfortunately these benchmarks cannot be run from inside Podman, so you must copy the directory to your host and run the binaries there. You may need to wait 10x as long for the gVisor benchmarks to complete, since these benchmarks are significantly slower with gVisor than with Linux. Alternatively, you can decrease the number of iterations in `pipe.c` and recompile. Run these commands outside the container:

```
podman cp lfi:/home/lfi/microbenchmarks .
cd microbenchmarks
sudo ./gvisor/runsc --network none do /bin/bash
# ./run-linux.sh
```

#### A.6.3 CoreMark (optional).

If you don't have access to SPEC 2017, you can run CoreMark with

```
cd coremark
./bench.sh
```

The results we saw on an M1 machine are in the `expected-m1/` folder.

### A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

## References

[1] Bytecode Alliance. Wasmtime, 2023. URL: https://wasmtime.dev/.

[2] Bytecode Alliance. Webassembly micro runtime, 2023. URL: https://bytecodealliance.github.io/wamr.dev/.

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 95–109, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/121132.121151.

[4] Arm. *Arm Architecture Reference Manual for A-profile architecture*, 2016. FEAT_LVA, p. A2-90.

[5] Arm. *Arm Architecture Reference Manual for A-profile architecture*, 2016.

[6] Arm. *Arm Architecture Reference Manual for A-profile architecture*, 2016. FEAT_CSV2_2, p. A2-76.

[7] Arm. *Arm Architecture Reference Manual for A-profile architecture*, 2016. SCXTNUM_EL0, EL0 Read/Write Software Context Number, p. D13-5913.

[8] Arm. *Arm Cortex-X3 Core: Software Optimization Guide*, r1p2 edition, 2021.

[9] Arm. Exploration tools, 2023. URL: https://developer.arm.com/downloads/-/exploration-tools.

[10] Wasm2c authors. wasm2c: Convert wasm files to c source and header, 2023. URL: https://github.com/WebAssembly/wabt/tree/main/wasm2c.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995. doi:10.1145/224057.224077.

[12] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: An experiment in operating system structure and state management. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3185768.3185771.

[14] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 45–58, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1629575.1629581.

[15] Frank Denis. Performance of webassembly runtimes in 2023, 2023. URL: https://00f.net/2023/01/04/webassembly-benchmark-2023/.

[16] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 75–88, USA, 2006. USENIX Association.

[17] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In Rebecca Isaacs and Yuanyuan Zhou, editors, *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 293–306. USENIX Association, 2008. URL: http://www.usenix.org/events/usenix08/tech/full_papers/ford/ford.pdf.

[18] Google. gvisor documentation, 2021. URL: https://gvisor.dev/docs/.

[19] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1996.

[20] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062363.

[21] Liang He. A problem about "traps in dead code", 2023. URL: https://github.com/bytecodealliance/wasm-micro-runtime/issues/2773.

[22] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1), apr 2016. doi:10.1145/2893177.

[23] Pat Hickey. How fastly and the developer community are investing in the webassembly ecosystem, 2020. URL: https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem.

[24] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, apr 2007. doi:10.1145/1243418.1243424.

[25] Intel. *Intel 64 and IA-32 architectures software developer's manual*, 2020.

[26] RISC-V International. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, 20211203 edition, 2021. sv57, p. 87.

[27] Dougall Johnson. Apple microarchitecture research, 2021. URL: https://dougallj.github.io/applecpu/firestorm.html.

[28] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Доверя́й, но проверя́й: SFI safety for native-compiled Wasm. In *NDSS*. Internet Society, 2021.

[29] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 437–452, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064217.

[30] Dexter Kozen. Language-based security. In Mirosław Kutyłowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999*, pages 284–298, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[31] Andrew Lamoureux. Ground-up aarch64, 2021. URL: https://binary.ninja/2021/04/05/groundup-aarch64.html.

[32] Michael Larabel. Amazon graviton3 vs. intel xeon vs. amd epyc performance, 2022. URL: https://www.phoronix.com/review/graviton3-amd-intel.

[33] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132786.

[34] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 49–64, USA, 2016. USENIX Association.

[35] LLVM. Aarch64: compress jump tables to minimum size needed to reach destinations, 2017. URL: https://reviews.llvm.org/D32564.

[36] LLVM. Assembling a complete toolchain, 2023. URL: https://clang.llvm.org/docs/Toolchain.html.

[37] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132763.

[38] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 115–128, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043568.

[39] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a cisc architecture. 2005.

[40] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USA, 2006. USENIX Association.

[41] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *J. Funct. Program.*, 12(1):43–88, jan 2002. doi:10.1017/S0956796801004178.

[42] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: Better, faster, stronger sfi for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 395–404, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254111.

[43] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, may 1999. doi:10.1145/319301.319345.

[44] musl. musl libc, 2023. URL: https://musl.libc.org/.

[45] G. J. Myers and B. R. S. Buckingham. A hardware implementation of capability-based addressing. *SIGARCH Comput. Archit. News*, 8(6):12–24, oct 1980. doi:10.1145/641914.641916.

[46] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. Swivel: Hardening webassembly against spectre. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1433–1450. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

[47] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 266–281, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3582016.3582023.

[48] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[49] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/263699.263712.

[50] NodeJS. uvwasi, 2023. URL: https://github.com/nodejs/uvwasi.

[51] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An operating system for a personal computer. *Commun. ACM*, 23(2):81–92, feb 1980. doi:10.1145/358818.358822.

[52] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 1–12. USENIX Association, 2010. URL: http://www.usenix.org/events/sec10/tech/full_papers/Sehr.pdf.

[53] seL4. sel4 performance, 2023. URL: https://sel4.systems/About/Performance/home.pml.

[54] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[55] Christopher Small. A tool for constructing safe extensible c++ systems. In *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, COOTS'97, page 13, USA, 1997. USENIX Association.

[56] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 33–46, USA, 2010. USENIX Association.

[57] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381326.

[58] Gang Tan. *Principles and Implementation Techniques of Software-Based Fault Isolation.* Now Publishers Inc., Hanover, MA, USA, 2017.

[59] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. Erim: Secure, efficient inprocess isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1221–1238, USA, 2019. USENIX Association.

[60] Kenton Varda. Webassembly on cloudflare workers, 2018. URL: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

[61] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Andrew P. Black and Barbara Liskov, editors, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 203–216. ACM, 1993. doi:10.1145/168619.168635.

[62] Sebastian Winkel and Jason Agron. Introducing intel® advanced performance extensions (intel® apx), 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html.

[63] Keith Winstein. wasm2c: run tests with -o2 on non-windows, 2022. URL: https://github.com/WebAssembly/wabt/pull/1939.

[64] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, page 457–468. IEEE Press, 2014.

[65] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 99–110, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1806596.1806610.

[66] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, 2010. doi:10.1145/1629175.1629203.

[67] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization storage channels (S2C): Timer-less cache Side-Channel attacks on the apple m1 via hardware synchronization instructions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1973–1990, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong.

[68] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 29–40, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2046707.2046713.

[69] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. Armor: Fully verified software fault isolation. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, page 289–298, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2038642.2038687.

[70] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 558–569, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660267.2660344.