LFI-XL: Scaling Software-Based Sandboxing to Modern Applications

Zachary Yedidia*, Abhishek Sharma†, Anand Balaji†, Taehyun Noh†,
Tal Garfinkel‡, Hovav Shacham§, Shravan Narayan†
*Stanford University †UT Austin ‡Google §UC San Diego

***Abstract*—Software-based fault isolation (SFI) confines memory-safety bugs in native code, but modern SFI schemes limit sandboxes to 4GiB in size to exploit 32-bit addressing tricks for low overhead. This cap makes it difficult to use SFI systems both in applications like browser JavaScript engines that require large address spaces and on platforms with small virtual address spaces like Android. We show the cap is often unnecessary and we present efficient SFI designs for AArch64 and x86-64 that support sandboxes sized to any power of 2. We additionally show how modern hardware features can be used to accelerate SFI, which is particularly important for low-overhead full isolation in large sandboxes. In the stores-only mode usable by production browser deployments, our software-based designs have 2% (AArch64)/7% (x86-64) overhead compared to native code on SPEC 2017, versus 1%/4% for equivalent software-based 4GiB schemes. Full load-and-store isolation can drop as low as 1%/3% with hardware support for control flow (Intel CET) and data access (Intel MPK/Arm POE), with some constraints on sandbox count. We evaluate our design by sandboxing libraries in the Firefox browser and the SpiderMonkey JavaScript engine.**

1. Introduction

Memory-safety vulnerabilities remain the dominant class of security bugs in C/C++ systems, accounting for 50–70% of all bugs in Windows, Chrome, and Android in long-term studies [1], [2], [3]. A promising way to address this is software-based fault isolation (SFI). SFI uses specialized compilers to confine untrusted application components to a specified memory region by inserting runtime checks during compilation. SFI tools thus confine memory-safety bugs, preventing them from corrupting the whole program.

While SFI has been successfully deployed in a few settings, it has not seen widespread use despite decades of research. One reason for this is the numerous limitations of SFI tools—limitations that are not fundamental but currently hamper adoption. The most notable limitation is that modern SFI tools almost universally fix sandboxes at an inflexible 4GiB virtual memory footprint on 64-bit architectures [4], [5], [6], [7], [8], [9], [10] for performance reasons (§5).

Unfortunately, this limitation has many real-world implications for SFI as a security tool. For instance, current SFI tools cannot sandbox JavaScript engines in browsers, even though browsers like Firefox already deploy SFI successfully to isolate third-party libraries [11], [12]. This is unfortunate, because the JavaScript engine is a popular target for in-the-wild attacks [13], [14], [15], but its support for memory-consuming web features such as ArrayBuffers and WebAssembly makes it particularly challenging to sandbox.

This memory limit also hurts current deployments of small-footprint sandboxes. Fastly, for example, uses SFI to isolate different tenants in its edge platform; these typically have memory footprints of a few MiB but must reserve 4GiB of virtual memory, limiting the total number of sandboxes to 16k on commodity hardware—a limit they were reportedly close to hitting in production even as of several years ago [16]. Deployments of SFI for Firefox on Android also face this problem when sandboxing libraries, as Android employs a 39-bit address space, resulting in a limit of 64 sandboxes[1]. This has led to out-of-memory crashes in production, and even discussions of falling back to slower SFI techniques without this restriction [17]. Thus, we need SFI tools that can not merely create large sandboxes, but also create flexibly sized sandboxes as needed by the domain.

Sandboxing modern application components, however, requires addressing more than just the SFI memory limits. SFI tools [4], [5], [6] today make numerous other design choices that compromise their applicability for component sandboxing—choices that are contrary to SFI's best practices for easy adoption and maintainability. These practices were discussed when SFI was first introduced in 1993 [18], but are often overlooked in modern implementations.

As one example, SFI tools today employ designs targeted for full sandboxing of all memory loads and stores, but these designs are not capable of providing a fast stores-only sandbox—a sandbox that only pays the required performance penalties to ensure stores are isolated. The "stores-only sandbox" mode offers a valuable option when sandboxing JavaScript engines—it has much lower overheads than a full sandbox (§7), while providing sufficient isolation to ensure security in browsers. Specifically, browsers employ site isolation [19] so each site's data is managed by separate processes. Thus, sandboxing a JavaScript engine within such a process only needs to protect the integrity of host memory (and not its confidentiality).

As another example, SFI tools today also modify the application binary interface (ABI) to reduce the memory footprint of sandboxed code (§5.3); however, this breaks

1. §7.5 details how the total sandbox limit was calculated.

compatibility of data structures shared between sandboxed and unsandboxed code [11].

The *goal* of this paper, motivated by these challenges, is to re-examine how to design and build SFI tools that retain efficiency with a flexible memory limit. Additionally, we aim to solve two real deployment challenges in today's SFI tools: (1) sandboxing code found in complex systems like JavaScript engines, and (2) following SFI's best practices for easy adoption and maintainability.

A *non-goal* of this paper is to build the fastest SFI tool by using a specialized scheme that trades off general deployability—for instance, schemes that only allow one sandbox [10] or rely exclusively on hardware features supported on a small fraction of architectures and operating systems [20], [21], [22], [23], [24], [25]. We may use hardware extensions as optimizations, but never as requirements without which the tool cannot operate.

Our solution. In this paper, we present LFI-XL, our SFI tool that is carefully designed to address these gaps without compromising on efficiency. LFI-XL is a high-quality, maintainable SFI implementation that supports flexibly sized sandboxes of any power-of-two size for x86-64 and AArch64—the most commonly used consumer hardware platforms.

LFI-XL safely supports sandboxing the complex code seen in real-world systems including sandboxing of hand-written assembly (with only minor modifications required to the assembly) and sandboxing of just-in-time (JIT) compilers—and transitively the code they produce—features that almost no current production SFI tool [4], [6] provides. Additionally, LFI-XL can use widely available hardware extensions like the shadow stack provided by Intel Control-Flow Enforcement Technology (CET) on x86-64 to optimize SFI. Importantly, unlike prior work, these are layered on top of a software-only SFI scheme as optimizations rather than as a standalone scheme, so LFI-XL works on all platforms—falling back to the software-only scheme if hardware extensions are unavailable.

LFI-XL also follows the best practices (implicitly) suggested in Wahbe's original SFI paper [18]. LFI-XL does so through careful trade-offs across four categories: (1) *Compatibility*: a variable-size sandbox that runs with the same ABI as the host application, (2) *Performance*: a fast stores-only masking mode and optimized handling of overlooked hot paths like thread-local storage, (3) *Security*: reduced complexity via a simple binary validator and an in-sandbox stack, and (4) *Maintainability*: minimally invasive patches to production compilers that avoid complex optimizations.

We evaluate LFI-XL's performance on the standard SPEC 2017 benchmark suite as well as additional benchmarks specifically for large and small sandbox configurations. For the large sandbox configuration, we evaluate the performance of sandboxing the high-performance SpiderMonkey JavaScript engine used by the Firefox browser. SpiderMonkey consists of a mix of C++ code and assembly, employs multiple interpreters and just-in-time code compilers [26], and, like all JIT compilers, employs complicated optimizations [27]. We also include a server workload benchmark (§7.4). For the small sandbox configuration, we evaluate the performance of sandboxing buggy third-party libraries in Firefox [11] running on Android—an important use case due to Android's limited virtual memory [17].

We observe that LFI-XL incurs overheads of 16.0%–22.5% for full sandboxing on SPEC 2017 across several x86-64 and AArch64 platforms, and stores-only overheads of 2.8% and 7.9% on AArch64 and x86-64, respectively. On SpiderMonkey, LFI-XL's stores-only large sandbox lowers scores by 16.4% on x86-64 and 11.3% on AArch64 when running the industry-standard JetStream benchmark [28]. On library sandboxing in Android, LFI-XL's small sandbox incurs 20.4% overhead when sandboxing a font-rendering library in Firefox (versus the much larger 50.1% overhead Firefox incurs today with WebAssembly [12]).

When LFI-XL is permitted to take advantage of hardware extensions such as Intel Memory Protection Keys (MPK) or, in the future, Arm Permission Overlay Extension (POE), together with Intel CET, overheads drop to 2.5% on SPEC 2017 and SpiderMonkey's score loss to 7.2% on Intel x86-64.

This paper makes the following contributions:

1) We design a practical flexible-size SFI scheme for AArch64 and x86-64 using architecture-specific insights, which supports a fast stores-only sandbox mode, maintains the application's ABI, and can optionally leverage hardware extensions when available.
2) We characterize the SFI design space (§5), comparing prior SFI optimizations and recommending choices by use case, workload, and maintainability.
3) We build LFI-XL, a performant, maintainable, and widely compatible sandbox tool in a production-scale compiler toolchain that supports sandboxing complex applications like JavaScript engines.

The LFI-XL artifact is available at https://github.com/zyedidia/lfixl-root.

2. Why Modern SFI Tools Fall Short

The original SFI schemes from Wahbe et al. [18] in 1993 did not suffer from many of the limitations discussed so far.

For instance, they could create sandboxes that had any size that was a power of two; could reduce overheads by only sandboxing writes (*stores-only sandboxing*); and came with simple binary validators that could ensure a binary was correctly sandboxed (eliminating the possibility that a compiler bug may compromise sandboxing).

This is because these original schemes were designed specifically for the use case of isolating untrusted components in applications. Despite numerous compatibility and performance improvements to Wahbe's original SFI scheme in the following years [9], [29], [30], [31], [32], [33], SFI saw very limited adoption until it was picked to address a different use case proposed by Wahbe [34]: isolating native code supplied by websites.

In the late 2000s, Google wanted to allow websites to supply native binary code (rather than JavaScript) to the Chrome browser, to be run directly on hardware to support performance-sensitive applications like games. At

that time, an untrusted binary could not safely be confined in a separate process—OSes like Linux lacked mechanisms to restrict system calls in child processes, and OS-level system-call restrictions were not considered robust enough to rely on—so Google's Native Client (NaCl) [4] chose SFI as its confinement mechanism. Browsers executing native code thus became the first production use case for SFI, and many SFI optimizations, tools, and research efforts target it.

While NaCl's use case looks similar to sandboxing untrusted components at first glance, it resulted in SFI schemes with fundamentally different trade-offs. For instance, NaCl's goal was to provide equivalent isolation to running the website-supplied code in a separate process; thus, there was no need to optimize for stores-only schemes[2]; NaCl did not have to follow the host machine's ABI—it could pick ABIs that offer performance and memory optimizations for its use case; NaCl was also originally designed when 32-bit computers were common and did not need to optimize for bigger memory footprints. While NaCl for 32-bit OSes [4] made only some of these changes, NaCl for 64-bit OSes [5] diverged more substantially, exacerbating these differences.

In 2017, browser developers deprecated NaCl [35] in favor of a more standardized solution: WebAssembly (Wasm) [6]. The rationale shifted slightly—SFI was chosen this time to allow fast in-process communication between native code and JavaScript code (which has access to the browser DOM APIs)—but the trade-offs remained the same: Wasm's SFI, like NaCl's, is optimized for the web use case. If anything, Wasm is an even worse fit for confining untrusted components in applications—for instance, it sacrifices performance [36] by restricting instructions and system calls to those available across many devices (an important requirement for web standardization).

However, applications nevertheless rely on tools like NaCl and Wasm to support SFI in their production use cases [11], [16], and in doing so inherit many of their domain-specific design decisions as limitations in the context of SFI's original use case of isolating untrusted components.

Why rely on NaCl/Wasm to isolate untrusted components? Despite these limitations, Wasm and even NaCl (many years after its deprecation) remain the gold standard for in-process isolation, offering high-quality compilers that can sandbox code. This is because substantial engineering is required to maintain a sandboxing compiler toolchain over the long term: most SFI-style toolchains (usually built for research papers) are stuck as forks of outdated Clang/GCC versions, and applications like SpiderMonkey, which follows the C++17 standard, are hard to compile with such old compilers.

While one could, in theory, try to build modern SFI tools using Wahbe's original scheme, such a tool would not have the multiple decades of SFI improvements that were needed to adapt SFI to modern architectures [7], [8], [9], [37].

There is a clear need for a modern, scalable, maintainable SFI toolchain that is optimized for sandboxing untrusted components in applications. In particular, this SFI toolchain and the SFI scheme it implements must be carefully designed to keep the changes (on top of production compilers like Clang/GCC) minimal and modular so that it may be maintained long term as the underlying compiler evolves. Ideally, the toolchain changes are carefully designed so they can easily be merged into upstream C compilers. In the next section, we present LFI-XL, our answer to this challenge.

Threat model. LFI-XL assumes sandboxed code is arbitrarily malicious: an attacker may fully control the bytes the sandbox executes, subject only to passing a trusted binary validator—a tool which checks that the executing code contains all required SFI checks prior to executing the code. The trusted computing base (TCB) consists only of the validator and an SFI runtime (which makes policy decisions such as which system calls are permitted). LFI-XL's compiler, which is built on LLVM, produces binaries containing SFI checks and is considered untrusted and thus not in the TCB. The sandbox is considered broken if the sandboxed code can read from or write to memory that does not belong to the sandbox, transfer control to any code outside the sandbox other than the runtime's designated entry points, or invoke a system call without mediation by the runtime; in stores-only sandboxing, out-of-bounds reads are permitted, while all other violations remain sandbox breaks. We focus on confining memory-safety attacks to sandbox memory; microarchitectural side channels (Spectre) and denial-of-service by sandboxed code (infinite loops) are out of scope. We assume hardware correctly implements the ISA subset the validator accepts.

3. LFI-XL: A Scalable SFI Toolchain

We start with an overview of the design goals and constraints of LFI-XL, our SFI toolchain that addresses the challenge of scalable SFI. These constraints were chosen with an eye to keeping it both scalable enough for real-world use cases and compact/modular enough to maintain or upstream back to LLVM in the longer term.

- **Efficient sandboxes with flexible memory limits.** LFI-XL's primary requirement is efficient support for flexibly chosen sandbox sizes, rather than the standard 4GiB footprint that most available tools offer. For scenarios where a 4GiB limit suffices, LFI-XL retains the well-known optimized schemes [8], [38], which it builds on.
- **Hardware for optimizations only.** LFI-XL should provide a high-quality SFI implementation for popular platforms like x86-64 and AArch64 across multiple generations of hardware: it may take advantage of hardware extensions in newer CPUs for more efficient isolation, but this must not compromise isolation on hardware without these extensions—application developers choose the scheme that fits them best.
- **Minimize implementation footprint.** LFI-XL should limit modifications of LLVM (the compiler it builds on) to avoid becoming the deep compiler forks seen in prior SFI toolchains [4]. The implementation should be self-contained in isolated compiler passes, leverage existing

2. Some of NaCl's publications discuss a stores-only scheme [5]; however, support in tooling was discontinued, possibly because NaCl's SFI design could not extract the performance benefits of this mode (§5.3).

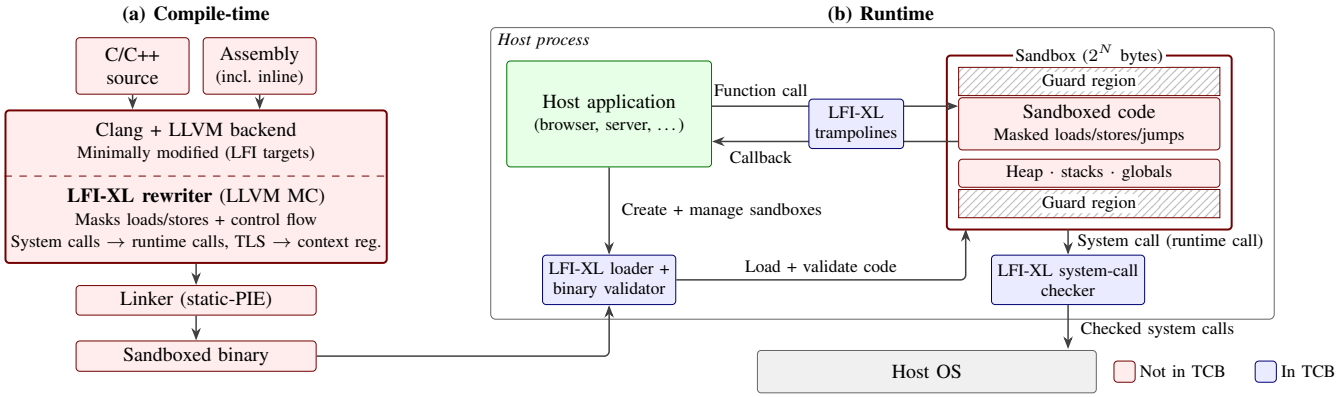


Figure 1: Overview of LFI-XL. (a) The compiler inserts SFI guards and rewrites system calls into runtime calls. (b) The runtime validates binaries before loading them, transfers control via trampolines, and enforces a system-call policy.

compiler infrastructure, and avoid touching unrelated parts of the compiler (such as the frontend and register handling). All flavors of SFI schemes (e.g., with and without hardware support) should share the same passes, infrastructure, and design.

- ▶ **Prioritize features that simplify adoption.** LFI-XL should let developers easily adopt sandboxing in their application—avoiding compatibility-breaking optimizations that change ABIs, calling conventions, file formats, etc. (all done in prior tools)—and let them opt in to much lower-overhead features like stores-only sandboxing when suitable for their domain.
- ▶ **Support complex applications.** LFI-XL must support complicated applications that use features like inline assembly and JIT compilation—the latter requiring an SFI scheme amenable to checking with a provided binary validator [18], [39], [40]—as well as features found in real applications that may not show in benchmark suites, such as efficient thread-local storage.

We designed LFI-XL by leveraging the learnings from many prior SFI implementations [4], [5], [6], [7], [8], [9], [16], [29], [30], [31], [32], [37], [41], [42], [43], [44], [45], [46]. We next describe LFI-XL's SFI scheme for x86-64 and AArch64 (§4), the concrete trade-offs behind it (§5), and how LFI-XL leverages hardware extensions as optimizations (§6).

4. LFI-XL's Flexible Sandbox SFI

In this section, we describe how LFI-XL implements a flexible SFI scheme and tooling that can sandbox code on the AArch64 and x86-64 platforms. A key aspect of LFI-XL's scheme is that it is simple by design, eschewing many of the complex design decisions made by prior SFI tools when the gains are marginal or come with compatibility or scaling limitations. We defer discussion of the specific design trade-offs that permitted this simplicity to §5, and instead first focus on the high-level SFI scheme.

Figure 1 shows LFI-XL's two halves: the compile-time pipeline that produces sandboxed binaries, and the runtime that loads, validates, and mediates the resulting sandboxes. LFI-XL's compile-time tooling implements tailored SFI schemes for AArch64 and x86-64. AArch64 is a RISC-style platform with plenty (32) of general-purpose registers (GPRs); thus, LFI-XL's scheme relies on reserving GPRs to perform efficient SFI checks. In contrast, x86-64 is relatively GPR-starved (16) with variable-length instructions; thus, LFI-XL's scheme minimizes reserved registers and adopts stronger control-flow policies to ensure only valid targets can be reached (e.g., jumps must not land between instructions).

At a high level, LFI-XL's schemes, like all SFI schemes, operate by inserting runtime guards into application binaries to restrict all memory accesses and all control flow to a specified memory region (the sandbox). The start of the sandbox memory is referred to as the *base* address, and is usually an address that is aligned to the size of the sandbox. The SFI guards are implemented as unconditional bitwise masks rather than explicit bounds checks to keep performance overheads small [18]. The combination of these masks/guards and the original instruction (the one being guarded) is referred to as a *macro-instruction* in SFI. We now discuss the specific macro-instructions we use for AArch64 and x86-64.

4.1. AArch64 Design

LFI-XL's AArch64 scheme is relatively simple and closely follows the spirit of the SFI schemes originally designed by Wahbe et al. [18]. Specifically, the availability of GPRs allows LFI-XL to assign specific types/roles to registers reserved for SFI.

Memory-access policy. For a sandbox of size 2^N bytes, LFI-XL fixes register `x27` to the aligned sandbox base address and reserves `x24` as the sandbox offset—a value in the range $[0, 2^N)$. Thus, any operation performed using `x24` relative to `x27` is guaranteed to be within the sandbox region. To guard a memory access, the compiler inserts an `and` instruction masking it with the constant `SIZE_MASK` ($2^N - 1$), producing an offset in `x24`. Due to the clever encoding of AArch64's `and` instruction, any mask (a run of zeros followed by ones) can be represented in an immediate

Arch	Original code	Large-sandbox equivalent
AArch64	`ldr rt, [xN]`	`and x24, xN, #MASK` `ldr rt, [x27, x24]`
AArch64	`str rt, [xN]`	`and x24, xN, #MASK` `str rt, [x27, x24]`

x86-64	`movq (%rN), %rax`	`movq %rN, %r11`
		`andq %r13, %r11`
		`movq (%r14,%r11), %rax`

x86-64	`movq %rax, (%rN)`	`movq %rN, %r11`
		`andq %r13, %r11`
		`movq %rax, (%r14,%r11)`

Table 1: Representative LFI-XL rewrites for loads and stores, shown for zero-displacement operands. Complex addressing modes apply the same guard to the effective address, folding the sandbox base into the memory operand when possible. Stores-only mode emits only the store rewrites.

operand without having to reserve an additional register. The sandbox size is therefore fixed at compile time—to the value of the immediate used in the `and` instruction. Table 1 shows the zero-displacement case. AArch64's register-offset addressing mode lets the compiler fold the base addition into the memory operation, so the guarded access uses `x27 + x24` without a separate `add`.

For accesses with immediates and scaled indices or when the register-offset addressing mode is unavailable, the masked effective address is materialized in a separate register. For this purpose, LFI-XL additionally reserves `x26` as a scratch register and `x28` as the sandbox address—a pointer within the sandbox region. The compiler inserts an explicit `add` instruction to compute `x27 + x24` into `x28`. Any subsequent operation through `x28` (with immediate ranges covered by the guard region §4.3) is guaranteed to be safe.

Control-flow policy. LFI-XL must also ensure that all indirect control transfers remain within the sandbox. Unlike x86-64 (§4.2), AArch64's fixed-size instructions mean that indirect control-flow transfers always land on valid instructions, negating the need for alignment restrictions such as bundling and label checks. Additionally, since register invariants are maintained at all control transfers, there is no need to prevent jumps from landing within a macro-instruction. For example, using a masked jump to transfer control directly to another jump (skipping past the mask) is safe, as the address register used by the jump is already guaranteed to be safe. To enforce this, the compiler inserts masks similar to those used for memory accesses. On the backward edge, control-flow integrity (CFI) is enforced by masking the link register on modification, ensuring that returns are always performed to a valid in-sandbox address.

4.2. x86-64 Design

As noted above, the primary differentiating design decisions made by LFI-XL's x86-64 scheme (in contrast to its AArch64 scheme) are motivated by platform specifics: (1) it reserves only three general-purpose registers for sandboxing—`%r14` as the sandbox base, `%r13` as the sandbox mask, and `%r11` as a scratch register—and (2) it uses bundling to enforce CFI, as x86-64 instructions are variable-length.

Memory-access policy. To perform a memory access operation (either a load or a store), the compiler copies the candidate pointer into the scratch register `%r11`, masks it with `%r13`, and then accesses memory at `%r14 + %r11`. Unlike AArch64, x86-64 has no encoding for an `and` instruction with a 64-bit mask immediate. While this makes the sandbox size dynamic—changing the mask in `%r13` changes the size without having to recompile sandboxed code—it comes at the cost of reserving an additional register.

x86-64's base-plus-index addressing mode lets the final base addition fold into the load or store, as shown in Table 1. The additional cost over LFI-XL's 4GiB scheme is therefore primarily the explicit copy and `andq`. More complex addressing modes follow the same rule as AArch64: keep the native addressing form when the guard region makes the displacement safe, and otherwise compute the effective address explicitly before masking.

A subtlety on x86-64 is that the `andq` instruction clobbers the flags, whereas the load or store it guards does not. In the implementation of the LFI-XL rewriter in LLVM, the compiler marks load/store instructions with information about whether the flags need to be preserved or not. For cases where the flags must be preserved, we fall back to masking via the `pextq` instruction (from the BMI2 extension) instead, which performs a similar masking operation without modifying the flags[3].

Control-flow policy. To perform safe indirect control-flow transfers (both on the forward and backward edge), LFI-XL uses a similar masking scheme to the one used on memory access. In addition to ensuring that control flow remains within the sandbox, LFI-XL must also ensure that jumps cannot skip SFI masks by landing within a macro-instruction. A further challenge on x86-64 is that instructions are variable-length, and indirect jumps must not be allowed to land in the middle of instructions.

Our software-only solution to this is bundling [9]—which ensures that individual instructions as well as macro-instructions all lie within 32-byte code "bundles". Here, the start of each bundle is the only valid landing pad, which is ensured by masking out the lower 5 bits of all indirect control-flow targets before jumping to them.

To ensure backward-edge control-flow transfers are safe, all `ret` instructions are transformed into `pop-mask-jmp` sequences, where the value popped off the stack is masked—directly attempting to mask values on the stack before a `ret` instruction could lead to a race condition where a concurrent thread modifies the stack after the mask is performed. LFI-XL additionally evaluates alternate label-based and hardware-backed control-flow policies (§6) historically proposed as alternatives to bundling; we discuss them in §5.7.

Table 2 provides a full summary of the LFI-XL schemes and compares them to the schemes developed by LFI [8],

3. The base x86-64 scheme permits the use of the BMI2 extension as this is now widely available—on over 97% of x86-64 machines in one survey [47].

[38]—a state-of-the-art SFI tool for 4GiB sandboxes. The details of LFI-XL's optimizations are discussed in §5.5.

4.3. Additional (Standard) LFI-XL Features

LFI-XL employs features common to most SFI toolchains: system-call policy enforcement, guard regions, a binary validator, and trampolines and springboards. While LFI-XL's use of these features does not significantly deviate from prior work [18], [37]—in particular, they follow the designs of Native Client [4], [5], [48] and LFI [8]—we describe them below to keep the discussion self-contained.

System-call policy. LFI-XL prevents sandboxed code from directly using system-call instructions, since unconstrained system calls would allow sandboxed code to break out of the sandbox [49]. All system calls are instead intercepted by LFI-XL's trusted SFI runtime, which checks their parameters and performs only those deemed safe. System-call instructions are rewritten into runtime calls through a read-only table of entry points that the runtime maps directly below the sandbox base (`ldur x30, [x27, #-8]; blr x30` on AArch64 and `leaq next(%rip), %r11; jmpq *-8(%r14)` on x86-64). The validator accepts only these sequences, so the runtime's entry points are the only host addresses reachable from the sandbox. The entry trampoline saves the sandbox registers, switches to the host stack, and calls a handler that implements about sixty Linux system calls, checks that pointer arguments lie within the sandbox, confines memory management to the sandbox, and allows pages to become executable only after running the validator.

Guard regions. Guard regions, one of the most important optimizations in SFI toolchains, are unmapped memory regions placed before and after the sandbox memory. They allow eliminating memory-address guards in specific situations, for example on x86-64's relative memory accesses with 32-bit signed offsets, when the guard size is 2GiB on either end of the sandbox. The larger the guard region, the more memory accesses can be performed guard-free, a valuable trade-off between performance and virtual-memory footprint.

Binary validators. LFI-XL follows SFI best practices [18], [39], [40] and provides a binary validator to eliminate its compiler from the TCB. When untrusted programs can generate new code at runtime (e.g., JIT compilers), validators go from "best practice" to "required for security", as the runtime must ensure that newly produced code conforms to SFI requirements. The validator makes a single linear pass and accepts a region only if every instruction is in an allow-list (which excludes privileged and system instructions such as `svc`, `syscall`, and `wrpkru`) and every macro-instruction has exactly the form the rewriter emits: guarded memory operands, reserved registers written only by their guard instructions, and, on x86-64, bundle-aligned targets with no instruction crossing a bundle boundary. The large-sandbox scheme adds a single invariant (namely, `x24` may only be written by `and x24, xN, #mask` with the configured sandbox mask). Since the validator is one of the main components of the TCB (apart from the SFI runtime), we keep it as small and simple as possible (§5.5): each validator is under a thousand lines of C, excluding the instruction decoder and generated tables.

Trampolines and springboards. LFI-XL provides trusted routines called trampolines and springboards to safely transfer control between the sandbox and host. On the way into the sandbox, springboards save host registers (clearing them when sandboxing reads), switch stacks, and set any reserved registers. Trampolines mirror this on the way out.

5. LFI-XL Design Trade-offs

LFI-XL bears some resemblance to many prior SFI schemes. However, it also makes specific departures to address the challenges of sandboxing modern applications, as outlined in §1. To be clear, our intention is not to discount the valuable contributions of work that discovered the techniques we opt not to use, but to instead discuss the trade-offs associated with SFI design, focusing on:

- ▶ Implementing SFI schemes that are compatible with various applications—through support for variable-size sandboxes (§5.1), minimal virtual memory footprints (§5.2), and an unmodified native host ABI (§5.3).
- ▶ Optimizing sandbox performance by adding a stores-only mode (§5.3) and speeding up often-overlooked hot paths like thread-local storage (§5.4, §5.5).
- ▶ Designing schemes that treat stack and heap memory uniformly (§5.6) and allowing for a simple binary validator (§5.5) that minimizes the TCB.
- ▶ Engineering maintainable SFI schemes with the help of minimal LLVM patches implemented at a stable layer, with no complex optimizations (§5.7, §5.8).

5.1. Avoids the 4GiB Design Point

As discussed, modern SFI schemes size the sandbox at exactly 4GiB so that they can take advantage of optimizations for address masks. Specifically, both x86-64 [7] and AArch64 [8] offer instructions that effectively perform a 4GiB SFI mask on an address as part of the memory access instruction itself—via segmentation-based addressing modes on x86-64 and the `uxtw` addressing mode on AArch64.

However, our observation is that the benefits of these constructions are more marginal than originally suggested—specifically, using a few extra instructions and a reserved register on both x86-64 and AArch64 still offers reasonable performance when used carefully (Table 1).

But even minor modifications to LFI-XL's scheme can amplify overheads for flexible sizes. For instance, eliminating even one reserved register on x86-64 pessimizes performance: replacing the mask register with inline constants bloats the code, and attempts to eliminate the base register by using the segment register to hold the base [7] fail because the segment register cannot be used for control-flow masks. Thus, we believe LFI-XL's design is a reasonable trade-off.

Parameter	AArch64		x86-64	
	LFI-XL (2^N)	LFI 4GiB	LFI-XL (2^N)	LFI 4GiB
Sandbox base register	`x27`	`x27`	`%r14`	`%gs`, `%r14`
Mask	`and` immediate	32-bit trunc.	`%r13`	32-bit trunc.
Scratch register	`x26`	`x26`	`%r11`	`%r11`
Address register	`x28`, `x24` (offset)	`x28`	–	–
Context (TP) register	`x25`	Runtime call†	`%gs`	Runtime call†
Sandbox size	2^N compile-time	4GiB	2^N runtime	4GiB
Guard region	4GiB (large)/128KiB (small)	≤4GiB	4GiB (large)/128KiB (small)	≤4GiB
CFI mechanism	masking (fixed-width ISA)		masking + 32B bundles (with CET shadow stack if available)	

Table 2: LFI-XL's SFI scheme parameters across architectures and sandbox sizes, compared to LFI [8], [38], a state-of-the-art 4GiB SFI tool. †As part of this work, upstream LFI now also reserves a register (`x25`/`%r15`) for the TP, instead of its original runtime call mechanism.

LFI-XL requires developers to supply the maximum sandbox size at compile time; this information is used to unlock size-specific optimizations.

4GiB sandbox optimization. When targeting a 4GiB sandbox, LFI-XL falls back to the existing well-optimized SFI schemes of prior tools—a sandbox of exactly 4GiB simply uses the LFI scheme [7], [8], [38].

5.2. Limits Guard Region Footprint

As described in §4.3, guard regions trade virtual-address footprint for fewer guards. Most SFI tools, including LFI-XL, cap them at 4GiB with some exceptions—NaCl's sandbox on 64-bit OSes [5], for instance, used 80GiB guard regions, improving performance but limiting the number of concurrent sandboxes to 250 [11]. LFI-XL's observation is that such scalability trade-offs are not necessary—4GiB guard regions afford excellent performance, and even much smaller guard regions unlock most of the performance benefits. This is particularly important for small sandboxes.

Small sandbox considerations. For small sandboxes, optimizing guard region footprint matters more. LFI-XL uses 128KiB guard regions in this case, and all memory accesses with displacements exceeding 128KiB are rewritten to load the address into a temporary before masking it. This keeps the footprint small enough to allow a large number of sandboxes while still avoiding nearly all rewrites.

5.3. Retains the ABI & a Fast Stores-Only Sandbox

SFI tools often use an optimization where pointers are represented by the compiler as offsets from the base of the sandbox [4], [6]. When paired with the 4GiB optimization, this allows compilers to define pointers as 32-bit integers, which can result in dramatic savings for some applications [50]. Unfortunately, this optimization comes at a great cost: it eliminates both ABI compatibility for sandboxed code and the ability for sandbox tools to offer efficient stores-only masking.

ABI compatibility. The compatibility challenge is visible even in simple code. Consider a 64-bit program that shares the following data structure with a sandboxed component.

```
struct foo { int* a; int b; }
```

The offset of field `b` according to the host program is 8 bytes, while the offset according to sandboxed code is 4 bytes. SFI compilers sometimes go even further [4], [6], defining sizes of basic datatypes like C's `long` as 4 bytes (rather than 8 bytes on 64-bit POSIX platforms). The argument against these sorts of changes is perhaps best demonstrated by the Linux kernel's deprecation of the x32 ABI [51] (an ABI giving x86-64 programs an x86-32-style address space for performance/memory savings): outside of special cases, the savings do not matter for most applications and do not justify the added toolchain complexity.

Stores-only masking. The use of offsets also eliminates the benefit of offering stores-only sandboxes. As originally proposed by Wahbe et al. [18], stores-only masking allows SFI schemes to eliminate all guards for loads—since loads typically account for two-thirds of an average program's memory accesses, this results in huge performance savings. However, SFI schemes that use an offset representation for pointers must always re-add the base prior to accessing a given offset, unconditionally adding a new instruction prior to performing a load—even when load sandboxing is not required.

LFI-XL's full 64-bit pointers avoid both issues: sandbox and host ABIs match, and loads can be left unsandboxed.

5.4. Optimizes Thread-Local Storage

Multithreaded programs rely on thread-local storage (TLS): per-thread objects stored in a memory region addressed through the current thread's thread pointer (TP), which platforms keep in a register (e.g., `%fs` on x86-64 Linux, `tpidr_el0` on AArch64 Linux). Sandboxes in SFI systems need their own TLS within the sandbox. We explore three potential approaches:

TP save/restore. The sandbox uses the same TP register as the host, with the trampoline swapping TPs on sandbox entry/exit. This is simple, but writing the TP register can be slow (roughly 40 cycles for `%fs` on x86-64), and clobbering the host TP conflicts with the platform: Windows context

switches may reset it, macOS's TP register is read-only, and on Linux it breaks signal handlers and GDB debugging.

TP runtime call. The sandbox's TP is stored in a thread-local host variable, and uses of the TP in sandbox code are rewritten into runtime calls that look it up (the approach NaCl took on x86-64). These calls are slow, prohibitively so for SpiderMonkey in our initial design. Worse, the host TLS lookup itself may require a function call (whenever the host library can be loaded with `dlopen`, and always on macOS), which would run unsafely on the sandbox stack. Avoiding this requires the *host* libc to reserve a dedicated TLS slot for the SFI system, which is usually not tolerable.

Dedicated TP register (LFI-XL). The simplest and fastest approach is to reserve a register holding a pointer to the TP and other thread-local SFI runtime state (the "context register"), which neatly solves both the performance and compatibility issues above. The cost is dedicating an entire register: a non-issue on AArch64 with its large register file, while on x86-64 we pay the cost or use `%gs` if it is available.

5.5. Keeps the SFI Validator Simple

Numerous clever SFI optimizations [18], [31], [37] have been proposed to optimize SFI guards, such as eliminating redundant guards or hoisting guards out of loops. While they improve performance [31], these optimizations complicate binary validators, which must reconstruct or check the logic justifying each eliminated guard. While such full-featured validators have been studied [32], [40], [52], maintaining them in production imposes an additional burden—in at least one instance [40], keeping the validator updated as compiler backends evolved required a complete rewrite [53]. Thus, LFI-XL avoids such optimizations.

However, LFI-XL can and does use some existing and novel SFI optimizations that do not complicate the validator. We describe a few examples of such optimizations below.

Guard the stack pointer on modification. Similar to prior SFI schemes [4], [18], both LFI-XL's AArch64 and x86-64 schemes mask the stack pointer on modification rather than on each memory access. This allows stack pushes and pops to rely on the guard region and largely remain unmasked.

Guard elimination. Unlike previous work [8], LFI-XL's AArch64 scheme takes a simpler approach to eliminating redundant guards that does not require reserving any additional registers or modifying the validator. Redundant guards of the same register (into `x28`) within a single basic block can be removed. When the rewriter emits a guard, it stores the source as the "active guard source." Subsequent guards of that register can be omitted. Jump targets (labels), modifications of the active guard source, or new guards all invalidate the active guard source. Since this optimization adheres to the existing scheme (simply removing certain modifications of `x28`), it does not require any changes to the validator and provides a 1–2% performance improvement.

5.6. Simplifies Stack Safety

LFI-XL simplifies stack safety by considering the stack unsafe, following the approach suggested by Wahbe and most SFI tools [4], [18]: sandboxed code is given a separate stack that is simply part of sandbox memory, with the trampolines and springboards (§4.3) switching stacks at domain crossings.

However, recent SFI tools like WebAssembly [6] and Zero-Cost [41] demonstrate an alternate approach in which sandboxed code shares the stack with the host, relying on the SafeStack [54] compilation pass to ensure that all stack operations are well-formed. In this model the sandbox memory contains only the heap, so a heap memory operation cannot corrupt the stack.

This was perhaps the most difficult trade-off LFI-XL had to make as there is no clear winner between these options.

Application compatibility. SafeStack relies on compiler instrumentation at the IR level, so applications with substantial hand-written assembly (e.g., video codecs like dav1d) or dynamically generated code (e.g., SpiderMonkey) cannot adopt it without significant re-engineering. Language features like `setjmp`/`longjmp`, C++ exceptions, and variadic arguments also need special handling in the SFI compiler. The separate stack has none of these compatibility issues.

Hardware acceleration. The shared stack approach is also incompatible with MPK/POE acceleration (§6.2). The safe stack resides outside the sandbox but must be accessed on nearly every call. SFI schemes protect it by permitting access only through a blessed stack-pointer register (e.g., `%rsp`), a distinction that page-granularity protection keys cannot express. Covering the safe stack with MPK instead requires one key per thread, quickly exhausting the 15 available keys. The separate stack, which is inside the sandbox and treats stack accesses as general accesses, is fully compatible.

Signal safety. The separate stack does create a signal-safety hazard for multithreaded sandboxes: a signal delivered during sandbox execution runs the host handler on the current stack, inside sandbox memory, where a parallel sandbox thread can modify the handler's frames. (The shared stack avoids this.) The standard remedy is an alternate signal stack via `sigaltstack`. Windows has no equivalent, but LFI-XL could use the CET shadow stack (SHSTK, §6.1) as a partial mitigation: SHSTK prevents a corrupted return address from redirecting the handler's control flow, though data-only attacks on the handler's stack remain possible. The complete fix must come from the OS: we recommend Microsoft add such a facility for Windows exception handlers.

5.7. Simplifies Control-Flow Safety (for x86-64)

As mentioned in §4.2, LFI-XL's x86-64 implementation relies on bundling to restrict control flow to safe targets, similar to prior systems [4], [5], [9]. However, bundling is not without its challenges: we discuss these below to justify why we believe this is still the right choice for LFI-XL.

While conceptually simple, bundling imposes a few trade-offs: it inflates code size via padding, incurs runtime cost

due to rewriting every return as a masked indirect jump (sacrificing return prediction), and has presented enough maintenance challenges for compiler developers that it was removed in LLVM 22 [55]. In designing LFI-XL, we explored whether some of these challenges were fundamental or could be addressed, and also considered alternatives to bundles.

Improving bundling's maintainability. Bundling was first added to LLVM by NaCl [4] around 15 years ago, and we observed that some of the maintainability criticisms of bundling [55] are specific to NaCl's implementation rather than the concept. NaCl implemented bundling in LLVM's assembly layer when the tooling was much simpler (e.g., before features like LTO). As the tooling evolved, the implementation got in the way of unrelated code paths—introducing performance regressions or blocking optimizations for code that did not rely on bundling. Using the insights stated by LLVM developers [55], we re-implemented bundling in a clean, modular way in around 400 lines of code in the latest LLVM.

Direct branch target alignment. Bundling requires indirect branch targets to be bundle-aligned, but on x86-64 direct branch targets can be left unaligned, since they are encoded in the instruction and can be statically checked by the validator. Leaving direct branch targets unaligned reduces binary size, as fewer targets must be padded to a bundle boundary, and conventional wisdom in prior SFI toolchains like NaCl [4] makes the case that this optimization is worthwhile. However, we find that it has little runtime performance effect in most situations, while adding complexity and performance overhead to the validator, which must now track instruction boundaries (not just bundle boundaries) to check that each direct branch lands on a valid instruction. This is particularly problematic for JITs that perform code replacement: the validator must additionally ensure that no direct branch in an already-validated page targets what has become the middle of an instruction in the replaced code. For these reasons, while LFI-XL leaves direct branch targets unaligned in ahead-of-time (AOT) compiled code, it requires even direct branch targets to be bundle-aligned when used in JIT scenarios.

CFI-style alternatives to bundling. Given the inherent performance overhead of bundling, we also considered alternatives to bundling, but found that these would not work well. For instance, we looked at securing control flow as WebAssembly does [6]—securing forward-edge indirect jumps by rewriting them as lookups into a table of valid targets generated by the compiler and securing backward edges by relying on the SafeStack [54] pass. Unfortunately, we found that this scheme reduced compatibility with the hand-written assembly in projects like SpiderMonkey, as indirection tables can be generated automatically only from high-level code. We also looked at a label-based CFI scheme [56] for both forward and backward edges—unfortunately, the performance of the forward-edge checks alone was of the same magnitude as bundling, so we gave up on this effort.

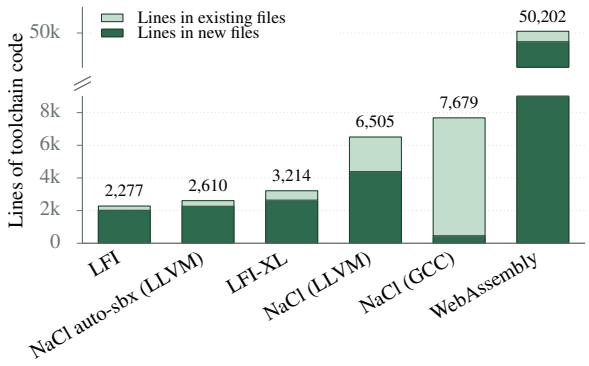


Figure 2: Lines of code (excluding comments and blanks) added to compiler toolchains to implement various SFI systems. LFI is measured from a version implemented in LLVM with support for AArch64 and x86-64.

5.8. Enables a Maintainable Implementation

LFI-XL's overall design (§3) is optimized from the outset to be implementable as one standalone pass that inserts the required SFI guards. This pass merges the robustness of integrated compiler SFI systems [4], [6] with the simplicity of assembly-rewriter SFI systems [8], [30], which leave the compiler mostly unchanged.

Handling hand-written assembly. To support projects with hand-written assembly, LFI-XL only requires that the assembly be manually modified to avoid LFI-XL's reserved registers (a straightforward change), with all remaining SFI transformations handled automatically by our toolchain.

Implementation. We implement LFI-XL as a pass within LLVM's assembler (the machine code (MC) layer), following the auto-sandboxing assembler approach adopted by NaCl after its deprecation [48]. Because all code—compiler-generated, inline assembly, or standalone assembly files—flows through the assembler, LFI-XL's guards are applied uniformly without modifying LLVM's frontend or optimization pipeline. The pass processes the code stream one instruction at a time, replacing each instruction with its macro-instruction immediately before encoding: it rewrites memory operands, indirect control flow, and writes to the stack pointer and link register, and replaces system-call instructions with runtime calls and thread-pointer accesses with context-register accesses. Every LFI-XL configuration uses the same pass with different rewrites enabled. Avoiding complex design elements like SafeStack (§5.6) and optimizations based on complex analysis (§5.5) also helps significantly. In total, our implementation is roughly 3,500 lines of code on top of standard Clang/LLVM, covering both architectures and a reimplementation of bundling. Additionally, when leveraging hardware to optimize LFI-XL (§6), we use hardware features as one-to-one replacements for LFI-XL's software mechanisms, toggled at the compiler level, so all schemes reuse the same compilation infrastructure. Figure 2 compares the toolchain changes needed for LFI-XL with those of

prior SFI systems. We see that NaCl modifies a lot of existing code and Wasm adds a lot of new code, making both harder to maintain. In contrast, LFI-XL builds on the MC infrastructure—similar to the approach of LFI and NaCl's (newer) auto-sandboxing assembler. LFI-XL demonstrates that with just a marginally larger implementation, we can get fully flexible-sized sandboxes for both AArch64 and x86-64 architectures.

6. Leveraging Hardware as Optimizations

By default, LFI-XL does not rely on hardware extensions, in order to remain compatible with different generations of hardware. However, developers can configure LFI-XL to use hardware features in modern CPUs that accelerate specific parts of SFI schemes. We describe how these features augment LFI-XL's existing software-based scheme.

LFI-XL is adapted to use Intel Control-Flow Enforcement Technology (CET) [57], Intel Memory Protection Keys (MPK) [58], and Arm Permission Overlay Extension (POE) [59][4]. This choice is motivated by extensive prior work showing the value of this hardware [20], [21], [25], [60], [61], [62], [63]. On x86-64, we add support for both MPK and CET, since LFI-XL's bundle-based CFI scheme on x86-64 limits performance. On AArch64, we only use POE because the existing SFI scheme has little control-flow overhead thanks to Arm's fixed-width instructions and dedicated return register.

In practice, hardware availability and software support for these features vary across platforms. Table 3 summarizes the current state: CET's SHSTK is the only feature broadly available both in hardware and across multiple operating systems, while MPK has been available for longer but is supported only on Linux.

6.1. CET for Control-Flow Integrity

LFI-XL uses CET [57] to accelerate its CFI scheme on x86-64. The CET hardware extension consists of two features: the *Hardware Shadow Stack (SHSTK)* which secures backward-edge control flow, and *Indirect Branch Tracking (IBT)* which can help secure forward-edge control flow. Since no major OS supports IBT for user code, we focus only on SHSTK.

CET's SHSTK feature augments the CPU with a hardware-checked shadow stack. Executing a `call` instruction pushes its return address to both the shadow and normal stacks. Executing a `ret` pops return addresses from both stacks and raises a CPU fault if they mismatch.

Accelerating SFI with SHSTK. LFI-XL utilizes SHSTK to secure its backward edge on x86-64 as an alternative to bundling, while continuing to rely on bundles for the forward edge. Using SHSTK to accelerate LFI-XL is straightforward. Rather than rewriting `ret` instructions into bundle-aligned jumps, LFI-XL can simply leave them unchanged. Additionally, return targets (instructions following `call`s) no longer need to be bundle-aligned. The hardware guarantees that the return target either originated from a `call` within the sandbox, or is the original `call` that entered the sandbox.

SHSTK additionally includes instructions for manipulating the shadow stack for non-local control flow (e.g., C++ exceptions), which LFI-XL's validator disallows. The sandbox must instead make runtime calls to unwind the shadow stack, which the SFI runtime does safely using these instructions (never unwinding beyond the return addresses pushed by the most recent call into the sandbox).

Importantly, using SHSTK provides a performance benefit because it allows the sandbox to use the `ret` instruction, which is optimized via speculative execution [64], [65].

Why CET is insufficient for JITs. While using CET to accelerate LFI-XL is straightforward in an AOT setting, its integration into JIT settings is more challenging. This is because SFI schemes that rely on SHSTK alone for the backward edge cannot safely confine a JIT compiler that performs code replacement (replacing existing code pages with new validated code). To see why, consider the following scenario within a sandbox:

1) A thread executes a `call` within a validated JIT code page, pushing its return address to the shadow stack.
2) Another JIT thread replaces the code page with new validated code. However, the bytes that previously followed the `call` (whose address is still on the first thread's shadow stack) now fall in the middle of a validated instruction in the new code.
3) The first thread then returns. The shadow stack allows this, but control now transfers to a potentially unsafe instruction stream inside another validated instruction.

The issue arises from the stale return address on the first thread's shadow stack when replacement takes place, despite both the original and new JIT code being validated. The only way to resolve this is to enforce that return targets in replaceable code pages are bundle-aligned. However, targets in AOT code that will never be replaced do not need to be aligned. When sandboxing SpiderMonkey with LFI-XL (§7.3), the JIT only generates bundle-aligned return targets, which the validator checks.

6.2. MPK/POE for Data Isolation

LFI-XL can similarly leverage Intel MPK and Arm POE to accelerate its data isolation. MPK/POE works by associating each page with one of 15/7 domains ("protection keys") via a system call; permissions of entire domains can then be toggled by writing to a register with a single user-space instruction—a very fast way to enable/disable access to parts of memory. LFI-XL's runtime tags all of a sandbox's memory with a single protection key and, on sandbox entry, sets the permission register to allow access only to pages with this key. The hardware then enforces data isolation directly, so the per-access load and store masks of Table 1 are simply not emitted. The validator

4. While POE hardware is not yet available, we consider it in our design as it would operate similarly to MPK.

Feature	Hardware availability	Linux	Windows
CET-SHSTK	Intel 11th Gen+ (2020); AMD Zen 3+ (2020)	6.6+ (2023)	W10 19H1+ (2020)
CET-IBT	Intel 11th Gen+ (2020); unsupported on AMD	Unsupported	Unsupported
MPK	Intel Skylake-SP+ (2017); AMD Zen 3+ (2020)	4.9+ (2016)	Unsupported
POE	Expected in next-generation Arm CPUs	6.12+ (2024)	Unsupported

Table 3: Hardware availability and OS support for the CFI and data-isolation extensions used by LFI-XL. IBT has no fundamental obstacle to OS support, but neither Linux nor Windows enables it for user code today.

must forbid sandboxed code from modifying the key register itself (`wrpkru` and `xrstor` on x86-64), as otherwise it could re-enable access to host memory, which is simple since LFI-XL already enforces control-flow integrity using bundles (or CET). What remains is LFI-XL's control-flow masks plus a `wrpkru` domain switch on sandbox entry and exit. We note that this optimization is not suitable for every domain, both because of its hardware requirements and because the limited keys allow only 15 and 7 sandboxes on x86-64 and AArch64, respectively.

7. Evaluation

We evaluate LFI-XL's performance by answering four questions:

- ▶ What is the cost of lifting the 4GiB cap? We compare LFI-XL's large-sandbox configurations against the equivalent LFI configurations on SPEC CPU 2017 for both full sandboxing and the stores-only mode relevant for browser content processes (§7.1).
- ▶ How much do hardware features help? We measure the improvement from CET-based backward-edge control-flow isolation and MPK/POE-based data isolation over software masking (§7.2).
- ▶ How efficient is the large-sandbox scheme on a complicated, security-critical application? We run the SpiderMonkey JavaScript engine inside LFI-XL on JetStream 2.2 (§7.3).
- ▶ How does the small-sandbox scheme perform in the library-sandboxing use case? We sandbox Firefox's font-rendering library and compare against LFI and the Wasm-based sandbox Firefox ships today (§7.5).

We additionally evaluate LFI-XL on a large server workload (§7.4) and measure the cost of sandbox transitions (§7.6).

Choosing a baseline. LFI-XL has no clearly comparable baseline: no SFI system available today supports efficient large sandboxes. The closest, WebAssembly's `memory64` proposal [66], lifts Wasm's 4GiB limit but is not a realistic baseline as it lacks a functional libc (the WebAssembly System Interface supports only Wasm32 [67]) and is extremely slow [68], since the Wasm specification effectively requires bounds checks rather than masks. We therefore measure LFI-XL's overhead relative to native (unsandboxed) code, and compare its large-sandbox configurations against the fastest SFI compilers we could find that run SPEC 2017 in a 4GiB sandbox: the most recent versions of LFI [8] for AArch64 and its extended x86-64 version [7], [38]. For library sandboxing (§7.5), we also compare with WebAssembly [6], the tool used in production [12].

Experimental setup. To ensure general applicability, we evaluate on four machines spanning both target architectures: two AArch64 (Arm Cortex-X925 and Apple M2) and two x86-64 (Intel Core i7-13700K and AMD Ryzen 9 7950X). Benchmarks run on Ubuntu 24.04 (Ubuntu Asahi bare-metal on the M2), with frequency scaling disabled and benchmarks pinned to a performance core. The one exception is the small-sandbox evaluation (§7.5), which we run on a Google Pixel 8 running Android 14 (AP2A.240605.024), an appropriately memory-constrained environment. Because the domain-switch microbenchmarks (§7.6) report absolute latencies, we run them at stock clock frequencies, still pinned to a single core. We compile every configuration (native baseline, LFI, and LFI-XL) with Clang/LLVM 23, Musl 1.2.5, and `libc++`. We characterize the measured configurations along two axes. The size axis contrasts LFI's 4GiB sandbox with LFI-XL's large and small sandbox schemes. The data-isolation axis varies what subset of data accesses are sandboxed:

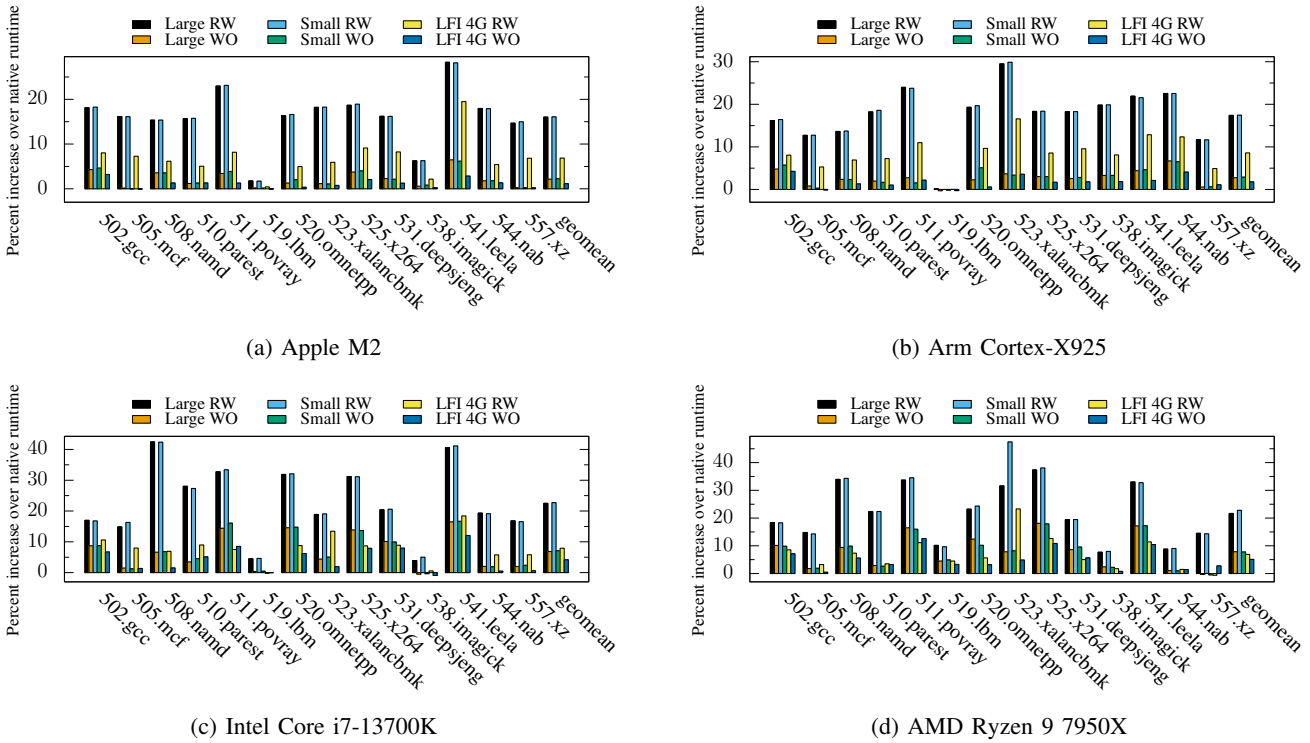
- ▶ **RW** (full): every load and every store is masked into the sandbox, providing full read-write isolation.
- ▶ **WO** (stores-only): only stores are masked; loads run unguarded, so the sandbox cannot corrupt host memory.
- ▶ **HW** (hardware-accelerated data masks): data masking uses MPK on x86-64. Since no current Arm hardware supports POE, this mode applies no data masks on AArch64. Control-flow masks match the software scheme on both platforms: bundling on x86-64 and the fixed-width scheme on AArch64.
- ▶ **HW+SHSTK** (hardware-accelerated data masks with a hardware shadow stack): data masks and forward-edge control-flow protections are enforced in the same way as the **HW** scheme, but the backward edge is protected with a hardware shadow stack.

7.1. Sandboxing SPEC 2017 with LFI-XL

We use the SPEC CPU 2017 `intrate` and `fprate` suites at reference inputs, restricting attention to the 14 C/C++ benchmarks our toolchain supports. We report overhead as the percent increase in run time over the native baseline, and summarize each configuration by the geometric mean across all benchmarks. Over three runs per benchmark, nearly all coefficients of variation are under 1%. Geomean overheads are shown in Table 4 (per-benchmark results are in Figure 3).

	SPEC CPU 2017								JetStream 2.2		
	LFI-XL Large		LFI-XL Small		LFI 4GiB				LFI-XL Large		
Platform	**RW**	**WO**	**RW**	**WO**	**RW**	**WO**	**HW**	**HW+SHSTK**	**WO**	**HW**	**HW+SHSTK**
Arm X925 (AArch64)	17.4%	2.8%	17.4%	2.9%	8.6%	1.8%	0.7%	–	11.3% (13.4%)	2.1%	–
Apple M2 (AArch64)	16.0%	2.1%	16.1%	2.3%	6.9%	1.1%	0.5%	–	15.0% (17.7%)	5.4%	–
Intel 13700K (x86-64)	22.5%	6.8%	22.7%	7.1%	7.9%	4.2%	3.0%	2.5%	16.4% (19.4%)	9.7%	7.2%
AMD 7950X (x86-64)	21.7%	7.9%	22.8%	7.8%	6.9%	5.1%	3.9%	5.2%	17.5% (20.0%)	8.2%	9.2%

Table 4: Left: geomean overhead relative to native on SPEC CPU 2017 for LFI-XL across sandbox-size schemes and data-isolation configurations. All x86-64 schemes use bundling. HW+SHSTK additionally uses CET to avoid rewriting `ret`. LFI's 4GiB sandbox performance is also shown as a reference. Right: JetStream 2.2 score reduction for SpiderMonkey running inside LFI-XL in the stores-only large-sandbox mode, with validation overhead in parentheses (HW modes have no validator). SHSTK requires CET, which AArch64 lacks.



(a) Apple M2

(b) Arm Cortex-X925

(c) Intel Core i7-13700K

(d) AMD Ryzen 9 7950X

Figure 3: Per-benchmark overhead of LFI-XL and LFI over native code on SPEC CPU 2017 for each evaluated machine.

Performance of full sandboxing. The cost of lifting the cap is concentrated almost entirely in load masking. In full RW mode, LFI-XL incurs 8.8/9.1 points of overhead over LFI on AArch64 and 14.6/14.8 points on x86-64, pushing LFI-XL's geomean to 16–17% on AArch64 and 22–23% on x86-64. Worst-case benchmarks also suffer: pointer-intensive C++ workloads such as `541.leela`, `523.xalancbmk`, and `520.omnetpp` approach or exceed 30%, and `541.leela` and `508.namd` exceed 40% on Intel.

Performance of stores-only sandboxing. In stores-only mode LFI-XL's overhead is 2.1%–2.8% on AArch64 and 6.8%–7.9% on x86-64. Lifting the 4GiB cap is nearly free in this mode: compared to LFI, LFI-XL adds only 1.0 points on AArch64 (1.8% → 2.8% on the X925) and 2.6–2.8 points on x86-64 (4.2% → 6.8% on Intel). The larger penalty on x86-64 reflects the rewrite in Table 1: AArch64 folds the mask into a single `and` immediate and the base into the addressing mode, whereas x86-64 must materialize the pointer into a scratch register and apply an explicit `andq` because it has no compact encoding for an arbitrary 64-bit mask. It must also sometimes use `pextq` rather than `andq` when flags must be preserved across the memory access.

Performance of the small-sandbox scheme. The small scheme's footprint savings cost essentially nothing over the large scheme: on the Arm X925, its geomean overhead is 17.4% in RW mode and 2.9% in stores-only mode. This is expected as the two schemes emit the same per-access masks and differ only in control-flow masking and the handling of large displacements. The large configuration additionally places code in the first 4GiB of the sandbox so that control-

flow instructions can use cheaper 4GiB guards, but this optimization has a negligible effect on overall performance.

Takeaways. Lifting the 4GiB cap in software is expensive for full sandboxing, though still useful in some settings. In stores-only mode the large sandbox is cheap enough to be practical in many domains, including web browsers. Hardware features improve performance further (§7.2).

7.2. Sandboxing SPEC 2017 with LFI-XL HW

MPK (x86-64) and POE (AArch64) enforce data isolation in hardware, eliminating the per-access masks entirely. The overhead of even a fully isolated large sandbox then collapses to that of the control-flow-only scheme, leaving only CFI cost plus a small domain-switch overhead, measured for MPK in Table 6 (§7.6). However, we caution that POE is not yet available on any hardware—our AArch64 numbers assume data masking is free (they elide all data masks)—and MPK remains unsupported on some operating systems, so this approach is not always viable.

The HW column of Table 4 isolates the cost of LFI-XL's control-flow integrity scheme. On AArch64 it is nearly free—0.5%–0.7%—because fixed-width instructions force every branch target to be aligned and the dedicated link register lets the validator constrain returns without rewriting them. On x86-64 the software bundling scheme is the source of baseline overhead (3.0% on Intel, 3.9% on AMD). On our Intel machine, SHSTK gives a small geomean improvement, with large gains on return-heavy benchmarks and a regression on `omnetpp`; we find that the AMD SHSTK performs worse than the bundling-based scheme. Figure 4 reports the per-benchmark results for the HW and HW+SHSTK configurations on both x86-64 machines.

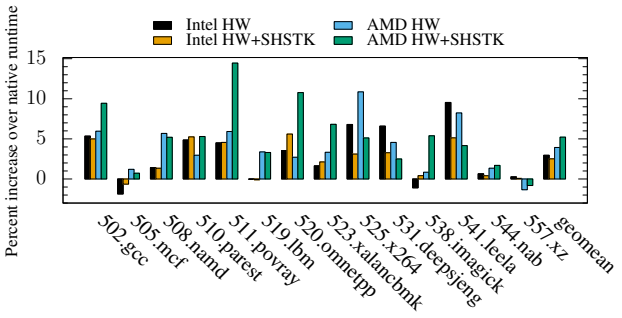


Figure 4: Per-benchmark overhead of LFI-XL's hardware-accelerated configurations over native code on SPEC 2017, for both x86-64 machines. HW enforces data isolation with MPK and the control-flow masks with bundling. HW+SHSTK additionally uses CET's hardware shadow stack to regain `ret` instructions.

These benchmarks make clear that AMD's shadow stack implementation is not as optimized as Intel's. We suspect this stems from the challenge of optimizing the microarchitecture while adhering to CET's strict Spectre-related guarantees—SHSTK returns may not speculatively execute until the values from the stack and shadow stack are compared [57].

7.3. Sandboxing a JS Engine with LFI-XL Large

To validate that LFI-XL supports a complex application that uses JIT compilation, we run the SpiderMonkey JavaScript engine inside an LFI-XL sandbox.

We integrate SpiderMonkey with LFI-XL in two steps: (1) modify the JIT compilers to emit native code compliant with LFI-XL's SFI scheme, and (2) make JIT code replacement and validation safe in the presence of concurrent sandbox threads. The first step, demonstrated by prior work [69], [70], [71], requires instrumenting the architecture-specific code emission backend shared by all JIT compilers to insert SFI macro-instructions. For the second step, we adopt Mohabi's approach [71]: JIT code pages are dual-mapped as read-executable within sandbox memory and writable in the LFI-XL runtime. Any JIT code replacement calls into LFI-XL's runtime, which validates the new code, makes the sandbox mapping read-only, and updates the JIT code pages via the external mapping.

Benchmark. We run the JetStream 2.2 [28] benchmark using SpiderMonkey's command-line JavaScript shell and report the reduction in JetStream score relative to native SpiderMonkey. We use the stores-only sandbox as this is sufficient for the browser threat model (§1) and use a large sandbox of size 1TiB to support modern web features with large footprints such as ArrayBuffers. On x86-64, we also evaluate LFI-XL with CET shadow stack acceleration. Unlike the setup in §7, we give the benchmark environment 2 cores so the engine can use off-thread compilation.

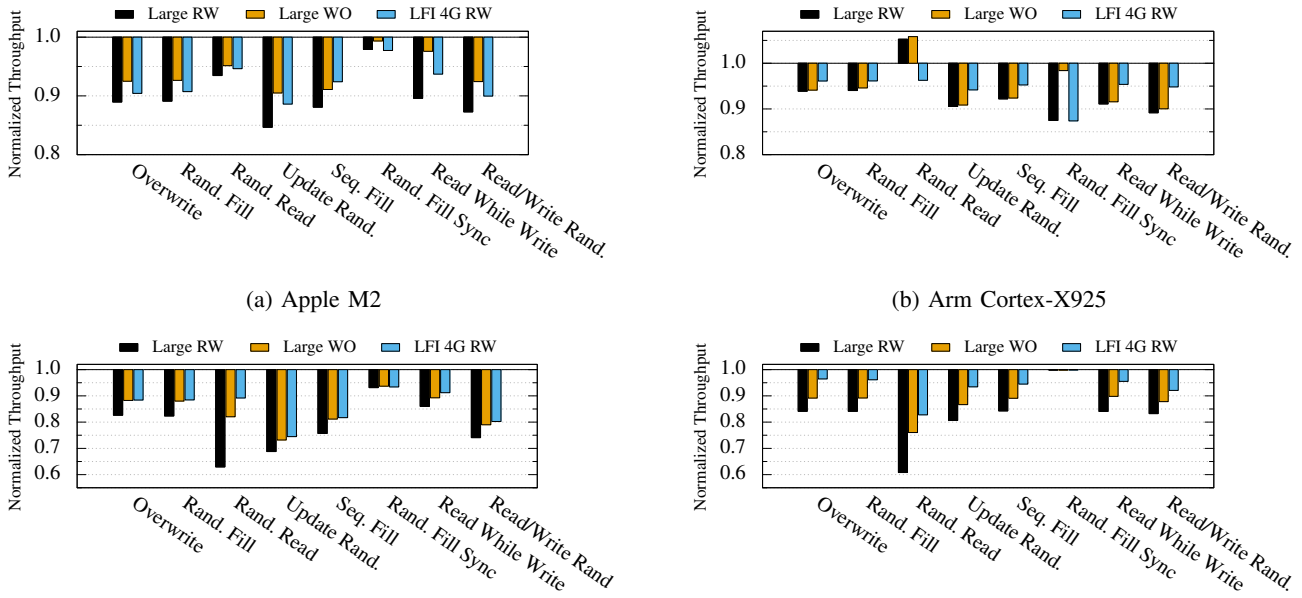
Results. Table 4 summarizes the results. Without runtime validation, SpiderMonkey sandboxed with LFI-XL scores 11–15% lower than native on AArch64 and 16–18% lower on x86-64. Verification lowers scores by a nearly uniform further 2–3 points across every configuration. The overall result aligns with our previous takeaway from AOT benchmarks.

7.4. Sandboxing Server Workloads with LFI-XL Large

We use RocksDB [72] to demonstrate LFI-XL's applicability to large server-side workloads. RocksDB is a persistent key-value store implemented as an embeddable C++ library.

Benchmark. We benchmark RocksDB 10.0.1 using workloads from version 1.7.0 of the Phoronix `pts/rocksdb` test profile [73]. We evaluate RocksDB compiled with LFI and the LFI-XL large-sandbox scheme in both full and stores-only modes, along with native RocksDB as the baseline. We use the same benchmark setup as described in §7, except that we leave 1 P-core for the Phoronix benchmark harness and allocate all the remaining P-cores to the benchmark environment. We run 15 trials of each workload and report the median operations per second, normalized per workload against the native median.

Results. Figure 5 shows the per-workload overheads on the four machines that we perform our evaluation on. On x86-64, LFI-XL's RW configuration has a maximum throughput loss



(a) Apple M2

(b) Arm Cortex-X925

(c) Intel Core i7-13700K

(d) AMD Ryzen 9 7950X

Figure 5: Per-workload LFI-XL and LFI RocksDB throughput relative to native for each evaluated machine.

Plat.	Metric	LFI-XL Small	Wasm *(Baseline)*	LFI *(Modern SFI)*
Pixel 8	Perf. Overhead	20.4%	50.1%	5.9%
	Mem. size	256MiB	4GiB	4GiB
	Guard size	128KiB	4GiB	128KiB
	Max inst. count	2,000	32	121

Table 5: Overhead and scalability of sandboxing the `libgraphite` font-rendering library in Firefox using LFI-XL Small in RW mode, compared with WebAssembly (which Firefox uses today [12]) and LFI [8], a state-of-the-art 4GiB SFI toolchain. For scalability, we show the usable memory per sandbox, the size of the guard region, and the maximum number of simultaneous sandbox instances that can be run.

of 37.1% and 39.2% (on Intel and AMD, respectively) on the Random Read workload. The performance of the WO large-sandbox configuration relative to LFI is consistent with the results from Table 4, with the LFI-XL WO configuration reclaiming the performance lost from moving to a large sandbox—even outperforming LFI on all but one workload on the Apple M2 with a maximum throughput loss of 9.50%.

7.5. Sandboxing a Library with LFI-XL Small

To evaluate LFI-XL's small-sandbox scheme, we measure its performance for Firefox's library sandboxing on Android—a platform with constrained (virtual) memory (§1). Firefox currently sandboxes libraries using WebAssembly [12], but WebAssembly's large virtual memory footprint can result in crashes due to virtual memory exhaustion [17], motivating use of LFI-XL with smaller sandboxes.

Benchmark. We modify Firefox to replace the sandboxing tool for one of its sandboxed libraries—the `libgraphite` font-rendering library—with LFI-XL Small configured with 256MiB of memory[5]. We take the Wasm sandbox that Firefox currently uses as our baseline and additionally modify Firefox to sandbox the library with LFI—for a comparison with an optimized 4GiB sandbox. We use the Graphite performance benchmark from prior work [7], [11], which measures the time taken to reflow a page that uses a Graphite font at different sizes, and report the median over 100 iterations.

Results. Table 5 summarizes the results. LFI-XL offers better performance and scalability than Wasm (the sandboxing solution used today). While LFI's 4GiB sandbox is faster than LFI-XL, this comes at the cost of compromising sandbox scalability. LFI-XL Small instead offers a useful middle ground: a moderate performance impact without compromising scalability.

Scalability. The maximum number of simultaneous sandboxes is bounded by the usable virtual address space per process divided by each sandbox's footprint, including guard regions. Commodity x86-64 and AArch64 systems provide 2^{47} and 2^{48} bytes of user address space, respectively, while Android exposes only 2^{39} bytes per process. A Wasm [6] sandbox occupies 8GiB (2^{33} bytes: a 4GiB sandbox followed by a 4GiB guard region), allowing 16,384 sandboxes on x86-64, 32,768 on AArch64, and only $2^{39}/2^{33} = 64$ on Android. A 256MiB (2^{28}-byte) LFI-XL Small sandbox instead places its 128KiB guard regions inside the sandbox, allowing 524,288 sandboxes on x86-64, 1,048,576 on AArch64, and 2,048 on Android, consistent with the 2,000 sandboxes

5. We chose 256MiB conservatively based on the fact that Firefox limits the Wasm sandbox's usable memory to 16MiB on 32-bit platforms [74].

	LFI-XL (ns)			Reference (ns)		
Platform	**Call**	**Callback**	**RT-call**	**Call**	**Syscall**	**Ctx. sw.**
Arm X925	18.8 ±<0.1	4.8 ±<0.1	10.1 ±0.1	0.52 ±0.04	83.9 ±0.6	1445 ±10
Apple M2	23.6 ±0.2	5.2 ±<0.1	14.8 ±<0.1	0.97 ±<0.01	122.1 ±0.1	2041 ±11
Intel 13700K	7.8 ±0.1	11.6 ±<0.1	8.4 ±<0.1			
cached `%gs`	6.0 ±0.2	10.6 ±<0.1	7.8 ±0.1	0.19 ±<0.01	57.5 ±0.1	1451 ±22
MPK	27.1 ±0.5	27.9 ±<0.1	27.0 ±<0.1			
AMD 7950X	15.8 ±0.8	11.6 ±<0.1	16.3 ±0.5			
cached `%gs`	7.3 ±0.5	5.0 ±0.1	8.4 ±0.1	0.80 ±0.01	217.2 ±0.2	4080 ±13
MPK	30.9 ±1.1	25.6 ±<0.1	31.6 ±0.4			

Table 6: LFI-XL's domain-switch microbenchmarks. *Call* is a host-to-sandbox call and return, *Callback* a sandbox-to-host call and return, and *RT-call* a sandbox system call handled by the LFI-XL runtime. The reference columns show a native function call, a host system call, and a process context-switch round trip. The x86-64 variants are the default (full `%gs` write), cached `%gs` (`wrgsbase` skipped when `%gs` is unchanged), and MPK. Values are the mean ± standard deviation of 10 runs at stock clock frequencies.

instantiated on the Pixel 8. For comparison, a 256GiB LFI-XL Large sandbox, whose 4GiB guard regions are shared by adjacent sandboxes, occupies 260GiB, allowing 504 sandboxes on x86-64, 1,008 on AArch64, and one on Android. These limits represent the ideal case: fragmentation, alignment requirements, and non-sandbox memory use lower them in practice, and when allocating very large numbers of sandboxes, Linux's `vm.max_map_count` limit of 65,530 mappings per process (which an administrator can raise) is exhausted first.

7.6. Performance of Sandbox Transitions

A deployable SFI system must also make crossing the sandbox boundary cheap, since applications such as browsers call into and out of sandboxed components frequently. The library isolation benchmark in §7.5 exercises sandbox transitions, but we also report separate microbenchmarks in Table 6 for domain switches. For reference we also report a native (unsandboxed) call, a host system call, and a full process context-switch round trip. LFI-XL's transitions cost a few to tens of nanoseconds—one to nearly three orders of magnitude below a process context switch, the coarser isolation mechanism they replace, and below the cost of a single host system call.

8. Related Work

SFI was introduced by Wahbe et al. [18] to confine an untrusted module to a memory region by masking the high bits of every computed address, a construction independent of the region's size. Later systems specialized this idea for commodity hardware: PittSFIeld [9] and vx32 [29] on x86, RockSalt [39] with a machine-checked verifier, and Google's Native Client [4], [5], [75], which combined bundle alignment with a validator to run untrusted x86 and Arm binaries in the browser. LFI-XL inherits this lineage (masking plus a validator) and, like the original construction but unlike its modern descendants, is not tied to a fixed sandbox size (§4).

4GiB sandboxes. Modern SFI has converged on a 4GiB sandbox so that address masks fold into 32-bit truncating operations (§5.1). WebAssembly [6] adopts a 32-bit linear memory for exactly this reason. Its `memory64` proposal [66] and `custom-page-sizes` proposal [76] relax this limit, but at a substantial performance cost [68]. Segue [7] reuses x86 segmentation to eliminate masks within 4GiB. LFI [8], [38] is a recent scheme covering AArch64 and x86-64 with the same 4GiB cap. LFI-XL, like LFI, supports a full-pointer ABI and a stores-only mode but removes the size cap, showing that arbitrary power-of-two sandboxes are achievable at single-digit stores-only overhead.

SFI optimizations. Zeng et al. explored optimizing SFI guards with static analysis: combining control-flow integrity with range analysis to eliminate and hoist data-sandboxing checks [31], and Strato [32], a retargetable framework that applies such optimizations at the compiler's IR level. Zero-Cost [41] uses SafeStack [54] and CFI [56] to allow the sandboxed code to safely use the host stack. These optimizations lower overhead, but complicate the scheme and validator. LFI-XL instead forgoes such optimizations and employs only those that are simple, maintainable, and leave the scheme's invariants (and validator) unchanged (§5.5).

Memory protection keys. A large body of work builds in-process isolation directly on Intel MPK/PKU [20], [25], [60], [61] (or by re-purposing CET [21]), with several studies cataloging the difficulty of using these primitives safely [49], [63]. In these systems, MPK is the primary isolation boundary. LFI-XL instead treats MPK—and its AArch64 analogue POE—as an *accelerator* for a scheme that remains secure in software when the feature is unavailable (e.g., on older hardware), avoiding the hardware requirement that would otherwise limit deployability.

Eliminating sandbox bugs. Prior work has shown that building sandboxing tools without binary validators inevitably leads to bugs [39], [40], [77], [78]. Many SFI tools that follow best practices are thus designed for binary validation [4], [5], [8], [9], [18] (with the notable exception of WebAssembly [6]). While no system is perfect, attention to validation in SFI systems rules out design bugs by construction—bugs that can affect fine-grained memory-safety tools [79].

Hardware control flow. Intel CET [57] provides indirect-branch tracking and a shadow stack, while Arm offers Pointer Authentication [80] and the Guarded Control Stack [81]. Production engines already use these—V8 is adopting CET's shadow stack and has plans for forward-edge CFI [82], and WebKit on iOS uses PAC [83]. But PAC and coarse-grained CFI have known weaknesses allowing bypass [84], [85]. LFI-XL integrates CET into a complete SFI scheme with validation, using the shadow stack to avoid rewriting returns while still providing complete security guarantees.

Related hardware extensions. HFI [86] proposes new ISA extensions designed specifically to accelerate fault isolation, and CHERI [87] enforces memory safety with hardware

capabilities; both require non-commodity hardware. LFI-XL instead repurposes isolation features already available in commodity CPUs. Intel has also announced a hardware extension called APX [88] that adds x86-64 general-purpose registers, which may improve the performance of SFI schemes.

Sandboxing of JS engines. NaClJIT [69] and RockJIT [70] were the first efforts to isolate JavaScript engines with SFI: NaClJIT sandboxed V8 by compiling its runtime with NaCl and constraining the JIT to emit only validator-passing code (51–60% overhead on x86-64), while RockJIT enforced control-flow integrity and sandboxed only stores—an early stores-only design (14.6% overhead). Both date from an era when engines were smaller and a 4GiB sandbox was a reasonable fit, which is no longer true of modern engines (§1). Chrome's V8 sandbox (Ubercage) [89] uses a large (1TiB) masked region, but it is a bespoke mechanism customized to V8. In contrast, LFI-XL is designed as a general-purpose large-sandbox SFI scheme. Mohabi [71] investigated the application changes needed in browsers to sandbox JavaScript engines (including the approach to isolating JIT compilers used in our evaluation) and demonstrated their approach using a standalone SFI tool that supports directly rewriting x86-64 assembly to insert checks; in contrast, LFI-XL shows how such SFI tools (1) can be built as part of maintainable production compilers, (2) can support multiple mainstream architectures, and (3) can leverage hardware extensions.

9. Conclusion

SFI is a well-established way to contain memory-safety bugs, but modern tools fix sandboxes at 4GiB for performance, a self-imposed limit that shuts out large-footprint targets such as JavaScript engines and limits how many sandboxes a system can create. LFI-XL provides efficient AArch64 and x86-64 SFI at any power-of-two size, letting deployments choose sandbox sizes for their domain. In the stores-only mode, lifting the cap is cheap (1.0 point on AArch64 and 2.6–2.8 points on x86-64 over a 4GiB sandbox on SPEC 2017) and hardware (CET, MPK/POE) lowers overheads further as a portable optimization. LFI-XL also follows SFI best practices: minimal compiler changes, an unmodified ABI, and a small validator. It safely sandboxes complicated systems with hand-written assembly and JIT compilation.

Acknowledgments

We thank the reviewers for their insightful feedback. This work was supported in part by NSF grants #2120696, #2120642, #2327337, #2146755, and gifts from Mozilla, Qualcomm, and Stanford FDCI and IOG Research Hub. Zachary Yedidia is also a student researcher at Google.

References

[1] J. Vander Stoep and C. Zhang, "Queue the hardening enhancements," https://security.googleblog.com/2019/05/queue-hardening-enhancements.html, May 2019.

[2] The Chromium Projects, "Memory safety," https://www.chromium.org/Home/chromium-security/memory-safety/.

[3] R. Levick and S. Fernandez, "We need a safer systems programming language," https://www.microsoft.com/en-us/msrc/blog/2019/07/we-need-a-safer-systems-programming-language.

[4] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.

[5] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *SEC*. USENIX, 2010.

[6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," *Communications of the ACM*, vol. 61, no. 12, pp. 107–115, 2018.

[7] S. Narayan, T. Garfinkel, E. Johnson, Z. Yedidia, Y. Wang, A. Brown, A. Vahldiek-Oberwagner, M. LeMay, W. Huang, X. Wang *et al.*, "Segue & ColorGuard: Optimizing SFI performance and scalability on modern architectures," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 987–1002.

[8] Z. Yedidia, "Lightweight fault isolation: Practical, efficient, and secure software sandboxing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 649–665.

[9] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *15th USENIX Security Symposium (USENIX Security 06)*, 2006, pp. 209–224.

[10] L. Deng, Q. Zeng, and Y. Liu, "ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries," in *IFIP International Information Security and Privacy Conference*. Springer, 2015, pp. 386–400.

[11] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 699–716.

[12] B. Holley, "WebAssembly and back again: Fine-grained sandboxing in Firefox 95," https://hacks.mozilla.org/2021/12/webassembly-and-back-again-fine-grained-sandboxing-in-firefox-95/, Nov. 2021.

[13] S. Glazunov, "Project Zero - in-the-wild series: Chrome exploits," https://googleprojectzero.blogspot.com/2021/01/in-wild-series-chrome-exploits.html, Jan. 2021.

[14] Kaspersky, "Operation Triangulation," https://www.kaspersky.com/about/press-releases/connecting-the-dots-kaspersky-reveals-in-depth-insights-into-operation-triangulation, Oct. 2023.

[15] M. Stone, "2022 0-day in-the-wild exploitation…so far," https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html, 2022.

[16] P. Hickey, "Announcing Lucet: Fastly's native WebAssembly compiler and runtime," https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019.

[17] Bugzilla, "Bug 2009188," https://bugzilla.mozilla.org/show_bug.cgi?id=2009188.

[18] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[19] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: process separation for web sites within the browser," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, pp. 1661–1678.

[20] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.

[21] M. Xie, C. Wu, Y. Zhang, J. Xu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "CETIS: Retrofitting Intel CET for generic and efficient intra-process memory isolation," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2989–3002.

[22] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "ARMlock: Hardware-based fault isolation for ARM," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 558–569. [Online]. Available: https://doi.org/10.1145/2660267.2660344

[23] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process isolation for High-Throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504. [Online]. Available: http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[24] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys–efficient In-Process isolation for RISC-V and x86," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1677–1694.

[25] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: software abstraction for Intel memory protection keys (Intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, ser. USENIX ATC '19. USA: USENIX Association, 2019, pp. 241–254.

[26] Mozilla, "SpiderMonkey — Firefox source docs," https://firefox-source-docs.mozilla.org/js/index.html.

[27] J. de Mooij, M. Gaudet, I. Ireland, N. Henderson, and J. N. Amaral, "CacheIR: The benefits of a structured representation for inline caches," in *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2023, pp. 34–46.

[28] "JetStream 2.2 JavaScript and WebAssembly benchmark," https://browserbench.org/JetStream2.2/.

[29] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[30] C. Small and M. Seltzer, "MiSFIT: Constructing safe extensible systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 34–41, 1998.

[31] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *CCS*. ACM, 2011.

[32] B. Zeng, G. Tan, and Ú. Erlingsson, "Strato: A retargetable framework for Low-Level Inlined-Reference monitors," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 369–382.

[33] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 75–88.

[34] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and language-independent mobile programs," in *PLDI*. ACM, 1996.

[35] Google Chrome Team, "(P)NaCl Deprecation Announcements," Online: https://developer.chrome.com/docs/native-client/migration, Jun. 2017.

[36] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *USENIX ATC 19*, 2019, pp. 107–120.

[37] G. Tan, "Principles and implementation techniques of software-based fault isolation," *Foundations and Trends in Privacy and Security*, vol. 1, no. 3, pp. 137–198, Oct. 2017.

[38] Z. Yedidia, G. Ramseyer, and D. Mazières, "Deterministic client: Enforcing determinism on untrusted machine code," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 633–649. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/yedidia

[39] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: better, faster, stronger SFI for the x86," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 395–404.

[40] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Доверяй, но проверяй: SFI safety for native-compiled Wasm," in *NDSS*. Internet Society, 2021.

[41] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan, "Isolation without taxation: near-zero-cost transitions for WebAssembly and SFI," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–30, 2022.

[42] M. Seaborn, "Sandboxing libraries in Chrome using SFI: zlib proof-of-concept," https://docs.google.com/presentation/d/1RD3bxsBfTZOIfrlq7HzGMsygPHgb61A1eTdelIYOurs/, 2013.

[43] Bytecode Alliance, "Wasmtime," https://wasmtime.dev/.

[44] GitHub, "wasm2c," https://github.com/WebAssembly/wabt/tree/main/wasm2c.

[45] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: WebAssembly as a practical path to library sandboxing," 2019. [Online]. Available: https://arxiv.org/abs/1912.02285

[46] GitHub, "WebAssembly Micro Runtime," https://github.com/bytecodealliance/wasm-micro-runtime.

[47] Valve, "Steam hardware & software survey: July 2026," https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam?platform=pc, Jul. 2026, "BMI2" under "Other Settings(Windows)".

[48] dominickd, "Auto-sandboxing assembler for Native Client," https://docs.google.com/document/d/1MDZDfFv_g4dPEglzPTudTVt-w7e3q1oE1zCUOLwGIQ4/edit.

[49] E. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1409–1426.

[50] I. Sheludko and S. Aboy Solanes, "Pointer compression in V8," https://v8.dev/blog/pointer-compression, Mar. 2020.

[51] M. Larabel, "Linux developers looking at retiring the x32 ABI," https://www.phoronix.com/news/Linux-x32-ABI-2026, 2026.

[52] G. C. Necula and P. Lee, "Research on proof-carrying code for untrusted-code security," in *S&P*. IEEE, 1997.

[53] C. Fallin, "Verification of addressing / memory accesses with VeriWasm or similar," https://github.com/bytecodealliance/wasmtime/issues/6090, Mar. 2023.

[54] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[55] MaskRay, "LLVM integrated assembler: Engineering better fragments," https://maskray.me/blog/2025-07-27-llvm-integrated-assembler-engineering-better-fragments.

[56] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *TISSEC*, 2009.

[57] Intel Corporation, "A technical look at Intel® Control-Flow Enforcement Technology," https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html, 2020.

[58] "Intel® 64 and IA-32 architectures software developer's manual," 2020.

[59] "Arm architecture reference manual for A-profile architecture: Permission Overlay Extension," https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions.

[60] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 936–952.

[61] B. Im, F. Yang, C.-C. Tsai, M. LeMay, A. Vahldiek-Oberwagner, and N. Dautenhahn, "The Endokernel: Fast, secure, and programmable subprocess virtualization," *arXiv preprint arXiv:2108.03705*, 2021.

[62] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening WebAssembly against Spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1433–1450.

[63] X. Chen, Z. Li, T. Jain, V. Narayanan, and A. Burtsev, "Limitations and opportunities of modern hardware isolation mechanisms," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 349–368.

[64] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[65] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[66] A. Rossberg, "Memory64 proposal for WebAssembly," https://github.com/WebAssembly/memory64, 2020.

[67] WebAssembly, "WASI SDK," https://github.com/WebAssembly/wasi-sdk#notable-limitations.

[68] B. Smith, "Wasm Memory64: Bounds-checking strategies," https://github.com/WebAssembly/memory64/issues/3, Apr. 2020.

[69] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *PLDI*. ACM, 2011.

[70] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *CCS*. ACM, 2014.

[71] A. Sharma, A. Balaji, Z. Yedidia, A. Du, T. Noh, I. Ireland, J. de Mooij, M. Gaudet, T. Garfinkel, D. Stefan *et al.*, "Mohabi: Disaggregating and sandboxing the Firefox JavaScript engine," in *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, 2026, pp. 207–228.

[72] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "RocksDB: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3483840

[73] OpenBenchmarking.org, "RocksDB Benchmark: pts/rocksdb-1.7.0," https://openbenchmarking.org/test/pts/rocksdb, 2025.

[74] Mozilla, "Firefox source: third_party/rlbox_wasm2c_sandbox/src/wasm2c_rt_mem.c," https://github.com/mozilla-firefox/firefox/blame/9b794146973d3e99d273c58f9f6a5cc1dcfc09cb/third_party/rlbox_wasm2c_sandbox/src/wasm2c_rt_mem.c#L75.

[75] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl: Portable Native Client executables," *Google White Paper*, 2010.

[76] WebAssembly, "Custom page sizes," https://github.com/WebAssembly/custom-page-sizes/blob/main/proposals/custom-page-sizes/Overview.md, 2026.

[77] J. A. Kroll and D. Dean, "BakerSFIeld: Bringing software fault isolation to x64," SRI Computer Science Laboratory, Tech. Rep., 2009. [Online]. Available: https://www.jkroll.com/papers/bakersfield_sfi.pdf

[78] H. Bridge and B. Chen, "Native Client security contest: The results are in!" https://developers.googleblog.com/en/native-client-security-contest-the-results-are-in/, Jul. 2009.

[79] A. Jacobs and S. Volckaert, "Not quite write: On the effectiveness of Store-Only bounds checking," in *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, 2024, pp. 171–187.

[80] "Pointer Authentication on Arm," https://learn.arm.com/learning-paths/servers-and-cloud-computing/pac/pac.

[81] "The Guarded Control Stack," https://developer.arm.com/documentation/ddi0487/mb/-Part-D-The-AArch64-System-Level-Architecture/-Chapter-D11-The-Guarded-Control-Stack.

[82] Google Chrome, "V8 CFI plans," https://docs.google.com/document/d/1O2jwK4dxI3nRcOJuPYkonhTkNQfbmwdvxQMyXgeaRHo, May 2022.

[83] B. Azad, "Examining pointer authentication on the iPhone XS," https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.

[84] S. Groß, "PAC and JIT hardening bypass," https://project-zero.issues.chromium.org/issues/42451144, 2020.

[85] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow bending: On the effectiveness of Control-Flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[86] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita *et al.*, "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*, 2023, pp. 266–281.

[87] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[88] Intel, "Introducing Intel® Advanced Performance Extensions (Intel® APX)," https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html.

[89] S. Groß, "V8 Sandbox - high-level design doc," https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8, Jul. 2021.

Appendix A. Ethics Considerations

Our work focuses on understanding, developing, and improving the performance of sandboxing (SFI) compiler toolchains, including those that leverage hardware extensions like Intel MPK and CET. We performed all experiments on local systems and did not use any systems with personal data.

While MPK and CET are technologies for enforcing security, our analysis was limited to analyzing their performance, and did not focus on finding (or find) any security-related bugs that required reporting. Our work also did not involve human subjects or live systems.

The following lists entities that are possible stakeholders in this paper.

- This paper's authors and author-affiliated institutions.
- CPU vendors like Intel, AMD, and Arm who ship hardware extensions that can be used by SFI toolchains.
- Any customers of Intel, AMD, and Arm chips interested in the quality of SFI tools or hardware security extensions in these platforms.
- Maintainers of upstream compiler toolchains who may want/have to implement SFI-style features in the future.
- Authors of the most related prior work, whose work we have cited, reproduced, or extended.

A.1. Potential Harms

1) Compiler vendors who decide to implement SFI support may take on new attack surface, increase maintenance requirements (taking developer time away from maintenance of other features in compilers), or increase the complexity of compiler toolchains.
2) Since we show that SFI tools have overheads in some implementations and workloads, even when using hardware extensions, our results could be (mis)interpreted in several scenarios. Any such misunderstanding, while unlikely, would mean that an important security tool is not available to customers, reducing the security of applications.
   - ► Software vendors may delay adopting SFI in their products, which can result in software being insecure and attackable for longer. This may be exacerbated if they incorrectly assume that the worse performance overheads apply to them without actually measuring their software.
   - ► Software vendors may also inadvertently assume the opposite—that their applications will have low SFI overheads as shown in some of our workloads—and choose to deploy it without careful analysis.
   - ► Intel/Arm may decide not to deploy security extensions that could improve SFI, if they believe that hardware extensions do not eliminate all the overhead. This would mean that hardware-assisted SFI is less likely to get deployed in practice.
   - ► Users may choose to disable SFI in applications (if an option is exposed via settings) if they worry that SFI will make their applications slow.

A.2. Mitigations

While the above scenarios are possible, we have taken great care to:

1) Outline how SFI tools can be implemented with minimal TCB and minimal maintenance requirements, so that compilers that adopt this approach will be able to offer the feature safely without ballooning the maintenance costs.
2) Articulate the security benefits and performance trade-offs of using sandboxing as well as the context in which these results apply to minimize the probability of misinterpretation. Our research, especially our observations on how to design performant SFI tools and our methodology for evaluating SFI tools, provides a valuable resource. The positive outcome of this research would be hardware and software vendors working together to improve the quality of SFI tools.

We believe our work clarifies the state of SFI, as well as the path forward to improve performance (and thus adoption in applications and security of end-user devices), by offering a clearer understanding of the costs and benefits. On balance, we believe the net benefits to the community outweigh the potential but low-probability risks of high maintenance burden or misinterpretation slowing down adoption.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper proposes a design for Software Fault Isolation (SFI) called LFI-XL that makes the sizes of the sandboxes more flexible: the sandbox sizes can be of size 2^N rather than 4GiB. LFI-XL includes support for store-only sandboxing, which is significantly faster than its read/write sandboxing mode. LFI-XL is implemented and evaluated for both x86-64 and AArch64, has extensions for different ISA extensions, but does not depend on them.

B.2. Scientific Contributions

- ► Creates a New Tool to Enable Future Science
- ► Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) This paper provides a valuable step forward in an established field. Software fault isolation (SFI) is a well-known technique for confining memory-safety bugs, but modern SFI schemes limit the size of sandboxes to 4GiB to achieve low overhead in 32-bit addressing mode. Further, the original SFI targets MIPS/Alpha with a simple SFI scheme. This paper introduces LFI-XL, which includes new SFI optimizations for addressing modes and CISC-support and support flexible sized sandboxes. This paper shows how the different choices and trade-offs in the design and implementation, why the specific choices work well together in the final design, and that variable-sized sandboxes can be practical for complex systems.
2) This paper creates a new tool to enable future science. This paper designs a variable-size SFI scheme for modern architectures (AArch64, x86-64) that can take advantage of extensions like MPK/POE and CET. This paper provides an implementation in an easy-to-maintain compiler toolchain.