Automated Formal Verification of a Software Fault Isolation System

Matthew Sotoudeh*, Zachary Yedidia*

*Stanford University, Stanford, USA
{sotoudeh, zyedidia}@stanford.edu

***Abstract*—Software fault isolation (SFI) is a popular way to sandbox untrusted software. A key component of SFI is the *verifier* that checks the untrusted code is written in a subset of the machine language that guarantees it never reads or writes outside of a region of memory dedicated to the sandbox. Soundness bugs in the SFI verifier would break the SFI security model and allow the supposedly sandboxed code to read protected memory. In this paper, we address the concern of SFI verifier bugs by performing an automated formal verification of a recent SFI system called Lightweight Fault Isolation (LFI). In particular, we formally verify that programs accepted by the LFI verifier never read or write to memory outside of a designated sandbox region.**

***Index Terms*—Verification, software fault isolation**

I. Introduction

Computer systems frequently need to run untrusted code, such as web browsers running applets and scripts, software linking against third-party libraries, or operating systems accepting user code into the kernel for packet filtering. In all of these cases, security concerns demand that the host application only run the untrusted code in ways that guarantee the untrusted code will not read from or write to the host application's memory. In theory, the untrusted code could be run in an entirely separate address space (process) from the host code. In practice, processes introduce far too much context switching overhead to be reasonable for many applications. Requiring the untrusted code to be written in a memory-safe language like Java, JavaScript, EBPF, or WebAssembly that can be executed in the same address space improves startup time but is significantly slower than native machine code. Furthermore, it makes the entire language interpreter or compiler a security-critical part of the trusted codebase.

This paper focuses on the *software fault isolation* (SFI) mechanism of software sandboxing. In SFI, the untrusted code is compiled to a memory-safe subset of the machine code (ARM64). That binary is only executed if it passes a *SFI verifier*, which checks that all memory access instructions in the binary target memory within the sandbox. Of course, verifying arbitrary machine code is hard; in practice, SFI techniques require that the guest code be compiled to use a restricted subset of memory operations that are easy to verify (a naïve SFI scheme, for example, might require every memory read or write be immediately preceded by a bounds check).

While SFI is a popular technique with many different instantiations [1]–[3] we focus on the recent Lightweight Fault Isolation (LFI) system [3]. LFI is a very low-overhead sandboxing system specialized for ARM64. There is ongoing work to bring LFI to production use-cases, and large-scale deployment motivates extra scrutiny on the LFI verifier, which is tasked with the security-critical job of verifying that a given binary cannot access memory outside of its dedicated sandbox.

We discuss how the LFI verifier accomplishes this in Section II, but briefly, it involves a whitelisted subset of ARM64 instructions that are chosen to maintain certain memory isolation guarantees. However, the ARM64 includes a very large number of instructions, and it is conceivable that those in the whitelist interact in nontrivial ways to break out of the sandbox. Indeed, earlier versions of LFI have had soundness bugs because we did not realize that certain whitelisted instructions could modify registers in unexpected ways.

This paper describes our work on **automatic formal verification of the lightweight fault isolation SFI system** for the ARM64 architecture; the LFI verifier is now equipped with a formal proof that every program it accepts cannot read outside of its dedicated sandbox memory region.

Our heavy automation is unique among existing formally verified SFI systems [2]. The only manual intervention needed is to write down an *SFI invariant*, which requires only about 20 lines of SMT-LIB2 code. To enable this, we codesigned the SFI system and verification pipeline. Traditional SFI systems frequently check for the presence or absence of *sequences* of instructions and sometimes do complicated dataflow analyses to ensure memory isolation. But these nontrivial analyses are very difficult to formally prove precise facts about. Instead, we realized that the LFI scheme is essentially a *stateless* SFI system, i.e., its verifier is nothing more than an instruction whitelist. The guest program may contain any instruction in the whitelist, in any order, and no other instructions. This means we need merely check that each whitelisted instruction, on its own, maintains the expected SFI invariants.

The rest of this paper summarizes the LFI system (Section II), describes our verification pipeline (Section III), and summarizes future and related work (Sections IV and V). An archival version of our source code and tools is available in [4].

II. Background on Lightweight Fault Isolation

This section briefly describes the LFI system for software fault isolation. Consider a setting where a web browser downloads and wants to execute untrusted machine code from the Internet (similar to the NaCl model used by Chromium). In this setting, the website operator providing the machine code will compile it using an untrusted, but LFI-aware compiler into ARM64 binary code. This binary code is then sent to the browser, who wishes to run it while having some guarantee

https://doi.org/

that it will not perform unauthorized memory operations. To do so, the browser provides the ARM64 binary to the *LFI Verifier*, which is essentially a sound program analyzer for memory isolation. If the verifier rejects the program, it might be unsafe and so the browser refuses to run it. On the other hand, if the verifier accepts the program, it is guaranteed to be memory isolated and so the browser can confidently run it. The program might make calls into the LFI runtime to request actions like drawing to the screen or sending network packets.

Safety relies on two crucial aspects: the LFI verifier and the LFI runtime. A buggy verifier might accept code that reads from the host computer's memory in insecure ways, while a buggy runtime might allow the binary to manipulate it into performing unpermitted operations.

This paper considers only the LFI verifier, and so we assume the LFI runtime is implemented correctly. The notion of memory isolation here is more coarse grained than memory safety in traditional programming languages: the untrusted program may access any part of a 4GiB 'sandbox' devoted to it. It can do anything in that sandbox; it might interpret portions of it as arrays and then index those application-interpreted arrays out-of-bounds, as long as it never successfully[1] reads or writes outside of its 4GiB sandbox.

The rest of this section briefly describes how the LFI verifier checks memory dereferences in its simpler sparse mode; complete details on the LFI system are available in [3].

A. Reserved Registers for Memory Isolation

The job of the LFI verifier (checking the given code is memory isolated) is, in full generality, undecidable. Hence, it must check instead for a subset of known-to-be-memory-safe constructs (sound but incomplete). The set of constructs it checks for must allow efficient compiled code.

The primary trick is *reserved registers*. There is a reserved *sandbox base register* (`x21` in LFI) pointing to the first byte of the program's 4GiB sandbox region. This region is assumed to be 4GiB aligned, so the bottom 32 bits of the `x21` register are zero. There is also a reserved *addressing register* (`x18` in LFI) which the verifier ensures is the only register (with some exceptions) used for the ARM64 load and store instructions. To a first approximation, the LFI verifier iterates over the binary, disassembles each instruction, and checks:

- No instruction ever writes to `x21`, so `x21` will always contain the sandbox base address.
- The only instructions that can write to `x18` are those of the form `add x18, x21, wN, uxtw`, which results in `x18` containing the high 32 bits of register `x21` concatenated to the low 32 bits of register `xN`.
- The only instructions that can load from or store to memory use `x18` as the memory address.

[1]Notably our verification only shows that the program never *successfully* reads or writes outside of the sandbox; valid LFI programs can sometimes make *attempted* reads or writes as long as those attempts are guaranteed to terminate execution, e.g., they access a page that the runtime guarantees is unmapped. This is discussed further in Section II-B.

A program that passes this check is thus unable to explicitly load memory outside of the 4GiB sandbox region, because it can only load from `x18` and `x18` can only be set by an operation that ensures its final value is within the sandbox region (i.e., has the same upper 32 bits as `x21`).

Note that, beyond this simple approximation, there are many corner cases that need to be handled [3]. For example, `x18` might be set to the very last byte in the 4GiB sandbox region, but then doing an 8 byte load from that address will actually read outside of the 4GiB sandbox region. To resolve this, the LFI specification [5] requires that the sandbox be surrounded by unmapped 'redzone regions' so that such corner case reads cause a trap rather than reading host memory. Use of the stack pointer register presents other complications, described below.

B. Handling the Stack

In addition to direct memory loads via the `x18` register, LFI also allows loads relative to the stack pointer (`sp`). This introduces extra complications as *writeback loads* are used to simulate a stack push. For example, the instruction

```
// sp = sandbox_base
ldr x0, [sp], #-8
// sp = sandbox_base - 8
```

dereferences the stack pointer and then decrements it by 8 bytes. But if the stack pointer is initially pointing at the base of the sandbox region, after executing this instruction it will now be pointing outside of the sandbox region. To resolve this, the LFI runtime guarantees that 4GiB of address space preceding and following the sandbox region is unmapped, so attempting to actually access this region would cause a trap and hence safe exit of the guest program (the largest offset accepted by the writeback load operation is smaller than 4GiB).

The reader might be concerned that a chain of such instructions might leave `sp` past even this 4GiB buffer region and hence possibly pointing at host memory:

```
// sp = sandbox_base
ldr x0, [sp], #-8
// sp = sandbox_base - 8
ldr x0, [sp], #-8
// sp = sandbox_base - 16
ldr x0, [sp], #-8
...
// sp = sandbox_base - 4GiB - 8?
ldr x0, [sp], #-8 // load bad data?
```

But indeed this is not possible because the second operation in this sequence would trap (accessing unmapped memory), transfer control back to the LFI runtime, and thus terminate the execution of the sandboxed code.

C. Compiler Instrumentation to Generate LFI-Passing Code

Compilers rarely produce code passing the LFI verifier, e.g., compiled code often loads from a register other than `x18` or writes to `x18` using a nonwhitelisted instruction. Hence, the LFI project includes a modified compiler toolchain that

produces LFI-passing binary. To do so, it tells the compiler to reserve registers `x18` and `x21`. Then, after the compiler produces an assembly file, we identify instructions that would fail LFI verification and rewrite them into sequences of instructions that pass LFI verification. For example, loading from `x5` may not be safe because the LFI verifier has no way of knowing that it points into the sandbox region:

```
ldur x2, [x5]
```

Instead, the rewriter might transform this into the two-operation sequence:

```
add x18, x21, w5, uxtw
ldur x2, [x18]
```

Which replaces the upper 32 bits of `x5` with those of `x21` before the load. If `x5` was in-bounds at this point in the program execution, then this transformation has no effect. But if `x5` was out-of-bounds, this transformation causes it to load an in-bounds address instead. This may lead to an application-level bug in the untrusted code, but effectively prevents it from accessing the host memory.

III. Verification of LFI

This section describes how we verify correctness of the LFI system. Recall that the LFI system is a whitelist of allowed instructions and a set of memory mappings guaranteed by the runtime (e.g., that the page following the sandbox region is trap-on-execute). There are actually two variants of LFI that we verify: sparse LFI (where every sandbox is surrounded by 4GiB redzone regions of unmapped memory) and dense LFI (where sandboxes are packed more tightly, with internal unmapped regions). In this section we focus on verification of sparse LFI; dense LFI is verified similarly. In general, we verify the following:

> If the registers satisfy a certain *SFI invariant* I, and the next instruction to be executed is one of the whitelisted instructions, then when executing the instruction both (1) no memory operations (reads, writes, executes) outside of the 4GiB sandbox succeed; and (2) either the execution terminates or the SFI invariant I holds on the successor state as well.

This is verified per instruction: so as long as the guest code starts in a state satisfying the SFI invariant I and all of the instructions in the sandbox region satisfy that property, it will continually satisfy that invariant during the execution of the guest program and not make any unsafe memory operations until it terminates. Execution can terminate by executing an instruction that causes a hardware trap (like reading from memory that is unmapped or executing an undefined instruction) or by calling in to the LFI runtime.

A. Register Invariants

In the sparse LFI variant, we verify the following invariants:

- `x21` points to the special 4GiB-aligned sandbox base address, which does not change during execution.
- `x18` and the stack pointer are both in the range $[\texttt{x21} - 128\texttt{MiB}, \texttt{x21} + 4\texttt{GiB} + 128\texttt{MiB})$.[2]
- The program counter (PC) is in the range $[\texttt{x21}, \texttt{x21} + 4\texttt{GiB})$.
- R_{30} is either in the range $[\texttt{x21}, \texttt{x21} + 4\texttt{GiB})$, or contains one of three special LFI runtime call addresses.

Note the second of these is to handle the case of writeback load operations as described in Section II-B.

B. Detecting Bad Side Effects

The main goal of our verification is to ensure that the guest never breaks out of its sandbox. We record any (symbolic) memory reads or writes performed during the (symbolic) execution of the instruction. We say a bad side effect occurs if any of them successfully accesses an address outside of the 4GiB sandbox region (note that accesses guaranteed to fail are not successful; e.g., reading from a page that the runtime guarantees is unmapped terminates execution of the sandboxed program). An instruction fails verification if it can ever perform a bad side effect when starting in a CPU state that satisfies the invariants and expected memory mappings.

C. Memory Mapping Assumptions and Detecting Faults

Per the LFI specification, we assume that the LFI runtime has set up the following memory mappings:

- The 4GiB regions before and after the sandbox 4GiB region are both unmapped, so accesses to them trap.
- The first 4KiB of the sandbox region is marked read-only, and the first three words at that location contain the addresses of special LFI runtime calls (analogous to system calls).

D. Handling Faults

We need to detect when the program execution terminates. We call this a 'fault,' and it can occur in one of a few ways:

- The PC (before or after the instruction) is not 4 byte aligned or is in a region of memory that we know is mapped no-execute.
- A memory read or write occurs to an address that we know is unmapped.
- The PC becomes one of the three special trampoline addresses.
- An explicit trap occurs, e.g., an undefined instruction.

Some nuance arises here because, to the best of our knowledge, ARM does not always make strict guarantees about the order of execution of traps. Consider a write to a memory range crossing two bytes, b and $b{+}1$. If b is unmapped memory but $b + 1$ is host memory outside of the sandbox region, the operation is safe only if the trap is guaranteed to occur *before* the $b + 1$ write happens, i.e., if the hardware bounds checks 'eagerly.' Rather than assume that the ordering of these checks is guaranteed or deterministic across microarchitectures, we check a much stronger property: bad side effects do not occur *even if the traps always happen last.*

[2] In dense mode, this is $+/-\ 8\texttt{KiB}$.

E. Implementation and Formal Model of ARM64

Critical to our approach is the ability to, given an instruction encoding and a desired logical property about the CPU state before and after execution of the instruction, check whether that property holds. This requires a formal model of the CPU semantics. For this, we relied on the ASL formal specification of ARM64 [6]. In particular, we used the ASLP interpreter [7] which can partially evaluate the ASL specification and returns a symbolic intermediate representation that summarizes the effects of the instruction on a symbolic model of the CPU.

We wrote a Python library that extracts this symbolic intermediate representation from the ASLP interpreter into a usable Python form. We then wrote a symbolic execution engine for this intermediate representation. The output of the symbolic execution is a series of SMT-LIB2 assertions encoding the execution of the instruction. After adding assertions checking the SFI invariant, we use the `Yices2` SMT solver to determine satisfiability [8].

Writing the symbolic execution engine was simpler than it sounds, because we only need it to be sound, not complete. In particular, because so few of the operations actually affect the reserved registers, we can leave many functions uninterpreted.

We ran into two issues with the ASL specification and ASLP partial evaluator that required extra work. First, the ASL spec and ASLP partial evaluator lacked proper support for expressing atomic instructions, which led to internal ASLP exceptions when partially evaluating those instructions. To solve this, we commented out the atomics-related ASL code that was causing the issues, as our verification is not interested in race conditions. Second, the ASL spec's handling of system access traps returns a special record type which also causes internal ASLP errors. We resolved this by replacing the system access trap generation functions with an assignment to a special `SYMEX_DID_TRAP` variable that indicates to our symbolic execution engine directly when a trap occurs.

F. Parallel Verification and Results

Our verification technique involves iterating through all whitelisted ARM64 instructions. To do so, we first iterate through all of the ARM64 instruction set (2^{32} total instructions) and check with the stateless LFI verifier for which ones were accepted. We evenly divide those accepted instructions across all of the verifier worker threads.

We ran verification in parallel on a 160-core Ampere Altra ARM64 machine. Verification for each of the sparse and dense LFI systems required approximately 20 hours each. Throughout the verification process we identified some inaccuracies and missing details in the LFI specification [5], including how LFI runtime calls are handled and memory mapping guarantees for the first and last sandboxes. We are in the process of correcting these oversights in a future version of the specification, but these are only missing details in the written specification. After clarifying these specification issues, the verification process completed and found no bugs.

IV. Limitations and Future Work

This section describes possible future work. First, we could use our verification pipeline to automatically synthesize SFI systems from their invariants by iterating over all 2^{32} ARM64 instructions and including only those that maintain the invariants. This could enable rapid experimentation with SFI systems, especially to different domains like determinism.

Second, we could try to more fully automate our verification pipeline by using invariant inference techniques to automatically derive the invariant for a given SFI system.

To speed up our verification pipeline we could also check groups of instructions at once, e.g., all add instructions targeting a nonreserved register. This would require the ability to extract shared semantics for groups of instructions or instruction encodings with symbolic bits. In theory the ASLP interpreter supports this [7] but we decided to use the simpler strategy for this paper and found performance was reasonable.

This work did not consider safety of the LFI runtime, which remains an unverified but trusted component of the system.

In this work we focused on verifying an SFI system for the ARM64 architecture. But LFI also supports the x86-64 architecture, and there has been recent work attempting to extend it to RISC-V. It would be interesting to produce a fully verified SFI system for x86-64 or RISC-V. We believe that the verification strategy in this paper would work well for verifying the RISC-V version of LFI, which involves a similarly sized set of whitelisted instructions and is also effectively stateless. However, making the x86-64 LFI system stateless requires recognizing a number of 'macro-operations' that are actually sequences of other instructions (similar to `nacljump` in the original NaCL paper [1]). This leads to the set of whitelisted (macro)instructions for x86-64 LFI being much larger than the set for ARM64 and RISC-V, making it infeasible to enumerate through all of them. In theory, as described earlier, a verification pipeline that lets us leave some bits (e.g., immediate values) symbolic would make verifying x86-64 LFI feasible as well.

V. Related Work

a) Software Fault Isolation: Pittsfield was an early SFI system [9]. The Native Client (NaCl) SFI system was shipped with Google Chrome and used for website-loaded add-ons such as required (at the time) by Netflix [1]. This shows the security critical position that SFI systems play, motivating the use of formal verification to ensure the absence of bugs. More recently, the LFI system has been proposed as an optimized SFI system for ARM64.

To our knowledge, ours is the first stateless SFI system. Native Client, for example, required the use of a special two-instruction 'nacljump' sequence of instructions to perform indirect jumps while maintaining control flow integrity. Automated verification of such a stateful SFI verifier would be more complicated, because we can no longer look at each instruction on its own, leading to an impractically large encoding space of instructions to automatically iterate over. Similarly, LFI as proposed in [3] allows some within-basic-block optimizations,

particuarly removing guards to stack pointer modification when a stack access is performed later in the basic block. Since then, we have found those optimizations can be removed with minimal performance impact, making it fully stateless.

b) Formal Models of ISAs: This work used the ARM64 formal specification extracted by the ASLP [7] partial evaluator. [10] and [11] are other formal specifications of hardware ISAs. At the beginning of this project we tried to use the SAIL [11] architecture semantics. In theory, using SAIL would have made it easier to extend our technique to architectures beyond ARM64, since SAIL contains models of multiple different architectures. However, we found that the SAIL models simulate the architecture at a lower level than we needed. Furthermore, because the SAIL models were so fine-grained, they required significantly more investment for us to understand the model and even significantly more time to just run the compilers necessary to work with the model. Because our verification technique involves running a symbolic execution of the architecture semantics for every one of the LFI-whitelisted instructions, we decided to instead go for the ASLP semantics which were both simpler for us to work with and had faster tooling.

c) Verified Sandboxing: While we believe ours is the first *automatically* verified SFI system, there have been formally verified sandboxing systems before. VeriWasm [2] has an interactively verified SFI system used as a WebAssembly target. vWasm [12] is a verified compiler from WebAssembly to x86-64 machine code written in F*. WAVE [13] verifies a WASM runtime. RockSalt [14] and ARMor [15] use interactive theorem provers to verify classic SFI verifiers.

d) Verified Program Analyses: The LFI verifier is itself essentially a program analyzer. The problem of verifying program analyzers has been explored in prior work, such as ALIVE [16] that verifiers program analyzers and transformations in a domain-specific language, and ALIVE2 [17] that does bounded translation validation for LLVM optimizations.

Acknowledgements

We thank the National Science Foundation (DGE-1656518, DGE-2146755), Stanford IOG Research Hub, and Stanford Future of Digital Currency Initiative for supporting grants.

References

[1] 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," in *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society, 2009, pp. 79–93. [Online]. Available: https://doi.org/10.1109/SP.2009.25

[2] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "SFI safety for native-compiled wasm," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[3] 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, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds. ACM, 2024, pp. 649–665. [Online]. Available: https://doi.org/10.1145/3620665.3640408

[4] M. Sotoudeh and Z. Yedidia, "(artifact) automated formal verification of a software fault isolation system," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16883694

[5] "LFI specification (draft)." [Online]. Available: https://www.scs.stanford.edu/~zyedidia/docs/lfi/lfi-spec.pdf

[6] A. Reid, "Trustworthy specifications of arm® v8-a and v8-m system level architecture," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 161–168. [Online]. Available: https://doi.org/10.1109/FMCAD.2016.7886675

[7] K. Lam and N. Coughlin, "Lift-off: Trustworthy armv8 semantics from formal specifications," in *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, A. Nadel and K. Y. Rozier, Eds. IEEE, 2023, pp. 274–283. [Online]. Available: https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_36

[8] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49

[9] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*, A. D. Keromytis, Ed. USENIX Association, 2006. [Online]. Available: https://www.usenix.org/conference/15th-usenix-security-symposium/evaluating-sfi-cisc-architecture

[10] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 1133–1148. [Online]. Available: https://doi.org/10.1145/3314221.3314601

[11] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for armv8-a, risc-v, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 71:1–71:31, 2019. [Online]. Available: https://doi.org/10.1145/3290384

[12] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-safe multilingual software sandboxing using WebAssembly," in *Proceedings of the USENIX Security Symposium*, August 2022.

[13] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "Wave: a verifiably secure webassembly sandboxing runtime," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2940–2955. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179357

[14] 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*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 395–404. [Online]. Available: https://doi.org/10.1145/2254064.2254111

[15] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: fully verified software fault isolation," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 289–298. [Online]. Available: https://doi.org/10.1145/2038642.2038687

[16] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with alive," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. M. Blackburn, Eds. ACM, 2015, pp. 22–32. [Online]. Available: https://doi.org/10.1145/2737924.2737965

[17] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr, "Alive2: bounded translation validation for LLVM," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 65–79. [Online]. Available: https://doi.org/10.1145/3453483.3454030