UNIVERSITY OF CALIFORNIA SAN DIEGO

Retrofitting fast and secure sandboxing in real systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Shravan Ravi Narayan

Committee in charge:

Professor Deian Stefan, Chair
Professor Sorin Lerner
Professor Stefan Savage
Professor Geoff Voelker
Professor Xinyu Zhang

2022

The dissertation of Shravan Ravi Narayan is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not be possible without my amazing collaborators who worked with me tirelessly with research, evaluations, and deployments. In alphabetical order, sincere thanks to Anjo Vahldiek-Oberwagner, Conrad Watt, Craig Disselkoen, Daniel Moghimi, David Thien, Dean Tullsen, Deepak Garg, Deian Stefan, Evan Johnson, Eric Rahm, Fraser Brown, Hovav Shacham, Joey Rudek, Kazem Taram, Matthew Kolosick, Michael LeMay, Nathan Froyd, Ranjit Jhala, Ravi Sahita, Salmin Sultana, Sorin Lerner, Sunjay Cauligi, Stefan Savage, Tal Garfinkel, Tyler McMullen, Yousef Alhessi, and Zhao Gang, who not only collaborated with me on numerous research projects, but put up with my long brainstorming sessions and sometimes unusual work schedule.

Equally important were my incredible friends, colleagues, and labmates: Dhiman Sengupta, Janet Johnson, Mario Alvarez, Nishant Bhaskar, Craig Disselkoen, Anish Tondwalkar, John Renner, Alex Sanchez-Stern, Daniel Moghimi, Tal Garfinkel, John Sarracino, Dimitar Bounov. In addition to helping me with my research, they also took on an equally important task — dragging me away from my research when it was clearly necessary.

I am incredibly thankful to my advisor, Deian Stefan for the countless hours of help over the years in helping building my technical, writing, presentation, and overall research skills. I am very thankful to Hovav Shacham who, while not my advisor on paper, went out of his way so many times over the years to advise and guide my research. I also owe a deep debt of gratitude and thanks to Fraser Brown, who assumed the role of "senior PhD student" to perfection, serving as both a role model and mentor on not only my research, writing, and talks, but also my post PhD career.

And last but not the least, a heartfelt thanks to my parents KGY Narayan and Janaki Narayan who at every step of my PhD have unhesitantly encouraged and supported me through this long process; my sister Sneha Narayan who not only showed me what a successful academic career looks like, but also helped me build my own academic path; and my wife Sujanya

Ravishankar who has supported me through my academic path, all while living on the opposite coast, and pursuing her own PhD.

Dissertation contents. This dissertation borrows material from numerous academic works from the dissertation author. The Introduction, in part, uses material from all works listed below.

Chapter 2, in full, is a reprint of the material as it appears in the USENIX Security Symposium 2020. Narayan, Shravan; Disselkoen, Craig; Garfinkel, Tal; Froyd, Nathan; Rahm, Eric; Lerner, Sorin; Shacham, Hovav; Stefan, Deian. USENIX, 2020. The dissertation author was the primary investigator and author of this material.

Chapter 3, contains material reprinted from the Network and Distributed System Security Symposium (NDSS) 2021. Johnson, Evan; Alhessi, Yousef; Thien, David; Narayan, Shravan; Brown, Fraser; Lerner, Sorin; McMullen, Tyler; Savage, Stefan; Stefan, Deian. NDSS, 2021. The dissertation author was one of the co-authors of this material.

Chapter 4, in full, is a reprint of the material as it appears in the USENIX Security Symposium 2021. Narayan, Shravan; Disselkoen, Craig; Moghimi, Daniel; Cauligi, Sunjay; Johnson, Evan; Gang, Zhao; Vahldiek-Oberwagner, Anjo; Sahita, Ravi; Shacham, Hovav; Tullsen, Dean; Stefan, Deian. USENIX, 2021. The dissertation author was the primary investigator and author of this material.

Chapter 5, contains material reprinted from the Symposium on Principles of Programming Languages (POPL) 2021. Kolosick, Matthew; Narayan, Shravan; Johnson, Evan; Watt, Conrad; LeMay, Michael; Garg, Deepak; Jhala, Ranjit; Stefan, Deian. POPL, 2021. The dissertation author was one of the co-authors of this material.

VITA

2015	Bachelor of Engineering, Computer Engineering, McGill University, Montreal, Canada
2016-2022	Research Assistant, Computer Science University of California San Diego, USA
2022	Master of Science, Computer Science University of California San Diego, USA
2022	Doctor of Philosophy, Computer Science University of California San Diego, USA

PUBLICATIONS

F. Brown, S. Narayan, Riad S. Wahby, Dawson Engler, R. Jhala, and D. Stefan. "Finding and preventing bugs in JavaScript bindings." IEEE Symposium on Security and Privacy (S&P), 2017.

M. Smith, C. Disselkoen, S. Narayan, F. Brown, and D. Stefan. "Browser history re:visited." USENIX Workshop on Offensive Technologies (WOOT), 2018.

J. Talpin, J. Marty, S. Narayan, D. Stefan, and R. Gupta. "Towards verified programming of embedded devices." Design, Automation and Test in Europe Conference (DATE), 2019. Invited paper.

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." USENIX Security Symposium, 2020.

T. Garfinkel, S. Narayan, C. Disselkoen, H. Shacham, and D. Stefan. "The road to less trusted code: Lowering the barrier to in-process sandboxing." Article in USENIX ;login; journal, Winter 2020.

E. Johnson, Y. Alhessi, D. Thien, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. "Доверя́й, но проверя́й: SFI safety for native-compiled Wasm." Network and Distributed System Security Symposium (NDSS), 2021.

S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, D. Stefan. "Swivel: Hardening WebAssembly against Spectre." USENIX Security Symposium, 2021.

S. Narayan, C. Disselkoen, and D. Stefan. "Tutorial: Using RLBox to sandbox unsafe C code." IEEE Secure Development Conference (SecDev), 2021.

M. Kolosick, S. Narayan, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan. “Isolation without taxation: Near zero cost transitions for SFI.” Principles of Programming Languages (POPL), 2022.

S. Narayan, J. Rudek, T. Garfinkel, K. Taram, D. Moghimi, S. Sultana, A. Vahldiek-Oberwagner, R. Sahita, D. Tullsen, D. Stefan. “HFI: Fast isolation and WebAssembly with hardware-based fault isolation.” Unpublished.

ABSTRACT OF THE DISSERTATION

Retrofitting fast and secure sandboxing in real systems

by

Shravan Ravi Narayan

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Deian Stefan, Chair

The applications we use today are developed as a combination of first-party code and code borrowed from third-parties. This has allowed developers to build large applications with rich feature sets. Unfortunately, when we borrow code we don't just borrow its functionality, we also inherit its bugs. These bugs are particularly serious for systems like browsers that are written in C and C++ as they are often memory safety bugs. In the last decade, security researchers have disclosed numerous instances of memory safety bugs in third-party code being targeted by attackers to compromise systems ranging from browsers to messaging clients.

The most pragmatic way to prevent such attacks is to sandbox this third-party code, i.e., confine the code to its own region of memory, separate from the rest of the application. This is

an old idea. Alas, it has only seen limited adoption; sandboxing today is only used to limit the damage of bugs in a handful of applications like OpenSSH and browsers; it has never been used to sandbox third-party code.

In this dissertation, we bridge this gap with an end-to-end framework to sandbox third-party code. This required addressing challenges on three fronts: engineering, security, and performance. On the engineering front, we needed to simplify the retrofitting of sandboxing in existing applications; to address this, we built RLBox, a type-driven framework that helps developers incrementally adopt sandboxing. On the security front, we needed to ensure that the sandboxing tools we use provide reliable security guarantees; to address this, we built VeriWasm, a validator that ensures bugs in sandboxing tools do not break sandboxing, and Swivel, a compiler that hardens sandboxing against Spectre — transient execution attacks which exploit the underlying hardware to break sandboxing. On the performance front, we needed to minimize the overheads of sandboxing, specifically, the overheads of an application switching to and from sandboxed code; to address this, we built Zero-Cost, a compiler pass that eliminates all transition costs between applications and sandboxed code.

Our work demonstrates that sandboxing is an effective way to secure today's applications from bugs in third-party code.

Chapter 1

Introduction

It is impossible for programmers to develop the applications we use today without borrowing code. This borrowing of code is key to building large applications. However, it also comes at a cost: borrowed code can contain bugs that can compromise an otherwise carefully written and tested application. Indeed numerous attacks have targeted bugs in borrowed code to compromise applications and steal data from the users of the applications.

For example, researchers at Google's Project Zero observed that malicious websites were stealing user information by compromising the Google Chrome web browser [Goo21b]. These websites carried out this attack by exploiting a bug in `libfreetype` — a third-party code component used by Chrome to display different fonts. This attack on Chrome is just one of the many attacks that have exploited bugs in such seemingly benign libraries.

The core problem targeted by these attacks is that by borrowing code, developers are not only acquiring the features of the borrowed code (such as the ability to display different fonts), but also all of its bugs. Of these bugs, the largest source of vulnerabilities we see in large applications like browsers and operating systems are *memory safety* bugs [Mil19, Chr20].

1.1 Memory safety bugs

Memory safety bugs are prevalent in browsers and operating systems because they are written in low-level languages like C and C++. The advantage of these languages is that they allow writing code that is extremely fast and portable across architectures. This is, in part, because these languages allow developers to control all aspects of memory management at a low-level, under the assumption that applications will handle such memory management correctly. Unfortunately, this assumption rarely holds in practice [Mil19, Chr20], and as a result, these application contain memory safety bugs such as use-after-frees (use of memory locations which are marked unused) and buffer overflows (unintentional access of memory locations outside the intended memory buffer).

The most direct way to combat such bugs is to ensure the languages we use are memory-safe by default. To this end, researchers have identified techniques to retrofit memory safety into languages like C and C++ [NZMZ09, NZMZ10]. These techniques unfortunately result in large overheads in applications' performance (around 120%) and thus have not seen real-world adoption. New memory-safe languages like Rust offer both safety and speed, but in practice, there is no easy way to migrate the billions of lines of existing C and C++ code.

We need a solution that allows us to securely use the existing C and C++ code safely, without giving up on performance.

1.2 Sandboxing buggy code in a separate process

Researchers have explored a variety of techniques to address the challenge of reusing buggy code by hardening unmodified code to attacks targeting memory safety bugs [ABEL09, WWN+15, NZMZ09, NZMZ10, BZP19, KSP+14, SPWS13, FSA97]. The most pragmatic approach — the focus of this dissertation — is to sandbox the buggy third-party code components in applications. Sandboxing can be implemented in a straightforward manner by simply moving

buggy code into a separate process; this would prevent attackers from exploiting memory safety bugs to compromise applications, as the effects of these bugs are confined to the memory of this separate process.

While process-based sandboxing seems simple, it has not seen adoption outside of a few applications such as OpenSSH [PFH03] and web browsers [RMO19]. We observe that the adoption of this technique is hindered due to two challenges:

1. Engineering effort. The engineering challenge of retrofitting sandboxing in large applications is substantial, often requiring significant changes or even full re-architecture of applications. Applications need this re-architecture so that they correctly handle data structures shared between the application and sandboxed components. Specifically, applications must marshall these shared data structures back and forth from the sandboxed process' memory, so that both the application code and the sandboxed components can access these data structures. Applications must also sanitize any data produced by a sandboxed component to check for malformed values. While adding these data sanitization checks is conceptually simple, in-practice developers have to correctly identify and insert an intractably-large number of security-sensitive checks when working with real applications. For example, it took Chrome over five years to re-architect the browser to isolate sites in separate processes [RMO19].

2. Performance. A central challenge of process-based sandboxing is the large performance overhead it introduces in applications. This is because applications can no longer invoke functions in process sandboxed code with a simple function call, rather they must use inter process communication (IPC) to communicate across processes — an operation that takes about 1000× longer than a function call (§2.6.2). As a result of these IPC overheads, we found that using process-based sandboxing to isolate image rendering code in web browsers incurs an overhead of approximately 10× (Chapter 2). The overhead of IPC is a well-known problem when sandboxing code using a separate processes [WLAG93].

Practical adoption of sandboxing is possible only once these blocking challenges have

```
int data = *p;
```

(a) Unsandboxed

```
if (p < 0x1000 || p > 0x2000)
  abort();
int data = *p;
```

(b) Slow sandboxing with bounds checks

```
p = 0x1000 + (p & 0xFFF);
int data = *p;
```

(c) Fast SFI sandboxing with masks

Figure 1.1: This code snippet illustrates how SFI tools use runtime checks to sandbox code to a specified region referred to as the *sandbox memory* (between 0x1000 to 0x2000 in this example). Figure 1.1a shows the original unsandboxed code, Figure 1.1b shows how we can naively sandbox code by inserting (slow) bounds checks, Figure 1.1c shows how SFI tools efficiently sandbox code with masking instructions.

been addressed. As a consequence, researchers have spent the last several decades attempting to bridge this gap. On the engineering side, there have been several tools to automate and assist with the partitioning an application into separate processes [Kil03, BS04, WSYL12, ST14, GWA+15, BSI19]; unfortunately these tools have not seen adoption. On the performance side, the breakthrough idea came from Wahbe et. al [WLAG93], who significantly improved the performance of sandboxing by eliminating the need for a separate process altogether.

Next, we discuss how Wahbe's approach, along with a several works that built on this approach, tried to address these challenges.

1.3 Sandboxing buggy code within a process

In 1993, Wahbe et. al [WLAG93] proposed a new, efficient approach to sandboxing code within a single process called *software-based fault isolation (SFI)*. SFI uses a special compiler to ensure the generated code always has a runtime check inserted prior to each memory access instruction (Figure 1.1). This runtime check ensures that the address accessed by the subsequent instruction is restricted to a specified region — the sandbox memory. Rather than implementing this runtime check using simple bounds checks (Figure 1.1b), SFI compilers use bit masking operations to implement these same checks in an extremely efficient way (Figure 1.1c). The result is that SFI can sandbox components within the same process with a runtime overhead between

10% and 30%.

In the last three decades, numerous works have built on Wahbe's design [Tan17], improving the performance and portability of SFI (ensuring it worked well across all CPU architectures, including the complex CISC architectures like x86). This work culminated in WebAssembly (Wasm) [HRS+17] — a modern portable SFI compiler toolchain that allows application developers to efficiently sandbox buggy components. In our experiments, we show that modern SFI toolchains like Wasm can isolate the image rendering code in browsers with an overhead of about 30%(§2.6) — a far better performance overhead than the 10× overheads of process-based sandboxing.

SFI allows us to sandbox components without re-architecting applications to use processes, and without incurring the overhead of IPC; but this comes at the cost of tackling a different set of challenges with retrofitting and performance. Specifically, applications must now deal with some incompatibilities introduced by modern SFI compilers, as well as the overheads introduced by SFI security checks. Additionally, SFI is vulnerable to bugs in the tooling as it relies on the compiler to automatically insert security checks. We explore these challenges next.

1.4 The challenges of in-process sandboxing with SFI

In this section, we explore the challenges that have hindered the adoption of SFI sandboxing. These challenges are three-fold: minimizing engineering effort, ensuring performance, and providing strong security guarantees. The first two — engineering effort and performance — echo the challenges we observe when sandboxing code with processes, while the third challenge — security — is unique to SFI sandboxing, and stems from SFI's use of compilers to ensure security. We discuss these challenges in more detail below.

1. Engineering effort. SFI compilers employ a number of changes during compilation to optimize the performance of sandboxed code. For example, SFI compilers change the calling convention of

sandboxed functions to optimize handling of sandbox metadata (such as the location of sandbox memory), and optimize memory consumption of sandboxed code by changing the size of pointers in data structures [HRS+17, SMB+10]. These optimizations, however, mean that retrofitting SFI in applications is more complicated than retrofitting sandboxing using processes. Specifically, the application is now responsible for adding code to account for these different calling conventions as well as data structure layout differences (ABI differences), in addition to the large number of code changes needed by the application for marshalling and sanitizing shared data structures.

2. Performance. SFI employs runtime checks to sandbox components in place of using a separate process; in essence, SFI has eliminated the overheads due to IPC in exchange for incurring overhead when executing sandboxed code. While SFI's approach eliminates the worst overheads of sandboxing, it still has limitations. First, SFI's approach only makes sense when applications invoke functions in the sandboxed code frequently (typical for workloads such as image rendering in the browser), but is worse for workloads that do this sporadically. Second, SFI replaces IPC with its own security checks which also have overhead. Specifically, these checks ensure the machine state shared between the application and the sandboxed component's code — the execution stack and CPU registers — are saved, scrubbed, and restored on each function call. These checks introduce substantial overheads in real systems — for instance, they slow down sandboxed font rendering anywhere between 10% and 4× (§5.4.2).

3. Security. SFI requires runtime checks to be automatically added during compilation by sandboxing compilers. These sandboxing compilers, however, have two limitations that make it challenging for SFI to provide strong sandboxing guarantees. First, SFI compilers are complex and often contain bugs that inadvertently break sandboxing [Han19a, Ryd20]. Second, SFI compilers cannot prevent code from bypassing sandboxing checks using Spectre attacks [KHF+19] (attacks that leverage microarchitectural behavior of the CPU to break security boundaries). Researchers have made attempts to address these challenges; however, these approaches have not been adopted in the current SFI compilers (such as Wasm) due to performance considerations. For

instance, previous work tackled the challenge of detecting SFI compiler bugs by ensuring SFI compilers only emit easily-verifiable runtime checks [MTT+12, MM06]; while this was adopted by the previous generation of SFI compilers [YSD+09], newer SFI compilers are designed to optimize runtime checks for performance [Sea13, HRS+17] and thus cannot use this approach. There is also work on fortifying the code produced by compilers to be Spectre resistant [VDG+21, SZOC19, Int20b, Mic20a]; however, as these approaches ensure security by restricting speculative execution[1], they also have a high overhead and have not been adopted in SFI compilers.

SFI has addressed many of the shortcomings of sandboxing using processes, however, it has done so by employing trade-offs which have limited its use for sandboxing buggy code in real applications. Thus, SFI techniques today are mainly used to efficiently run native *applications* in web browsers [HRS+17] and in cloud platforms [Pat19], however, it remains unused for the original problem it was meant to solve [WLAG93] — to sandbox third-party code within an application.

To address these challenges, we need a framework to make the sandboxing of third-party code practical in real applications. We discuss our approach to this framework next.

1.5 Overview of this dissertation

In this dissertation, we designed and built an end-to-end framework to address the core sandboxing challenges of engineering effort, security, and performance. This framework uses several distinct components. RLBox, for instance, helps reduce the engineering effort needed to retrofit sandboxing in existing applications, VeriWasm and Swivel help ensure sandboxing tools provide reliable security guarantees, while Zero-Cost improves the performance of sandboxing without giving up on security.

[1] Speculative execution is restricted by either using fence instructions or specialized code constructs like retpolines [Tur18]

This framework tackles the challenges of practical sandboxing, and has been adopted to bring fast, secure sandboxing to today's applications. For example, RLBox and Zero-Cost are used by the Firefox web browser to sandbox buggy libraries [Hol21], while VeriWasm is used by Fastly's FaaS platform to ensure that their sandboxing tools are not affected by bugs that compromise the security of sandboxing [Joh]. Below, we briefly describe each of these tools, and outline the rest of this dissertation.

Chapter 2 presents RLBox, a framework to easily retrofit sandboxing in existing large applications. RLBox simplifies the task of retrofitting sandboxing by leveraging the type system to tackle the laborious, error-prone, and security sensitive code changes needed to adopt sandboxing in application. Specifically, RLBox uses types to incrementally ensure (1) data from a sandboxed component is sanitized before use in the application, and (2) data is marshalled to and from the sandbox memory while accounting for any ABI differences. RLBox uses a combination of automation (where possible) and feedback in the form of compile-time errors to help developers make the changes needed to adopt sandboxing in their application.

Chapter 3 presents VeriWasm, a binary validator which scans binaries produced by SFI compilers to ensure that a binary has been correctly sandboxed. VeriWasm ensures that bugs in sandboxing compilers cannot compromise the security of sandboxing. To ensure VeriWasm can efficiently check binaries, we use abstract interpretation to check several properties of the produced binary, which together imply that the binary is correctly sandboxed. We also discuss how to ensure VeriWasm itself is bug-free by formally checking the validation approach for correctness.

Chapter 4 presents Swivel, a compiler based approach to hardening sandboxing against Spectre. Unlike prior defenses, Swivel provides sound protections against Spectre, *without* the use of fence instructions which would hinder the benefits from speculative execution. Swivel operates by building reliable security checks around the idea of *linear blocks* — a construct that breaks each sandboxed function into straight line sequences of instructions — so that any security

checks do not rely on the behavior of an inaccurate branch predictor.

Chapter 5 presents Zero-Cost, a set of restrictions that SFI compilers can follow during compilation, to ensure function calls from the application to sandboxed code are fast. Specifically, Zero-Cost imposes structure on the output of sandbox compilers to ensure that all code produced by the compiler safely handles the machine state shared between the application and sandboxed code (for example, the program stack and CPU registers). Zero-Cost thus allows applications to safely elide slow security checks on these resources that would normally be run before and after each function call to the sandboxed component.

1.6 Chapter acknowledgments

The Introduction, in part, uses material from the remaining chapters.

Chapter 2

RLBox: Retrofitting library sandboxing

All major browsers today employ coarse grain *privilege separation* to limit the impact of vulnerabilities. To wit, they run *renderers*—the portion of the browser that handles untrusted user content from HTML parsing, to JavaScript execution, to image decoding and rendering—in separate sandboxed processes [BJRt08, Moz18b, RG09]. This stops web attackers that manage to compromise the renderer from abusing local OS resources to, say, install malware.

Unfortunately, this is no longer enough: nearly everything we care about today is done through a website. By compromising the renderer, an attacker gets total control of the current site and, often, any other sites the browser has credentials for [DPF06]. With services like Dropbox and Google Drive, privilege separation is insufficient even to protect local files that sync with the cloud [JCH+16].

Browser vendors spend a huge amount of engineering effort trying to find renderer vulnerabilities in their own code [Kwo17]. Unfortunately, many remain—frequently in the dozens of third-party libraries used by the renderer to decode audio, images, fonts, and other content. For example, an out-of-bounds write in libvorbis was used to exploit Firefox at Pwn2Own 2018 [Bro18]. Both Chrome and Firefox were vulnerable to an integer-overflow bug in the libvpx video decoding library [Bug15]. Both also rely on the Skia graphics library, which had four

remote code execution bugs until recently [Nic19, Chr19].

To appreciate the impact of these vulnerabilities and the difficulty of mitigating them, consider a typical web user, Alice, that uses Gmail to read email in her browser. Suppose an intruder, Trudy, sends Alice an email that contains a link to her malicious site, hosted on `sites.google.com`. If Alice clicks on the link, her browser will navigate her to Trudy's site, which can embed an `.ogg` audio track or `.webm` video to exploit vulnerabilities in libvorbis and libvpx and compromise the renderer of Alice's browser. Trudy now has total control of Alice's Gmail account. Trudy can read and send emails as Alice, for example, to respond to password reset requests from other sites Alice belongs to. In most cases, Trudy can also attack cross site [DPF06], i.e., she can access any other site that Alice is logged into (e.g., Alice's `amazon.com` account).

Recent version of Chrome (and upcoming versions of Firefox) support *Site Isolation* [RMO19], which isolates different *sites* from each other (e.g., `*.google.com` from `*.amazon.com`) to prevent such cross-site attacks. Unfortunately, Trudy might still be able to access `{drive, pay, cloud}.google.com`, which manage Alice's files, online payments, and cloud infrastructure—since the renderer that loads the malicious `.ogg` and `.webm` content might still be running in the same process as those origins.

For many sites, Trudy might not even need to upload malicious content to the (trusted) victim origin (`sites.google.com` in our example). Most web applications load content, including images, fonts, and video, from different origins. Of the Alexa top 500 websites, for example, over 93% of the sites load at least one such cross-origin resource (§2.6.1). And the libraries handling such content are not isolated from the embedding origin, even with Site Isolation [RMO19].

To mitigate these vulnerabilities, we need to harden the renderer itself. To this end, we extend the Firefox renderer to isolate third party libraries in fine grain sandboxes. Using this, we can prevent a compromised library from gaining control of the current origin or any other origin in the browser.

Making this practical poses three significant challenges across three dimensions. First, *engineering effort*—we need to minimize the upfront work required to change the renderer to use sandboxing, especially as this is multiplied across dozens of libraries; minimizing changes to libraries is also important as this can significantly increase the burden of tracking upstream changes. Second, *security*—the renderer was not built to protect itself from libraries; thus, we have to sanitize all data and regulate control flow between the library and renderer to prevent libraries from breaking out of the sandbox. In our experience, bugs at the library-renderer boundary are not only easy to overlook, but can nullify any sandboxing effort—and other developers, not just us, must be able to securely sandbox new libraries. Finally, *efficiency*—the renderer is performance critical, so adding user-visible latency is not acceptable.

To help us address these challenges, we develop a framework called RLBox that makes data- and control-flow at the library-renderer interface explicit, using types. Unlike prior approaches to sandbox automation that rely on extensive custom analysis frameworks (§2.7), RLBox is simply a library[1] that leverages the C++ type system and is easy to incorporate into Firefox's predominantly C++ codebase.

Using type information, RLBox can identify where security checks are needed, automatically insert dynamic checks when possible, and force compiler errors for any security checks that require additional user intervention. Our type-driven approach enables a systematic way to migrate Firefox's renderer to use sandboxed libraries and allows RLBox to support secure and efficient sharing of data structures between the renderer and library (e.g., by making shared memory operations safe and by lazily copying data out of the sandbox).

To enable efficient sandboxing, we adapt and evaluate two isolation mechanisms for library sandboxing: software-based fault isolation (SFI) leveraging Google's Native Client (NaCl) [YSD+09, SMB+10] and a multi-core process-based approach. We also explore applying sandboxing at different granularities (e.g., per-origin and per-library sandboxing) to find the

[1] Our only external tooling is a ~100LOC Clang plugin, described in Section 2.5.1, that makes up for C++'s currently limited support for reflection on structs.

appropriate balance between security and sandboxing overhead.

To evaluate RLBox, we sandbox several libraries in Firefox: the libjpeg and libpng image decoding libraries, the libvpx and libtheora video decoding libraries, the libvorbis audio decoding library, and the zlib decompression library. Browsing a representative sample of both popular and unpopular websites (§2.6), we find the end-to-end memory overhead of RLBox to be modest—25% with SFI, 18% with process isolation—and transient, appearing only at content load time. The impact on page latency is small: 3% and 13% with SFI and process isolation, respectively. Our sandboxing does not noticeably impact the video frame rates nor audio decoding bitrate.

Our evaluation shows that retrofitting fine grain isolation, especially using SFI, is practical—and we've been integrating RLBox into production Firefox [Shr].[2] Since NaCl has been deprecated [Goo17] in favor of WebAssembly (Wasm) [HRS+17], our production sandbox also uses Wasm. We used RLBox with this Wasm-based sandbox to isolate the libGraphite font shaping library and are in the process of migrating several others [Fro20, Shr]. We describe this effort in Section 2.8.

Though we developed RLBox to sandbox libraries in Firefox, RLBox is a general library-sandboxing framework that can be used outside Firefox. To demonstrate this, we use RLBox to sandbox libraries in two different contexts: the Apache web server and Node.js runtime. For Apache, we sandbox the libmarkdown library that is used in the mod_markdown module [mod11]; we find that RLBox with the SFI sandbox increases the tail latency of the Apache server by 10% (4ms) and decreases the throughput by 27% (256 requests/second). For Node.js, we sandbox the C bcrypt library that is used by the JavaScript bcrypt module [nod10]; we measure RLBox with SFI to impose an overhead of 27% on hashing throughput.

Contributions. We present the case for sandboxing third party libraries in the browser renderer, and potential architectural trade-offs, including our approach (§2.1). We offer a taxonomy of security pitfalls encountered while migrating the Firefox code base to this architecture that were

[2]The Tor team is integrating our patches into the Tor Browser [Geo].

largely overlooked by previous work (§2.2), and RLBox, a framework we developed to prevent these pitfalls that leverages the C++ type system to enforce safe data and control flow (§2.3), and enables an incremental compiler-driven approach to migrating code to a sandboxed architecture (§2.4). We describe our implementation, including our software fault isolation and multi-core process-based isolation mechanisms (§2.5), and evaluate the performance of RLBox (§2.6). We close with a discussion of related work (§2.7) and our effort upstreaming RLBox into production Firefox (§2.8).

Availability. All work presented in this paper, including our modified Firefox builds, the RLBox library, and benchmarks are available and open source.[3]

2.1 Fine grain sandboxing: how and why

Renderers rely on dozens of third-party libraries to support media decoding and other tasks (e.g., decompression, which sites use to optimize page load times and bandwidth consumption). These are written almost exclusively in C and tasked with parsing a wide range of complex inputs. Unsurprisingly, exploitable vulnerabilities in this code are relatively frequent, even after years of scrutiny.

These libraries are a compelling place to employ sandboxing inside the renderer for several reasons. First, media content such as images and video are rich attack vectors, as web applications allow them to be shared pervasively. Over 93% of the Alexa Top 500 websites load such content cross-origin (§2.6.1). And nearly all forms of social media and peer-to-peer messaging platforms enable the sharing of images and video.

Next, content libraries can be effectively sandboxed, as they require little privilege to operate, i.e., once these libraries are memory isolated, the harm they can inflict is minimal. For example, an attacker that compromises an image decoding library could at worst change how

[3]Available at: https://usenix2020-aec.rlbox.dev.

images display. In contrast, sandboxing a highly privileged component like the JavaScript engine is largely ineffectual. An attacker with control over the JavaScript engine can run arbitrary JavaScript code and thus already has complete control of the web application.

Finally, the existing library-renderer interface provides a natural place to partition code. Compared to coarse grain techniques like privilege separation or Site Isolation, which spin up entire new renderer processes, spinning up a sandbox for a library is very cheap (§2.6). Moreover, because library sandboxes are only needed during content decoding, their memory overhead is transient.

Isolation strategies. A key question remains: *what grain of isolation should be employed?* In particular, different architectures have different implications for performance and security. Prior to RLBox, Firefox was largely exploring a coarse grain approach to library sandboxing, placing certain media libraries into a single sandboxed media process [Moz18b]. This approach has some benefits for performance as there is only one sandbox, but trades off security.

First, the assurance of the sandbox is reduced to that of the weakest library. This is less than ideal, especially when we consider the long tail of infrequently used media libraries required to preserve web compatibility (e.g., Theora) which often contain bugs. Next, the attacker gains the power of the most capable library. Some libraries handle *active content*—zlib, for example, is used to decompress HTTP requests that could contain HTML or JavaScript—as opposed to *passive content* such as images or fonts. Thus compromising a passive library like libvorbis, could still enable powerful attacks—e.g., modify the JavaScript decompressed by zlib. When multiple renderers share a common library sandbox, an intruder can attack across tabs, browsing profiles, or sites. Finally, coarse grain sandboxing does not scale to highly performance-critical libraries, such as libjpeg and libpng (§2.6.5).

RLBox lets us employ more granular sandboxing policies that can address these shortcomings. Its flexibility lets us explore the performance implications of various sandboxing architectures with different isolation mechanisms (§2.6.4).

In this paper, we largely employ a unique sandbox per <renderer, library, content-origin, content-type>. This mitigates many of the problems noted above, while still offering modest memory overheads. Per-renderer sandboxing prevents attacks across tabs and browsing profiles. Per-library ensures that a weakness in one library does not impact any other library. Per-content-origin sandboxing prevents cross origin (and thus cross site) attacks on content. For example, a compromise on sites.google.com as discussed in our example in Section 2, should not impact content from pay.google.com. Per-content-type sandboxing addresses the problem of passive content influencing active content.

Both finer and coarser grain policies are practically useful, though. In production Firefox, for example, we create a fresh sandbox for each Graphite font instance (§2.8). But, we also foresee libraries where, say, same-origin is sufficient.

Attacker model. We assume a web attacker that serves malicious (but passive) content—from an origin they control or by uploading the content to a trusted origin—which leads to code execution (e.g., via a memory safety vulnerability) in a RLBox sandbox. RLBox ensures that such an attacker can only affect (corrupt the rendering of and potentially leak) content of the same type, from the same origin. Per-object (or per-instance) sandboxing can further reduce the damage of such attacks. We, however, only use this policy when sandboxing audio, videos, and font shaping—we found the overheads of doing this for images to be prohibitive.

We consider side channels out-of-scope, orthogonal challenges. With side channels, an attacker doesn't need to exploit renderer vulnerabilities to learn cross-origin information, as browsers like Firefox largely do not prevent cross-origin leaks via side channels. Fuzzy-Fox [KS16] and cross-origin read blocking [RMO19] are promising ways to tackle these channels.

For the same reason, we consider transient execution attacks (e.g., Spectre [KHF+19]) out of scope. We believe that our SFI and our process-based isolation mechanisms make many of these attacks harder to carry out—e.g., by limiting transient reads and control flow to the sandbox memory and code, respectively—much like Site Isolation [RMO19]. But, in general, this is not

enough: an attacker could potentially exploit code in the renderer to transiently leak sensitive data. We leave the design of a Spectre-robust sandbox to future work.

The protections offered by RLBox are only valid if the Firefox code that interfaces with the sandboxed library code is retrofitted to account for untrusted code running in the library sandbox. As we discuss next, this is usually notoriously difficult to get right. RLBox precisely reduces this burden to writing a series of validation functions. In our attacker model, we assume these functions to be correct.

2.2 Pitfalls of retrofitting protection

The Firefox renderer was written assuming libraries are trusted. To benefit from sandboxing requires changing our threat model to assume libraries are untrusted, and modify the renderer-library interface accordingly (e.g, to sanitize untrusted inputs).

While migrating to this model we made numerous mistakes—overlooking attack vectors and discovering many bugs only after building RLBox to help detect them. We present a brief taxonomy of these mistakes, with examples drawn from the code snippet illustrating the interface between the renderer's JPEG decoder and libjpeg[4] shown in Figure 2.1. We discuss how RLBox helps prevent these attacks in Section 2.3.

For the rest of this section, we assume that libjpeg is fully sandboxed or memory isolated, i.e., libjpeg code is restricted from accessing arbitrary memory locations in the renderer process, and may only access memory explicitly dedicated to the sandbox—the *sandbox memory*. The renderer itself can access any memory location including sandbox memory.

[4]We use libjpeg interchangeably with libjpeg-turbo, the faster fork of the original libjpeg library which is used in Firefox.

```
// InitInternal registers callbacks for libjpeg to call during img decoding
nsresult nsJPEGDecoder::InitInternal() { ...
  mInfo.client_data = (void*)this; ...
  //Callbacks invoked by libjpeg
  mErr.pub.error_exit = my_error_exit;
  mSourceMgr.fill_input_buffer = fill_input_buffer;
  mSourceMgr.skip_input_data = skip_input_data; ...
}

// Adjust output buffers for decoded pixels
void nsJPEGDecoder::OutputScanlines(...) { ...
  while (mInfo.output_scanline < mInfo.output_height) { ...
    imageRow = ... +  (mInfo.output_scanline * mInfo.output_width); ...
  } ...
}

// Invoked if some input bytes are not needed
void skip_input_data (..., long num_bytes) { ...
  if (num_bytes > (long)src->bytes_in_buffer) { ... }
  else { src->next_input_byte += num_bytes; }
}

// Invoked repeatedly to get input as it arrives
void fill_input_buffer (j_decompress_ptr jd) {
  struct jpeg_source_mgr* src = jd->src;
  nsJPEGDecoder* decoder = jd->client_data;
  ...
  src->next_input_byte = new_buffer;
  ...
  if (/* buffer is too small */) {
    JOCTET* buf = (JOCTET*) realloc(...);
    if (!buf) {
      decoder->mInfo.err->msg_code = JERR_OUT_OF_MEMORY; ...
      ...
    } ...
  } ...
  memmove(decoder->mBackBuffer + decoder->mBackBufferLen, src->next_input_byte,
  ↪  src->bytes_in_buffer); ...
}

// Invoked on a decoding error
void my_error_exit (j_common_ptr cinfo) {
  decoder_error_mgr* err = cinfo->err; ...
  longjmp(err->setjmp_buffer, error_code);
}
```

Figure 2.1: *The renderer-library interface:* this code snippet illustrates the renderer's interface to the JPEG decoder and is used as a running example. The decoder uses libjpeg's streaming interface to decode images one pixel-row at a time, as they are received. Receiving and decoding concurrently is critical for responsiveness.

2.2.1 Insecure data flow

Failing to sanitize data. Failing to sanitize data received from libjpeg including function return values, callback parameters, and data read from sandbox shared memory can leave the renderer vulnerable to attack. For example, if the renderer uses the `num_bytes` parameter to the `skip_input_data()` callback on line 18 of Figure 2.1 without bounds checking it, an attacker-controlled libjpeg could could force it to overflow or underflow the `src->next_input_byte` buffer.

Pointer data is particularly prone to attack, either when pointers are used directly (with C++'s `*` and `->` operators) or indirectly (via memory functions such as `memcpy()` and `memmove()`, array indexing operations, etc.). For example, if the parameter `jd` of `fill_input_buffer()` is not sanitized (line 24), the read of `jd->src` on line 25 becomes an arbitrary-read gadget. In the same callback, if both `jd` and `src` are unsanitized, the write to `src->next_input_byte` on line 28 becomes an arbitrary-write gadget. Similar attacks using the `memmove()` on line 37 are possible.

Missing pointer swizzles. Some sandboxing mechanisms—e.g., NaCl (§2.5.2) and Wasm (§2.8)—use alternate pointer representations. Some sandboxing tools e.g., NaCl (§2.5.2) and Wasm (§2.8) use alternate pointer representations for efficiency.

Though this is often done for performance, in Wasm's case this is more fundamental: Wasm pointers are 32-bit whereas Firefox pointers are 64-bit. We must translate or *swizzle* pointers to and from these alternate representations when data is transferred between the renderer and the sandboxed libjpeg.

We found that doing this manually is both tedious and extremely error prone. This is largely because pointers can be buried many levels deep in nested data structures. Overlooking a swizzle either breaks things outright, or worse, silently introduces vulnerabilities. For instance, failing to swizzle the nested pointer `mInfo.err` on line 33 prior to dereferencing, can result in a write gadget (whose write-range depends on the precise pointer representation).

Leaking pointers. Leaking pointers from the Firefox renderer to the sandboxed libjpeg can allow

an attacker to derandomize ASLR [SPP+04] or otherwise learn locations of code pointers (e.g., C++ virtual tables). Together with an arbitrary-write gadget, this can allow an attacker-controlled libjpeg to execute arbitrary code in the renderer.

In our example, the renderer saves pointers to `nsJPEGDecoder` objects in libjpeg **`struct`**s (line 3), which alone allows an attacker to locate code pointers—the `nsJPEGDecoder` class is derived from the `Decoder` class, which defines virtual methods and thus has a virtual table pointer as the first field. Even initializing callbacks (line 6) could leak pointers to functions and reveal the location of Firefox's code segment[5].

Double fetch bugs. RLBox uses shared memory (§2.3) to efficiently marshal objects between the renderer and the sandboxed libraries. This, unfortunately, introduces the possibility of double fetch bugs [WKL+17, SGL+18, XQL+18].

Consider the `mInfo` object used in Figure 2.1. Since this object is used by both libjpeg and the renderer, RLBox stores it in shared memory. Now consider the bounds check of `mInfo.output_scanline` on line 12 prior to the assignment of output buffer `imageRow`. In a concurrent libjpeg sandbox thread, an attacker can modify `mInfo.output_scanline` after the check (line 12), and before the value is fetched (again) and used on line 13. This would bypasses the bounds check, leading to an arbitrary-write gadget. While this example is obvious, double-fetch bugs often span function boundaries and are much harder to spot.

2.2.2 Insecure control flow

Isolation prevents arbitrary control transfers from the sandbox into the renderer. Thus, out of necessity, callbacks for libjpeg must be explicitly exposed. But this alone is not sufficient to prevent attacks.

Corrupted callback state. Callbacks may save state in the sandboxed library. An attacker-

[5]Whether callback locations are leaked depends on the underlying sandboxing mechanism. While both our process isolation and NaCl use jump tables and thus do not leak, other sandbox implementations could leak such information.

controlled libjpeg can abuse the control flow of callbacks by corrupting this state. For example, on line 3 of Figure 2.1, the renderer stores a pointer to the `nsJPEGDecoder` object into the `client_data` field of `mInfo`. Inside `fill_input_buffer()` this pointer is used to access the `nsJPEGDecoder` object (line 26). Failing to sanitize `client_data` before using it allows an attacker to set the pointer to a maliciously crafted object and hijack control flow when Firefox invokes a virtual method on this object.

Unexpected callback invocation. Security bugs can also occur if an attacker controlled libjpeg invokes a permitted callback at unexpected times. Consider the `my_error_exit()` callback function, which uses `longjmp()` to implement error handling. On line 43, `longjmp()` changes the instruction pointer of the renderer based on information stored in `setjmp_buffer`. If an attacker invokes `my_error_exit()` before `setjmp_buffer` is initialized, they can (again) hijack the renderer control flow.

Callback state exchange attacks. Threading introduces another vector to attack callback state. When Firefox decodes two images in parallel, two decoder threads make calls to libjpeg. Firefox expects libjpeg to invoke the `fill_input_buffer()` callback on each thread with the corresponding `nsJPEGDecoder` object. But, an attacker could supply the same `nsJPEGDecoder` object to both threads when calling `fill_input_buffer()`. If the first thread reallocates the source buffer (line 31), while the second thread is using it to get input bytes, this can induce a data race and use-after-free vulnerability in turn.

2.3 RLBox: automating secure sandboxing

Modifying the Firefox JPEG decoder to guard against all the attacks discussed in Section 2.2 requires substantial code additions and modifications. Doing this manually is extremely error prone. Moreover, it also makes the code exceedingly fragile: anyone making subsequent changes to the decoder must now have an intimate knowledge of all the necessary checks to have

any hope of getting it right. Multiply this by the number of libraries Firefox supports and number of developers working on the renderer, and the problem becomes intractable.

We built the RLBox framework to tackle these challenges. RLBox helps developers migrate and maintain code in the Firefox renderer to safely use sandboxed libraries. We designed RLBox with the following goals in mind:

1. *Automate security checks:* Adding security checks un-assisted is labor intensive and error prone, as discussed (§2.2). However, most of the sandboxing pitfalls can be detected and prevented through static checks, i.e., through compile-time errors indicating where code needs to be manually changed for security, or eliminated with dynamic checks and sanitizations (e.g., pointer swizzling and bounds checking).

2. *No library changes:* We want to avoid making changes to libraries. When libraries come from third parties we do not necessarily understand their internals, nor do we want to. Additionally, any changes we make to libraries increases the effort required to track upstream changes.

3. *Simplify migration:* Firefox uses dozens of libraries, with occasional additions or replacements. Consequently, we want to minimize the per-library effort of using RLBox, and minimize changes to the Firefox renderer source.

In the rest of the section we give an overview of the RLBox framework and describe how RLBox addresses the pitfalls of Section 2.2 while preserving these goals.

2.3.1 RLBox overview

RLBox makes data and control flow at the renderer-sandbox interface explicit through its type system and APIs in order to mediate these flows and enforce security checks across the trust boundary.

RLBox mediates *data flow* with `tainted` types that impose a simple static information flow control (IFC) discipline [SM03]. This ensures that sandbox data is validated before any potentially unsafe use. It also prevents pointer leaks into the sandbox that could break ASLR.

RLBox mediates *control flow* through a combination of `tainted` types and API design. Tainting, for example, allows RLBox to prevent branching on `tainted` values that have not been validated. API design, on the other hand, is used to restrict control transfers between the renderer and sandbox. For instance, the renderer must use `sandbox_invoke()` to invoke functions in the sandbox; any callback into the renderer by the sandbox must first be registered by the renderer using the `sandbox_callback(callback_fn)` API.

Mediating control and data flow allows RLBox to:

- **Automate security checks:** Swizzling operations, performing checks that ensure sandbox-supplied pointers point to sandbox memory, and identifying locations *where* tainted data must be validated is done automatically.
- **Minimize renderer changes:** `tainted` data validation is enforced only when necessary. Thus, benign operations such as adding or assigning `tainted` values, or writing to tainted pointers (which are checked to ensure they point into the sandbox at creation time) are allowed by RLBox's type system. This eliminates needless changes to the renderer, while still ensuring safety.
- **Efficiently share data structures:** Static checks ensure that shared data is allocated in sandbox memory and accessed via `tainted` types. Data structures received by the renderer sandbox are marshaled lazily; this allows code to access a single field in a shared **`struct`** without serializing a big object graph. Finally, RLBox provides helper APIs to mitigate double fetches [WKL+17, SGL+18] when accessing this data.
- **Assist with code migration:** Compile-time type and interface errors (§2.4) guide the developer through the process of migrating a library into a sandbox. Each compile error points to the next required code change—e.g., data that needs to be validated before use,

or control transfer code that needs to be changed to use RLBox APIs.

- **Bridge machine models:** Sandboxing mechanisms can have a different machine model from the application (e.g., both Native Client and WebAssembly use 32-bit pointers and 32-bit `long`s regardless of the platform); by intercepting all data and control flow we can also automatically translate between the application and sandbox machine models—and for Wasm we do this (§2.8).

In the rest of this section, we discuss how tainting is used to mediate data flow (§2.3.2) and control flow (§2.3.4). We then describe how the renderer can validate and untaint data (§2.3.3). We detail our implementation as a C++ library later (§2.5.1).

2.3.2 Data flow safety

All data originating from a sandbox begins life tainted. Tainting is automatically applied by wrapping data with the `tainted<T>` constructor. Tainting does not change the memory layout of a value, only its type. Once applied, though, tainting cannot be removed. The only way to remove a taint is through explicit validation (§2.3.3).

In general, RLBox propagates taint following a standard IFC discipline. For example, we propagate taint to any data derived from `tainted` values such as data accessed through a `tainted` pointer, or arithmetic operations when one or more operands are `tainted`. We detail how RLBox implements `tainted` types and taint tracking in the C++ type system in Section 2.5.1. In the rest of this section we show how RLBox uses tainting to ensure data flow safety.

Data flow into the renderer. To protect the renderer from malicious inputs, all data flows from the sandbox into the renderer are `tainted`. Data primarily flows out of the sandbox through two interfaces. First, `sandbox_invoke()`, the only way to call into the sandbox, taints its return value. Second, the use of `sandbox_callback()`, which permits callbacks into the renderer from the sandbox, statically forces the parameters of callbacks to be `tainted`. Any code failing to

follow either of these rules would cause a compilation error.

As an example, consider the JPEG decoder code that calls libjpeg's `jpeg_read_header()` to parse headers shown below:

```
jpeg_decompress_struct mInfo;
int status = jpeg_read_header(&mInfo, TRUE);
```

With RLBox, the second line must be modified to use `sandbox_invoke()`, and `status` must be declared as `tainted`[6]:

```
tainted<int> status = sandbox_invoke(mRLBox, jpeg_read_header, &mInfo, TRUE);
```

In addition to the invoke and callback interfaces, data can flow into the renderer via pointers to sandboxed memory. RLBox, however, forces both these pointers and any data derived from them (e.g., via pointer arithmetic or pointer dereferencing) to be `tainted`—and, as we discuss shortly, using `tainted` pointers in the renderer is always safe.

Data flow into the sandbox. RLBox requires data flowing into sandbox from the renderer to either have a simple numeric type or a `tainted` type. Untainted pointers, i.e., pointers into renderer memory, are not permitted. This restriction enforces a code correctness requirement—sandboxed code only gets pointers it can access, i.e., pointers into sandbox memory—and, moreover, preserves the renderer's ASLR: any accidental pointer leaks are eliminated by construction.

Compile-time errors are used to guide the code changes necessary to use a sandboxed library. To demonstrate this, we continue with the example of JPEG header parsing shown above. To start, note that the `TRUE` parameter to `jpeg_read_header()` can remain unchanged as it has a simple numeric type (in C++). On the other hand, the parameter `&mInfo` points to a **`struct`** in renderer memory, which libjpeg cannot access; RLBox thus raises a compile-time error.

To address this compilation error, RLBox requires such shared data structures to be allocated in sandbox memory using `sandbox_malloc()`:

[6] In this paper, we use full type names such as `tainted<int>` for clarity. In practice, we use C++'s **`auto`** keyword to make code less verbose.

```
tainted<jpeg_decompress_struct*> p_mInfo =
 ↪  sandbox_malloc<jpeg_decompress_struct>(mRLBox);
tainted<int> status = sandbox_invoke(mRLBox, jpeg_read_header, p_mInfo, TRUE);
```

Placing shared data structures in sandboxed memory in this way simplifies data marshaling of pointer parameters during function calls—RLBox simply marshals pointers as numeric types, it does not eagerly copy objects. Indeed, this design allows RLBox to automatically generate the marshaling code without any user annotations or pointer bounds information (as required by most RPC-based sandboxing tools). Moreover, RLBox can do all of this without compromising the renderer's safety—renderer code can only access shared sandbox memory via `tainted` pointers.

While RLBox does not allow passing untainted pointers into libjpeg, pointers to callback functions, such as `fill_input_buffer()`, need to be shared with the sandbox—these can be shared either as function call parameters to libjpeg functions, return values from callbacks, or by directly writing to sandbox memory. RLBox permits this without exposing the raw callback pointers to libjpeg, through a level of indirection: *trampoline functions*. Specifically, RLBox automatically replaces each Firefox callback passed to libjpeg with a pointer to a trampoline function and tracks the mapping between the two. When the trampoline function is invoked, RLBox invokes the appropriate Firefox callback on libjpeg's behalf.

Benefits of tainted pointers. By distinguishing pointers to renderer memory from pointers to sandbox memory at the type level with `tainted`, RLBox can automatically enforce several important security requirements and checks. First, RLBox does not permit Firefox to pass untainted pointers to libjpeg. Second, RLBox automatically swizzles and unswizzles pointers appropriately when pointers cross the renderer-library boundary, including pointers in deeply nested data structures. (We give a more detailed treatment of pointer swizzling in Appendix 2.10.1.) Third, RLBox automatically applies pointer-bounds sanitization checks when `tainted` pointers are created to ensure they always point to sandboxed memory. Together, these properties ensure that we preserve the renderer's ASLR—any accidental pointer leaks are eliminated by construction—and that the

renderer cannot be compromised by unsanitized pointers—all `tainted` pointers point to sandbox memory.

2.3.3 Data validation

RLBox disallows computations (e.g., branching) on `tainted` data that could affect the renderer control and data flow. The Firefox renderer, however, sometimes needs to use data produced by library code. To this end, RLBox allows developers to unwrap `tainted` values, i.e., convert a `tainted<T>` to an untainted `T`, using validation methods. A validation method takes a closure (C++ lambda) that unwraps the tainted type by performing necessary safety checks and returning the untainted result. Unfortunately, it is still up to the user to get these checks correct; RLBox just makes this task easier.

RLBox simplifies the burden on the developer by offering different types of validation functions. The first, `verify(verify_fn)`, validates simple `tainted` value types that have already been copied to renderer memory (e.g., simple values), as shown in this example:

```
tainted<int> status = sandbox_invoke(mRLBox, jpeg_read_header, p_mInfo, TRUE);
int untaintedStatus = status.verify([](int val){
  if (val == JPEG_SUSPENDED ||
      val == JPEG_HEADER_TABLES_ONLY ||
      val == JPEG_HEADER_OK ) { return val; }
  else { /* DIE! */ }
});
if (untaintedStatus == JPEG_SUSPENDED) { ... }
```

Not all `tainted` data lives in renderer memory, though. Validating shared `tainted` data structures that live in sandbox memory is unsafe: a concurrent sandbox thread can modify data after it's checked and before it's used. The `copyAndVerify(verify_fn, arg)` validator addresses this by copying its arguments into renderer memory before invoking the `verify_fn` closure. To prevent

subtle bugs where a `verify()` function is accidentally applied to data in shared memory, RLBox issues a compile-time error—notifying the developer that `copyAndVerify()` is needed instead.

The `unsafeUnverified()` function removes tainting without any checks. This obviously requires care, but has several legitimate uses. For example, when migrating a codebase to use sandboxing, using `unsafeUnverified()` allows us to incrementally test our code before all validation closures have been written (§2.4). Furthermore, `unsafeUnverified()` is sometimes safe and necessary for performance—e.g., passing a buffer of libjpeg-decoded pixel data to the Firefox renderer without copying it out of sandbox memory. This is safe as pixel data is simple byte arrays that do not require complex decoding.

Validation in the presence of double fetches. Though our safe validation functions ensure that sandbox code cannot concurrently alter the data being validated, in practice we must also account for double fetch vulnerabilities.

Consider, for example, migrating the following snippet from the `nsJPEGDecoder::OutputScanlines` function:

```
1 while (mInfo.output_scanline < mInfo.output_height) {
2   ...
3   imageRow = reinterpret_cast<uint32_t*>(mImageData) +
4     ↪(mInfo.output_scanline * mInfo.output_width);
5   ...
6 }
```

Here, `mInfo` is a structure that lives in the sandbox shared memory. Buffer `imageRow` is a pointer to a decoded pixel-row that Firefox hands off to the rendering pipeline and thus must not be `tainted`. To modify this code, we must validate the results on lines 1 and 4 which are tainted as they rely on `mInfo`. Unfortunately, validation is complicated by the double fetch: a concurrent sandbox thread could change the value of `output_scanline` between its check on line 1 and its use on line 4, for example. Unsafely handling validation would allow the sandbox to control the value of `imageRow` (the destination buffer) and thus perform arbitrary out-of-bounds writes.

We could address this by copying `output_scanline` to a local variable, validating it once, and using the validated value on both lines. But, it's not always this easy—in our port of Firefox we found numerous instances of multiple reads, interspersed with writes, spread across different functions. Using local variables quickly became intractable.

To address this, RLBox provides a `freeze()` method on `tainted` variables and struct fields. Internally, this method makes a copy of the value into renderer memory and ensures that the original value (which lives in sandbox memory), when used, has not changed. To prevent accidental misuse of freezable variables, RLBox disallows the renderer from reading freezable variables and fields until they are frozen. RLBox does, however, allow renderer code to write to frozen variables—an operation that modifies the original value and its copy. Finally, the `unfreeze()` method is used to restore the sandbox's write access.

Unlike most other RLBox features, ensuring that a variable remains frozen imposes some runtime overhead. This is thus a compile-time, opt-in feature that is applied to select variables and **`struct`** fields.

Writing validators. We identify two primary styles of writing validators in our porting Firefox to use sandboxed libraries: we can focus on either preserving application invariants or on preserving library invariants when crossing the trust boundary. We demonstrate these two alternate styles, using the above `OutputScanlines` example.

1. *Maintaining application invariants:* The first focuses on invariants expected by Firefox. To do this, we observe that `imageRow` is a pointer into the `mImageData` buffer and is used as a destination to write one row of pixels. Thus, it is sufficient to ensure that the result of `output_scanline * output_width` is between `0` and `mImageDataSize - rowSize`. This means that the `imageRow` pointer has room for at least one row of pixels.

2. *Checking library invariants:* The second option focuses on invariants provided by libjpeg. This option assumes that the Firefox decoder behaves correctly when libjpeg is well-behaved.

Hence, we only need to ensure that libjpeg adheres to its specification. In our example, the libjpeg specification states that `output_scanline` is at most the height of the image: we thus only need to freeze `output_scanline` and then validate it accordingly.

2.3.4 Control flow safety

As discussed in Section 2.2, a malicious sandbox could attempt to manipulate renderer control flow in several ways. While data attacks on control flow are prevented by tainting (e.g., it's not possible to branch on a `tainted` variable), supporting callbacks requires additional support.

Control transfers via callbacks. It's unsafe to allow sandboxes to callback arbitrary functions in the renderer. It's also important to ensure they can only call functions which use `tainted` appropriately. Thus, RLBox forces application developers—via compile-time errors—to explicitly register callbacks using the `sandbox_callback()` function. For instance, line 6 in Figure 2.1, must be rewritten to:

```
mSourceMgr.fill_input_buffer = sandbox_callback(mRLBox,fill_input_buffer);
```

Statically whitelisting callbacks alone is insufficient—an attacker-controlled sandbox could still corrupt or hijack the control flow of the renderer by invoking callbacks at unexpected times. To address this class of attacks, RLBox supports unregistering callbacks with the `unregister()` method. Moreover, the framework provides RAII (resource acquisition is initialization) [Str13] semantics for callback registration, which allows useful code patterns such as automatically unregistering callbacks after completion of an invoked libjpeg function.

To deal with callback state exchange attacks (§2.2), RLBox raises a compile-time error when renderer pointers leak into the sandbox. For example, the JPEG decoder saves its instance pointer with libjpeg and retrieves it in the `fill_input_buffer()` callback, as shown on line 26 of Figure 2.1. RLBox requires the application developer to store such callback state in the application's thread local storage (TLS) instead of passing it to libjpeg. Thus, when `fill_input_buffer()`

is invoked, it simply retrieves the decoder instance from the TLS, preventing any pointer leaks or callback state modifications.

Non-local control transfers. A final, but related concern is protecting control flow via `setjmp()`/`longjmp()`. These functions are used for exception handling (e.g., `my_error_exit()` in Figure 2.1). They work like a non-local **`goto`**, storing various registers and CPU state in a **`jmp_buf`** on `setjmp()` and restoring them on `longjmp()`.

Naively porting libjpeg and libpng would store **`jmp_buf`** in sandboxed memory. Unfortunately, this doesn't work—there is no easy way to validate a **`jmp_buf`** that is portable across different platforms. We thus instead place such sensitive state in the renderer's TLS and avoid validation altogether. With libjpeg this is straightforward since the **`jmp_buf`** is only used in the `my_error_exit()` callback. libpng, however, calls `longjmp()` itself. Since we can't expose `longjmp()` directly, when sandboxing libpng, we expose a `longjmp()` trampoline function that calls back to the renderer and invokes `longjmp()` on libpng's behalf, using the **`jmp_buf`** stored in the TLS.

2.4 Simplifying migration

RLBox simplifies migrating renderer code to use sandboxed libraries while enforcing appropriate security checks. RLBox does this by removing error-prone glue code (e.g., for data marshaling) and by reducing the security-sensitive code required for migration. The resulting reduction in developer effort is evaluated in detail in Section 2.6.3.

The RLBox framework helps automate porting by (1) allowing Firefox developers to *incrementally* port application code—the entire application compiles and runs with full functionality (passing all tests) between each step of porting—and (2) guiding the porting effort with compiler errors which highlight what the next step should be. We illustrate how RLBox minimizes engineering effort using this incremental, compiler-directed approach using our running example.

To start, we assume a standard Firefox build that uses a statically linked libjpeg.

Step 1 (creating the sandbox). We start using RLBox by creating a sandbox for libjpeg using the `None` sandboxing architecture. As the name suggests, this sandbox does not provide isolation; instead it redirects all function calls back to the statically linked, unsandboxed libjpeg. However, RLBox still fully enforces all of its type-level guarantees such as tainting untrusted data. Thus, we can start using RLBox while still passing functional tests.

Step 2 (splitting data and control flow). Next, we migrate each function call to libjpeg to use the `sandbox_invoke()` API. RLBox flags calls passing pointers to the sandbox as compile-time errors after this conversion, as the sandbox will be unable to access the (application) memory being pointed to. To resolve this, we also convert the allocations of objects being passed to libjpeg to instead use `sandbox_malloc()`. For example, in Section 2.2, we rewrote:

```
tainted<int> status = sandbox_invoke(mRLBox, jpeg_read_header, &mInfo, TRUE);
```

to, instead, allocate the `mInfo` object in the sandbox:

```
tainted<jpeg_decompress_struct*> p_mInfo =
  sandbox_malloc<jpeg_decompress_struct>(mRLBox);
tainted<int> status = sandbox_invoke(mRLBox, jpeg_read_header, p_mInfo, TRUE);
```

At this point, we need to re-factor the rest of this function and several other JPEG decoder functions—`mInfo` is a data member of the `nsJPEGDecoder` class. Doing this in whole is exhausting and error-prone. Instead, we remove the `mInfo` data member and add one extra line of code in each member function before `mInfo` is first used:

```
jpeg_decompress_struct& mInfo = *(p_mInfo.unsafeUnverified());
```

This *unsafe alias pattern* allows the remainder of the function body to run unmodified, i.e., the alias defined in this pattern can be used everywhere the original `mInfo` variable is needed, albeit unsafely, as `unsafeUnverified()` temporarily suppresses the need for validation functions.

We also need to deal with return values from `sandbox_invoke()` which are `tainted`, either by writing validation functions to remove the taint or deferring this till the next step and using `unsafeUnverified()` to satisfy the type checker. Again, the entire application should now compile and run as normal.

Step 3 (hardening the boundary). Our next goal is to gradually remove all instances of the unsafe alias pattern, moving Firefox to a point where all data from the sandbox shared memory and all `tainted` return values are handled safely.

We can do this incrementally, checking our work as we go by ensuring the application still compiles and runs without errors. To do this, we simply move each unsafe-alias pattern downwards in the function source; as it moves below a given statement, that statement is no longer able to use the alias and must be converted to use the actual tainted value. This may involve writing validation functions, registering callbacks, or nothing (e.g., for operations which are safe to perform on `tainted` values). We can compile and test the application after any or all such moves. At the end, shared data is allocated appropriately, and all `tainted` values should be validated—no instances of `unsafeUnverified()` should remain.

Step 4 (enabling enforcement). Our final task is to replace the `None` sandbox with one that enforces strong isolation. To start, we remove the statically-linked libjpeg and change the sandbox type from `None` to `None_DynLib`. In contrast to the `None` sandbox, the `None_DynLib` sandbox dynamically loads libjpeg. Any remaining calls to libjpeg made without the `sandbox_invoke()` will fail with a symbol resolution error at compile time. We resolve these and finish by changing the sandbox type to `Process`, `NaCl` sandbox types that enforce isolation. We discuss these isolation mechanisms in more detail in Section 2.5.2.

2.5 Implementation

Our implementation consists of two components: (1) a C++ library that exposes the APIs that developers use when sandboxing a third-party library, and (2) two isolation mechanisms that offer different scaling-performance trade-offs. We describe both of these below, and also describe a third approach in Section 2.8.

2.5.1 RLBox C++ API and type system

The RLBox API is implemented largely as a pure C++ library. This library consists of functions like `sandbox_invoke()` that are used to safely transfer control between the application and library. These functions return `tainted` values and can only be called with `tainted` values or primitives. The library's wrapped types (e.g., `tainted<T>`) are used to ensure dataflow safety (e.g., when using a value returned by `sandbox_invoke()`). Since the implementation of the control flow functions is mostly standard, we focus on our implementation of tainted values.

The `tainted<T>` wrapper. We implement `tainted<T>` as a simple wrapper that preserves the memory layout of the unwrapped `T` value, i.e., `tainted<int>` is essentially `struct tainted<int> { int val; }`. The only distinction between tainted and untainted values is at the type-level. In particular, we define methods and operators on `tainted<T>` values that (1) ensure that tainted values cannot be used where untainted values are expected (e.g., branch conditions) without validation and (2) allow certain computations on tainted data by ensuring their results are themselves tainted.

In general, we cannot prevent developers from deliberately abusing unsafe C++ constructs (e.g., **`reinterpret_cast`**) to circumvent our wrappers. Our implementation, however, guards against common C++ design patterns that could inadvertently break our `tainted` abstraction. For example, we represent tainted pointers as `tainted<T*>` and not `tainted<T> *`. This ensures that developers cannot write code that inadvertently unwraps `tainted` pointers via pointer decay—

since all C++ pointers can decay to **void***. We also use template meta-programming and SFINAE to express more complex type-level policies. For example, we disallow calls to `verify()` on pointer types and ensure that callback functions have wrapped all parameters in `tainted`.

Operators on `tainted` data. For flexibility, we define several operators on `tainted` types. Operations which are always safe, such as the assignment (**operator**=) of a `tainted<`**int**`>`, simply forward the operation to the wrapped **int**. Other operators, however, require return types to be `tainted`. Still others require runtime checks. We give a few illustrative examples.

- *Wrapping returned values:* We allow arithmetic operators (e.g., **operator**+) on, say, `tainted<`**int**`>`s, or a `tainted<`**int**`>` and an untainted **int**, but wrap the return value.
- *Runtime checks:* We allow array indexing with both **int**s and `tainted<`**int**`>`s by defining a custom array indexing operator **operator**`[]`. This operator performs a runtime check to ensure the array indexing is within sandbox memory.
- *Pointer swizzling:* We also allow operations such as **operator**=, **operator***, and **operator**-> on tainted pointers, but ensure that the operators account for swizzling (in addition to the runtime bounds check) when performing these operations. As with the others, these operators return `tainted` values. In Appendix 2.10.1, we describe the subtle details of type-driven automatic pointer swizzling.

Wrapped structs. Our library can automatically wrap primitive types, pointer types, function pointers, and static array types. It cannot, however, wrap arbitrary user-defined structs without some added boilerplate definitions. This is because C++ (as of C++17) does not yet support reflection on struct field *names*. We thus built a ~100LOC Clang plugin that automatically generates a header file containing the required boilerplate for all struct types defined in the application source.

Other wrapped types. In addition to the `tainted` wrapper type, RLBox also relies on several other types for both safety and convenience. As an example of safety, our framework

distinguishes registered callbacks from other function pointers, at the type level. In particular, `sandbox_callback` returns values of type `callback<...>`. This allows us to ensure that functions that expect callbacks as arguments can in fact only be called with callbacks that have been registered with `sandbox_callback`. As an example of convenience, RLBox provides RAII types such as `stack_arr<T>` and `heap_arr<T>` which minimize boilerplate. With these types, developers can for instance invoke a function with an inline string: `sandbox_invoke(sbox, png_error, stack_arr("..."))`.

2.5.2 Efficient isolation mechanisms

The RLBox API provides a plugin approach to support different, low-level sandboxing mechanisms. We describe two sandboxing mechanisms which allow portable and efficient solutions for isolating libraries in this section. In Section 2.8 we describe a third mechanism, based on WebAssembly, that we recently integrated in production Firefox.

The first mechanism uses software-based fault isolation (SFI) [WLAG93] extending Google's Native Client (NaCl) [YSD+09, SMB+10], while the second uses OS processes with a combination of mutexes and spinlocks to achieve performance. These approaches and trade-offs are described in detail below.

SFI using NaCl. SFI uses inline dynamic checks to restrict the memory addressable by a library to a subset of the address space, in effect isolating a library from the rest of an application within a single process. SFI scales to many sandboxes, has predictable performance, and incurs low overhead for context switching (as isolation occurs within a single process). The low context-switch overhead (about 10× a normal function call) is critical for the performance of streaming libraries such as libjpeg, which tend to have frequent control transfers. We explore this in more detail later (§2.6).

To support library sandboxing with SFI, we extend the NaCl compiler toolchain [YSD+09, SMB+10]. NaCl was originally designed for sandboxing mobile code in the browser, not library

sandboxing. Hence, we made significant changes to the compiler, optimization passes, ELF loader, machine model and runtime; we give a thorough description of these changes in Appendix 2.10.2. To ensure that our changes do not affect security, we always verify the code produced by our toolchain with the *unmodified* NaCl binary code verifier.

We use our modified NaCl compiler toolchain to compile libraries like libjpeg along with a custom runtime component. This runtime component provides symbol resolution and facilitates communication with the renderer.

Process sandboxing. Process sandboxing works by isolating a library in a separate *sandbox process* whose access to the system call interface is restricted using seccomp-bpf [Moz18b]. We use shared memory between the two processes to pass function arguments and to allocate shared objects. Compared to SFI, process sandboxing is simpler and does not need custom compiler toolchains. When used carefully, it can even provide performance comparable to SFI (§2.6).

As with SFI, process sandboxing also includes a custom runtime that handles communication between the library and renderer. Unlike SFI, though, this communication is a control transfer that requires inter-process synchronization. Unfortunately, using a standard synchronization mechanism—notably, condition variables—is not practical: a simple cross-process function is over 300× slower than a normal function call.

Our process sandbox uses both spinlocks and condition variables, allowing users to switch between to address application needs. Spinlocks offer low control-transfer latency (20× a normal function call), at the cost of contention and thus scalability. Condition variables have higher latency (over 300× a normal function call), but minimize contention and are thus more scalable. In the next section we detail our Firefox integration and describe how we switch between these two process sandboxing modes.

2.5.3 Integrating RLBox with Firefox

To use the SFI or Process sandbox mechanisms efficiently in Firefox, we must make several policy decisions about when to create and destroy sandboxes, how many sandboxes to keep alive, and for the Process sandbox when to switch synchronization modes. We describe these below.

Creating and destroying sandboxes. We build Firefox with RLBox sandboxing web page decompression, image decoding, and audio and video playback. We apply a simple policy of creating a sandbox on demand—a fresh sandbox is required when decoding a resource with a unique `<renderer, library, content-origin, content-type>` as discussed in Section 2.1. We lazily destroy unused sandboxes once we exceed a fixed threshold. We determine this threshold experimentally. Most webpages have a large number of compressed media files as compared to the number of images (§2.6.1). Since we can only reasonably scale to 250 sandboxes (§2.6.5), we conservatively use a threshold 10 sandboxes for JPEG and PNG image decoding, and 50 sandboxes for webpage decompression. Browsers do not typically play multiple audio or video content simultaneously—we thus simply create a fresh sandbox for each audio and video file that must be decoded and destroy the sandbox immediately after.

Switching synchronization modes. For the Process sandbox, we switch between spinlocks and conditional variables according to two policies. First, we use spinlocks when the renderer performs a latency sensitive task, such as decoding an image using a series of synchronous libjpeg function calls and callbacks. But, when the renderer requests more input data, we switch to the condition variables; spinlocks would create needless CPU contention while waiting for this data (often from the network). Second, for large media decoding such as 4K images, we use condition variables: each function call to the sandbox process takes a large amount of time (relative to the context switch) to perform part of the decoding. Though we can use more complex policies (e.g., that take load into effect), we find these two simple policies to perform relatively well (§2.6).

Leveraging multiple cores. Since 95% of devices that run Firefox have more than 1 core, we can use multiple cores to optimize our sandbox performance.[7] In particular, our process sandbox implementation pins the sandboxed process on a separate CPU core from the renderer process to avoid unnecessary context switches between the renderer and sandboxed process. This is particularly important when using spinlocks since the sandbox process's spinlock takes CPU cycles even when "not in use"; pinning this process to a separate core ensures that the sandbox process does not degrade the renderer performance. In our evaluation (§2.6), we reserve one of the system's cores exclusively for the processes created by our sandboxing mechanism, when comparing against the stock and SFI builds (which use all cores for the renderer process' threads).

2.6 Evaluation

We present the following results:

- *Cross origin resources that could compromise the renderer are pervasive*, and could even be used to compromise websites like Gmail. We present measurements of their prevalence in the Alexa top 500 websites in section 2.6.1.
- *Library sandboxing overheads are modest*: section 2.6.2 breaks down the individual sources of sandboxing overhead, section 2.6.4, shows end-to-end page latencies and memory overheads for popular websites are modest with both isolation mechanisms(§2.5.2)– even on media heavy websites, section 2.6.5, shows that the CPU overhead of web page decompression and image decoding in the renderer process are modest, and that CPU and memory overheads scale well up to our current maximum of 250 concurrent sandboxes.
- *Migrating a library into RLBox typically takes a few days with modest effort*, as shown in Section 2.6.3.

[7] See https://data.firefox.com/dashboard/hardware, last visited May 15, 2019.

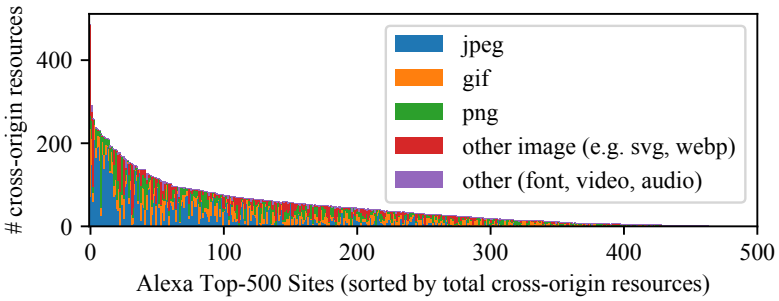


Figure 2.2: Cross-origin resource inclusion in the Alexa Top-500.

- *RLBox is broadly useful for library sandboxing beyond Firefox*, as demonstrated in 2.6.6, where we discuss our experience apply RLBox to sandboxing native libraries in Apache and Node.js modules.

Machine Setup. All benchmarks run on an Intel i7-6700K (4 GHz) machine with 64 GB of RAM, and hyperthreading disabled, running 64-bit Ubuntu 18.04.1. Firefox builds run pinned on two isolated CPU cores (i.e., no other process is allowed to run on these CPUs), to reduce noise. As discussed in Section 2.5.3, with the process sandbox build, the renderer process is pinned to one core, while the sandbox process uses the other core.

2.6.1 Cross-origin content inclusion

To evaluate how often web sites include content (e.g., images, audio, and videos) cross-origin, we crawled the Alexa top 500 websites. Our crawler—a simple Firefox extension—logs all cross-origin resource requests made by the website for 10 seconds after page load (allowing some dynamic content to be loaded). Figure 2.2 shows our measurements, categorized by the resource MIME type.

We find that 93% of the sites load at least one cross-origin media resource (primarily images), with mean of 48 and median of 30, cross-origin media resources loaded. Many of the resource loads (median 35 and mean 17) are not just cross-origin but also *cross-site*. In the presence of media parsing library bugs, such loads would undermine Site Isolation protections.

The pervasive use of cross-origin resources inclusion indicates that sandboxing libraries at the `<renderer, library, content-origin, content-type>` granularity can significantly reduce the renderer attack surface. Although not all cross-origin content is necessarily untrusted, the origin is nevertheless an important trust boundary in practice—and many websites do consider cross-origin media untrusted. For instance, Google allows users to freely upload content to `sites.google.com`, but serves such media content from `googleusercontent.com`. Google even re-encodes images, another sign that such images are untrusted.

Unfortunately, they do not re-encode video or audio files. To test this, we created a page on `sites.google.com` in which we embedded both the VPX video proof of concept exploit of CVE-2015-4506 [Bug15] and the OGG audio proof of concept exploit of CVE-2018-5148 [Bug18]. In both cases the files were unmodified. For VPX, we modified Firefox and Chrome (with Site Isolation) to re-introduce the VPX bug and visited our malicious website: in both browsers the video successfully triggered the bug.

We found we could include such malicious content as part of an email to a Gmail address. Gmail re-encodes images, but does not re-encode the video and audio files. The Gmail preview feature even allows us to play the audio track—which surprisingly, we found was hosted on `mail.google.com`.

2.6.2 Baseline RLBox overhead

To understand the overhead of different parts of the RLBox framework in isolation we perform several micro-benchmarks.

Sandbox creation overhead. The overhead of sandbox creation is ≈1ms for SFI sandboxes and ≈2ms for process sandboxes. These overheads do not affect page latency for any realistic website, and can be hidden by pre-allocating sandboxes in a pool. For this reason, we don't include sandbox creation time in the remaining measurements.

Control transfer overhead. To understand the overhead of a control transfer, we measure the

elapsed time for an empty function call with different isolation mechanisms. For reference, an empty function call without sandboxing takes about 0.02μs in our setup. With sandboxing, we measured this to be 0.22μs for SFI sandboxes, 0.47μs for Process sandboxes using spinlocks, and 7.4μs for Process sandboxes using conditional variables. SFI and Process sandboxes using spinlocks are an order of magnitude slower than a normal function call, but over 10$\times$ faster than Processes using condition variables, and are thus better suited for workloads with frequent control transfers.

Overhead of RLBox dynamic checks. The RLBox API introduces small overheads with its safety checks (e.g., pointer bounds checks, swizzling, and data validation functions). To measure these, we compare the overhead of rendering `.jpeg` and `.png` images on Firefox with sandboxed libraries with and without RLBox enabled. We find the difference to be negligible ($< 1\%$). This is unsurprising: most of RLBox's checks are static and our dynamic checks are lightweight masks.

Overhead of SFI dynamic checks. Unlike process-based sandboxing, SFI incurs a baseline overhead (e.g., due to inserted dynamic checks, padding etc.) [YSD+09]. To understand the overhead of our NaCl SFI implementation, we implement a small `.jpeg` image decoding program and measure its slowdown when using a sandboxed libjpeg. We find the overhead to be roughly 22%.

2.6.3 Migrating Firefox to use RLBox

In this section, we evaluate the developer effort required to migrate Firefox to use sandboxed libraries. In particular, we report the manual effort required by RLBox developers and the manual effort saved by using the RLBox API. Table 2.1 gives a breakdown of the effort. On average, we find that it takes a bit over two days to sandbox a library with RLBox and roughly 180 LOC (25% code increase); much of this effort is mechanical (following Section 2.4).

Table 2.1 also shows the tasks that RLBox automates away. First, RLBox eliminates 607 lines of glue-code needed by both the Process and SFI sandboxes to marshal function parameters

Table 2.1: Manual effort required to retrofit Firefox with fine grain isolation, including the effort saved by RLBox's automation. We do not report the number of days it took to port the JPEG and PNG decoders since we ported them in sync with building RLBox.

	Task	JPEG Decoder	PNG Decoder	GZIP Decompress	Theora Decoder	VPX Decoder	OGG-Vorbis Decoder
Effort saved by RLBox automation	Generated marshaling code	133 LOC	278 LOC	38 LOC	39 LOC	60 LOC	59 LOC
	Automatic pointer swizzles for function calls	30	96	5	36	46	34
	Automatic nested pointer swizzles	17	5	6	8	9	5
	Automatic pointer bounds checks	64 checks	25 checks	8 checks	12 checks	15 checks	14 checks
	Number of validator sites found	28	51	10	5	2	4
Manual effort	Number of person-days porting to RLBox	–	–	1 day	3 days	3 days	2 days
	Application LOC before/after port	720 / 1058	847 / 1317	649 / 757	220 / 297	286 / 368	328 / 395
	Number of unique validators needed	11	14	3	3	2	2
	Average LOC of validators	3 LOC	4 LOC	2 LOC	3 LOC	2 LOC	2 LOC

and return values for each cross-boundary function call; RLBox automatically generates this boilerplate through C++ templates and meta-programming. Second, RLBox automatically swizzles pointers. This is necessary for any sandbox functions or callbacks that accept pointers; it's also necessary when handling data-structures with nested pointers that are shared between the application and the sandbox. This is a particularly challenging task without RLBox, as manually identifying the 297 locations where the application interacts with such pointers would have been tedious and error-prone. Third, RLBox automatically performs bounds checks (§2.3.2); the number of required pointer bounds checks that were automatically performed by RLBox are again in the hundreds (138).

Finally, RLBox identifies the (100) sites where we must validate tainted data (§2.3.3). Though RLBox cannot automate the validators, we find that we only need 35 unique validators—all less than 4 lines of code. In practice, we found this to be the hardest part of migration since it requires understanding the domain-specific invariants.

2.6.4 RLBox overhead in Firefox

We report the end-to-end overheads of Firefox with library sandboxing by measuring page latencies of webpages, memory overheads in Firefox as well as audio video playback rates.

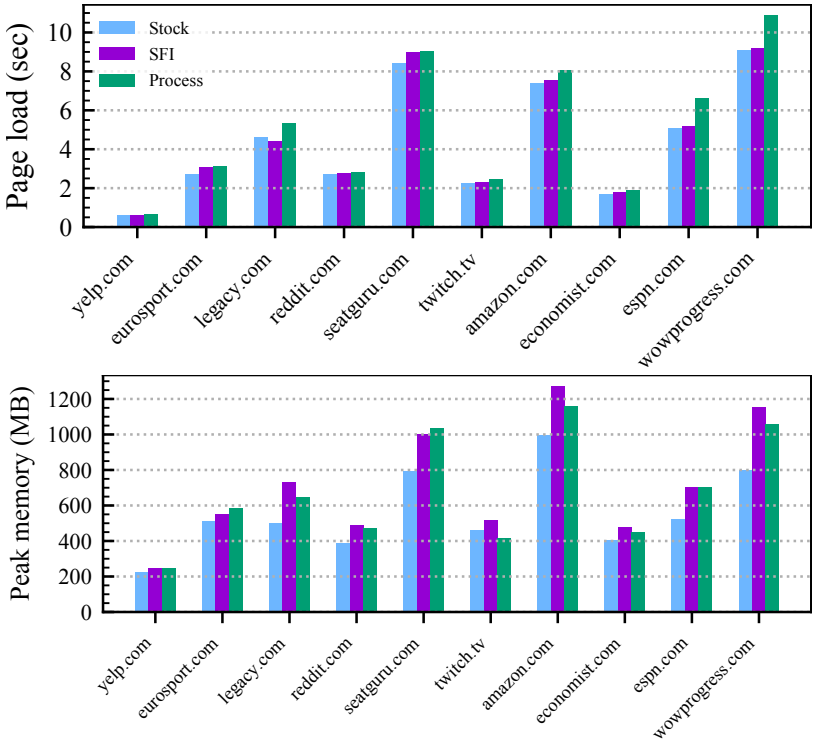


Figure 2.3: Impact of sandboxing on page load latencies and *peak* memory usage overheads. Firefox with SFI sandboxes incurs a 3% page latency and 25% memory overhead while Firefox with process isolation incurs a 13% page latency and a 18% memory overhead.

Experimental setup. We evaluate end-to-end performance with six sandboxed libraries: libjpeg-turbo 1.4.3, libpng 1.6.3, zlib 1.2.11, libvpx 1.6.1, libtheora 1.2, and libvorbis as used in Firefox 57.0.4. We report performance for two Firefox builds that use libraries sandboxed with SFI and Process mechanisms respectively. Both builds create fresh sandboxes for each `<renderer, library, origin, content-type>` combination as described in §2.1. We measure impact on page load times for both these builds.

End-to-end impact on real-world websites

Benchmark. We report the overhead of page load latency and memory overheads in Firefox with the six sandboxed libraries by measuring latencies of the 11 websites used to measure the overhead of Site Isolation [RMO19]. These websites are a representative sample of both popular

and slightly-less-popular websites, and many of them make heavy use of media resources. We measure page load latency using Firefox's Talos test harness [Moz18a]. We measure memory overheads with `cgmemtime` [Sau12]—in particular, the peak resident memory and cache usage during a run of Firefox. We run the test 10 times and report the median values of page latency and memory overheads.

Results. As shown in Figure 2.3, the page latency and CPU utilization overheads are modest. Our SFI build incurs a 3% overhead in page latency while the Process sandbox incurs an overhead of 13%. As a comparison point, the overhead of naively using process sandboxes (using only conditional variables without any CPU pinning) incurs an overhead of 167%.

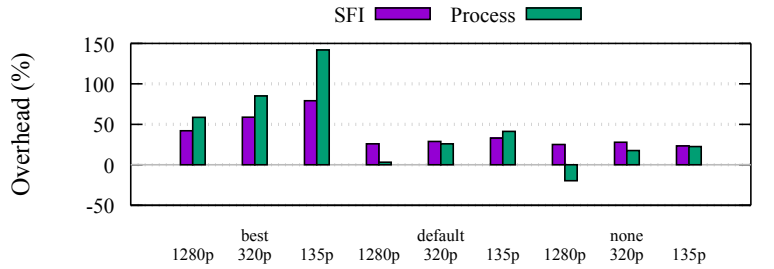
We find the average peak renderer memory overhead to be 25% and 18% for the SFI and Process sandboxes, respectively. This overhead is modest and, more importantly, transient: we can destroy a sandbox after the media has been decoded during page load.

Sandboxing video and audio decoding

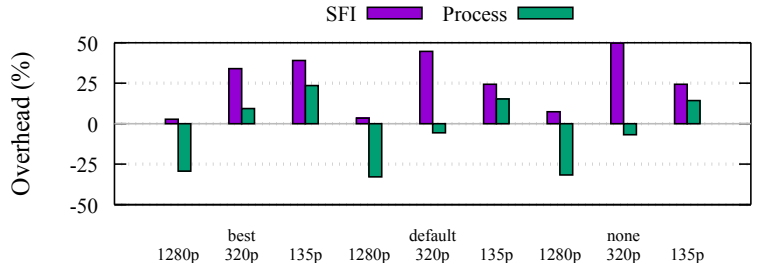
To understand the performance overhead of RLBox on video and audio decoding, we measure the performance of Firefox when decoding media from two video formats (Theora and VPX) and one audio format (OGG-Vorbis).

Benchmark. Our benchmark measures decoding performance on the Sintel sequence in Xiph.org's media test suite[8]. As this sequence is saved as lossless video, we setup the benchmark by first converting this to ultra HD videos of 4K resolution in the Theora and VP9 formats; we similarly convert the lossless audio to a high resolution OGG-vorbis audio file with a sample rate of 96 kHz and a bit rate of 500 Kb/s using the suggested settings for these formats [the19, vp919]. Our benchmark then measures the frame-rate and bit-rate of the video and audio playback—by instrumenting the decoders with timers—when using the sandboxed libraries. We run the test 5 times and report the median values of frame-rate and bit-rate.

[8]Online: https://media.xiph.org/. Last visited November 15, 2019.



(a) JPEG rendering overhead



(b) PNG rendering overhead

Figure 2.4: Per-image decoding overhead for images at 3 compression levels and 3 resolutions, normalized against stock Firefox. For most JPEGs, the SFI and Process sandbox overhead is 23-32%, and $< 41\%$ respectively. For PNGs, the SFI and Process sandbox overhead is 2-49% and $< 15\%$ respectively. We also observe some negative overheads and pathological cases where overhead is as high as 140%. The reasons for this are explained in Section 2.6.5.

Results. We find that neither the SFI nor the Process sandboxing mechanism visibly degrades performance. In particular, our sandboxed Firefox are able to maintain the same frame-rate (24 fps for the VPX video and 60 fps for the Theora video) and bit-rate (478 bits per second) as stock Firefox for these media files.

2.6.5 Microbenchmarks of RLBox in Firefox

To understand the performance impact of RLBox on the different libraries, we perform several microbenchmarks that specifically measure the impact of sandboxing webpage decompression, image decoding and sandbox scaling in Firefox.

Sandboxing webpage decompression

Firefox uses zlib to decompress webpages. Since webpage decompression is done entirely before the page is rendered, we report the overhead of sandboxing zlib by measuring the slowdown in page load time.

Benchmark. We create a webpage whose HTML content (excluding media and scripts) is 1.8 MB, the size of an average web page[9], and measure the page load time with Talos. We use the median page load time from 1000 runs of this test.

Results. For both SFI and Process sandboxing mechanisms, the overhead of sandboxing zlib is under 1%. In other words, the overhead of sandboxing zlib is largely offset by other computations needed to render a page.

Sandboxing image decoding

To understand performance impact of sandboxing on image rendering, we measure per-image execution time for the `.jpeg` and `.png` decoders, with different forms of sandboxing, and compare our results to stock Firefox. Decoder execution time is a better metric for image rendering performance than page load time because Firefox decodes and renders images asynchronously; the usual test harness would notify us that a page has been loaded before images have actually been decoded and displayed in full—and this might be visible to the user.

Benchmarks. We use the open Image Compression benchmark suite[10] to measure sandboxed image decoding overheads. To capture the full range of possibilities for performance overhead, we measure overheads of images at 3 sizes (135p, 320p, and 1280p) and three compression levels (best, default, and none) for each image in the benchmark suite. We run this test 4000 times for each image and compare the median decoder code execution times.

[9] See the HTTP Archive page weight report, https://httparchive.org/reports/page-weight. Last visited May 15, 2019.

[10] Online: https://imagecompression.info/test_images/. Last visited May 15, 2019.

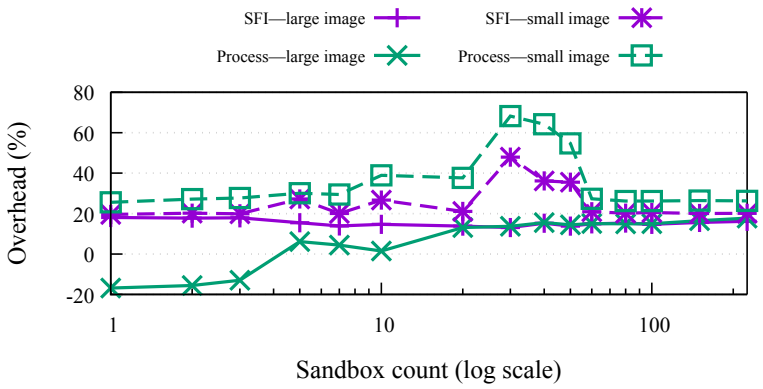


Figure 2.5: Performance overhead of image decoding with increasing the number of sandboxes (each image is rendered in a fresh sandbox).

Results. Since all images in the suite produce similar results, we give the results of one 8-bit image in Figure 2.4. We start with three high-level observations. First, both SFI and Process based sandboxes have reasonable overheads—23-32% and $< 41\%$ respectively for most JPEGs, 2-49% and $< 15\%$ respectively for PNGs. Second, Process sandbox sometimes has negative overheads. This is because the Process sandbox dedicates one of the two available cores exclusively for execution of sandboxed code (§2.5.3) including the sandboxed image rendering code, while the stock and SFI Firefox builds use all cores evenly. Third, for JPEGs at the best compression, the overhead relative to stock Firefox is high—roughly 80% for SFI and 140% for Process sandboxes. This is because decoding high compression images have low absolute decode times (~650μs), and thus have larger overheads as control transfer overheads between Firefox and the sandbox image libraries cost are not effectively amortized. However, in absolute terms, the differences are less than 1.5ms and have no impact on end-user experience.

Sandbox scaling characteristics

Web pages often contain tens of images of different types from multiple origins. Thus, the scaling properties of different isolation mechanisms are an important consideration.

Benchmark. We evaluate sandbox scaling by rendering pages with an increasing number of JPEG

images from unique origins. Each image thus creates a new sandbox which incurs both CPU and memory costs. CPU costs are measured by measuring the total amount of time executing image decoding functions. We measure memory overhead as before, but don't destroy any sandbox; this allows us to estimate the worst case scenario where memory usage is *not* transient. As before (§2.6.5), we measure the decoder execution time for 4000 image loads at each scale, and report the median overhead.

Results. Figure 2.5 shows the CPU overhead of image rendering as we increase the number of sandboxes for both large (1280p) and small (135p) JPEG images using default compression. This experiment allows us to make several observations. We can run up to 250 concurrent SFI sandboxes before we run into limitations like exhausting pre-allocated thread local storage or finding aligned free virtual memory. These limitations can be overcome with more engineering effort. We never came close to these limits browsing real websites, including those of Section 2.6.4. Both the SFI and the Process sandbox similarly scale well on both small and large images, with CPU overheads between 20% and 40% for most sandbox counts. The process sandbox, however, scales only because we use multiple synchronization modes described (§2.5).

Extra sandboxes add memory overhead for two reasons. First, each sandbox uses a private copy of code (e.g., libjpeg and libc for each libjpeg sandbox). Second, each sandbox has its own stack and heap. In this experiment, we observed that memory consumption increases linearly with the number of images (which corresponds to the number of sandboxes created). On average, an SFI sandbox consumes 1.6 MB, while Process sandboxing consumes 2.4 MB for each sandbox. Several optimizations to reduce memory consumption exist that we have not yet implemented. For example, the SFI sandbox currently loads a fresh copy of the code for each sandbox instance. We could optimize this by sharing code pages between sandboxes—and, indeed, we do this in production for our Wasm sandbox.

2.6.6 RLBox outside Firefox

RLBox is a general-purpose sandboxing framework that can be used in any C++ application. To demonstrate its utility beyond Firefox, we applied it in two different contexts: the Apache web server and Node.js runtime.

Apache allows developers to write C modules that extend its base functionality. These modules often depend on third-party libraries. For example, the mod_markdown [mod11] module uses the libmarkdown library to transform Markdown to HTML on the fly to support serving Markdown files.

To protect Apache from bugs in libmarkdown we modify mod_markdown to run libmarkdown in an RLBox SFI sandbox. The change required a single person-day, and added or modified roughly 300 lines of code. We measured the average and tail latency as well as throughput of the webserver using the autocannon 4.4.0 benchmarking tool [aut16] with default configurations (1 minute runtime, 10 parallel connections) serving a 16K markdown file. The unmodified webserver's average latency, tail latency and throughput were 10.5ms, 36ms and 940 requests/second, respectively; the sandboxed server's average latency, tail latency and throughput were 14ms, 40ms and 684 requests/second. Though the average latency and throughput overhead is modest, we observe that the tail latency—arguably the most important metric—is within 10% of baseline.

Node.js is a JavaScript runtime system written in C++, largely used for web applications. Like Apache, Node.js allows developers to expose new functionality implemented in native plugins to the JavaScript code. For example, the bcrypt [nod10] password hashing library relies on native code—indeed the JavaScript code largely wraps Provos' C bcrypt library. To protect the runtime from memory-safety bugs in the C library, we modify the C++ code in bcrypt to run the bcrypt C library in an RLBox SFI sandbox—a change that required roughly 2 person hours, adding or modifying 75 lines of code. We measured—using the benchmark.js library—the overhead in average hashing throughput (hashing a random 32-byte password) to be modest: 27%.

2.7 Related work

Isolation in the browser. Modern browsers since Chrome [BJRt08] rely on coarse grain *privilege separation* [PFH03] to prevent browser compromises from impacting the local OS [RG09]. However, a compromised renderer process can still use any credentials the browser has for other sites, enabling extremely powerful *universal cross-site scripting* (UXSS) attacks [DPF06].

In response to UXSS attacks and recent Spectre attacks, Chrome introduced *Site Isolation* [RMO19]. Site Isolation puts pages and iframes of different sites into separate processes. Unfortunately, as discussed in Section 2, this does not prevent UXSS attacks across related sites (e.g., `mail.google.com` and `pay.google.com`). Firefox's Project Fission [Moz19] proposes to go further and isolate at the origin boundary, similar to previous research browsers [GTK11, WGM+09, SKMT13], but this still does not protect the renderer when loading cross-origin resources such as images.

An unpublished prototype using SFI called MinSFI [Sea13] was developed at Google in 2013 to protect the Chrome renderer from compromise of zlib library; however, it was missing several features necessary for compatibility and efficiency, including threading and callback support. Additionally, the project was primarily focused on improving the efficiency of SFI rather than the integration challenges tackled by RLBox including handling `tainted` data, migration of code bases, etc.

In some parts of the renderer, there is no substitute for strong memory safety to prevent attacks. Servo [ABG+16] is an ongoing project to rewrite much of the Firefox rendering stack in Rust. However, for the foreseeable future, Firefox and other browsers will continue to rely on libraries written in C/++. This makes sandboxing the most viable approach to containing vulnerabilities.

Sandboxing. There has been some related work on providing APIs to simplify using sandboxed libraries (e.g., Codejail [WSYL12] and Google Sandboxing APIs [BSI19]). However, these

efforts do not provide the type-driven automation of RLBox (e.g., pointer swizzling and migration assistance) nor the safety of `tainted` types—leaving developers to manually deal with attacks of Section 2.2. Sammler et al. [SGDL19] formal model addresses some of these attacks using a type-direct approach, but require applications to be formally verified correct (in contrast to our local validators) to give meaningful guarantees.

There is a long line of work on sandboxing mechanisms with different performance trade-offs [WLAG93, SMB+10, Sea13, HRS+17, VOED+19]. Recent, excellent surveys [Tan17, SWG+16] present a comprehensive overview of these mechanisms. RLBox makes it easy for developers to use such mechanisms without modifying the application or library (§2.5.2). In production we use WebAssembly; WebAssembly stands out as a principled approach with wide adoption [HRS+17].

Data sanitization and leaks. There is a large body of work on static and dynamic approaches to preventing or mitigating missed sanitization errors; see the survey by Song et al. [SLR+19]. These tools are orthogonal to RLBox. Developers could use them to check their data validation functions for bugs.

DUI Detector [HCLS15] uses runtime trace analysis to identify missing pointer sanitizations in user code. Other work has looked at sanitizing user pointers in the kernel [BA08]. For example, type annotations [JW04] have been used to distinguish between untrusted user pointers and trusted pointers in OS kernel code. In contrast, RLBox automatically applies such pointer sanitizations by leveraging the C++ type system.

One approach to avoid double-fetch bugs is to marshal all shared data before using it. But, this comes at a cost. Marshaling tools and APIs typically require array bounds annotations, which is tedious and demands in-depth knowledge of the sandboxed library's internal data structures. Automatic marshaling tools like C-strider [SHF16] and PtrSplit [LTJ17] address this limitation; however, these tools either impose a significant overhead or lack support for multi-threading. RLBox uses shared memory and statically enforces that shared data is placed in shared memory,

avoiding the need for custom marshaling tools or annotations. The use of shared memory, however, introduces possible double-fetch bugs. While RLBox provides APIs to assist with double-fetches, the possibility of unhandled double-fetch bugs still remain. Several recent techniques detect double-fetches from shared memory [WKL+17, XQL+18, SGL+18] and can be used alongside RLBox.

Previous efforts have also sought to prevent leaking pointers that could compromise ASLR [LSL+15, CLH+15, BDL+16]. RLBox prevents pointer leaks by disallowing pointers to renderer memory to pass into the sandboxed library via the type system.

Porting assistance. Several privilege separation tools provide assistance when migrating to a sandboxed architecture. Wedge (Crowbar) [BMHK08] uses dynamic analysis to guide the migration, and also supports an incremental porting mode that disables isolation so that developers can test the partial port and identify the next step. SOAAP [GWA+15] uses code annotations and a custom compiler to guide the developer. PrivTrans [BS04] uses code annotations and automatic source code rewriting to separate code and data into separate components. In contrast, RLBox assists with porting without any custom tooling, purely through the use of compile-time errors, by identifying code that must be modified for security and shared data that must be migrated to sandbox memory (§2.4).

2.8 Using RLBox in production

Over the last 6 months we've been integrating RLBox into production Firefox. In this section, we describe the difference between our research prototype and the production RLBox, and our migration of the libGraphite font shaping library to use RLBox. We are in the process of migrating several other libraries and adding support for Firefox on Windows [Fro20, Shr].

2.8.1 Making RLBox production-ready

To make RLBox production-ready we adapt a new isolation mechanism based on WebAssembly and rewrite the RLBox API, using our prototype implementation as a reference.

SFI using WebAssembly. In production, we use Wasm instead of NaCl to isolate library code from the rest of Firefox within a single process. Though Wasm's performance and feature-set is not yet on par with NaCl's [JPBG19], NaCl has been deprecated in favor of Wasm [Goo17] and maintaining a separate toolchain for library sandboxing is prohibitive. Moreover, these limitations are likely to disappear soon: as part of the Bytecode Alliance, multiple companies are working together to build robust Wasm compilation toolchains, a standard syscall interface, SIMD extensions, etc.

Since our goal is to reap the benefits of these efforts, we need to minimize the changes to these toolchains. In particular, this means that we cannot adjust for differences between the Firefox and Wasm machine model as we did for NaCl [NGL+19]—by intrusively modifying the compiler, loader, runtime, etc. (§2.5.2). We, instead, take advantage of the fact that RLBox intercepts all data and control flow to automatically translate between the Firefox and Wasm machine models in the RLBox API.

Our only modification to the Lucet Wasm runtime is an optimized trampoline, which we are working on upstreaming [zer19]. Since Wasm is well-typed and has deterministic semantics, our trampolines safely eliminate the context-and stack-switching code, reducing the cost of a cross-boundary crossing to a function call. This optimization was key to our shipping the sandboxed libGraphite (§2.4)—it reduced the overhead of RLBox by 800%. The details and formalization of these *zero-cost trampolines* will be presented in a separate paper.

Meaningful migration error-messages. We re-implement RLBox in C++ 17 and use new features—in particular `if constexpr`—to customize the error messages that guide developers during migration (§2.4). Meaningful error messages (as opposed to a wall of generic, template failures) is key to making RLBox usable to other developers. Although implementing custom

error messages in our C++ 11 prototype is possible, it would make the implementation drastically more complex; C++ 17 allows us to keep the RLBox API implementation concise (under 3K lines of code) and give meaningful error messages.

2.8.2 Isolating libGraphite

We use RLBox to isolate the libGraphite font shaping library, creating a fresh sandbox for each Graphite font instance. We choose libGraphite largely because the Graphite fonts are not widely used on the web, but nevertheless Firefox needs to support it for web compatibility. This means that the library is part of Firefox attack surface—and thus memory safety bugs in libGraphite are security vulnerabilities in Firefox [Ste17, Yve16].

Evaluation To measure the overhead of our sandboxing, we use a micro-benchmark that measures the page render time when reflowing text in a Graphite font ten times, adjusting the font size each time, so font caches aren't used.[11] We find that Wasm sandboxing imposes a 85% overhead on the libGraphite code, which in turn slows down Firefox's font rendering component (which uses libGraphite internally) by 50%. We attribute this slowdown largely to the nascent Wasm toolchains, which don't yet support performance optimization on par with, say LLVM [JPBG19, Han19b]. Nevertheless, this overhead is not user-perceptible; in practice page rendering is slowed down due to the network and heavy media content, not fonts.

To measure memory overhead, we use `cgmemtime` to capture the peak resident memory and cache used by Firefox on the same micro-benchmark. We find the memory overhead to be negligible—the median peak memory overhead when loading the micro-benchmark ten times is 0.68% (peak memory use went from 431460 KB to 434426 KB).

Deployment. The rewritten RLBox library as well as the modifications to Firefox to use a sandbox libGraphite have been merged into the Firefox code base [Shr]. Our retrofitted Firefox

[11] Available at: https://jfkthame.github.io/test/udhr_urd.html.

successfully tested on both the Firefox Nightly and Beta channels, and ships in stock Firefox 74 to Linux users and in Firefox 75 to Mac users [Fro20].

2.9 Conclusion

Third party libraries are likely to remain a significant source of critical browser vulnerabilities. Our approach to sandboxing code at the library-renderer interface offers a practical path to mitigating this threat in Firefox, and other browsers as well.

RLBox shows how a type-driven approach can significantly ease the burden of securely sandboxing libraries in existing code, through a combination of static information flow enforcement, dynamic checks, and validations. RLBox is not dependent on Firefox and is useful as a general purpose sandboxing framework for other C++ applications.

2.10 Appendix

2.10.1 Automated Pointer Swizzling

Applications represent pointers as full 64-bit addresses (on 64-bit systems), however sandboxes such as the NaCl and Wasm SFI sandboxes, represents pointers as 32-bit offsets from the sandbox base address. Thus pointers must be swizzled to share data structures between the application and sandboxed library.

The `tainted_volatile` type. To automate pointer swizzling, RLBox internally adds an additional type, `tainted_volatile`, that it uses when dereferencing a `tainted` pointer. Like `tainted` wrappers, the `tainted_volatile` wrapper also refers to data from the sandboxed library, with the distinction that `tainted_volatile` data is stored in sandbox memory (memory accessible by sandboxed code) rather than application memory. These `tainted_volatile` types are primarily used as `rvalue` types and automatically convert into `tainted` types when copied into application mem-

ory, meaning that the programmer would never create a variable of `tainted_volatile` type and need not know of the type's existence. Distinguishing between `tainted` and `tainted_volatile` types allow RLBox to automate swizzling. By inspecting types, RLBox can swizzle pointers appropriately any time we write `tainted` pointer to a `tainted_volatile` location or vice-versa.

Context free swizzling. A further complication of implementing pointer swizzling is that RLBox needs to swizzle pointers without knowledge of the sandbox to which data belongs to or the location of sandbox memory! For instance, consider the following program.

```
1 auto mRLBox = createSandbox(...);
2 tainted<int**> foo = sandbox_invoke(mRLBox, ...);
3 tainted<int*> bar = sandbox_invoke(mRLBox, ...);
4 *foo = bar;
5 bar = *foo;
```

On line 4, we write `bar` to the location pointed to by `foo`. However, as `foo` is a `tainted` variable, `foo` points to a location in sandbox memory. Thus dereferencing `foo` results in a `tainted_volatile` reference (which points to sandboxed memory). So we must swizzle `bar` from the application pointer representation to the sandboxed pointer representation, prior to updating `*foo`. The swizzling occurs in the overloaded = operator of the `tainted_volatile` type which in C++ has a fixed type signature and is only permitted to take a reference to the assigned value—it does not include a reference to `mRLBox`. The reverse problem occurs on line 5 where the expression on the right is a `tainted_volatile`, while the expression on the left is a `tainted`. An easy albeit ugly solution could use a specific API for dereferencing, i.e. replacing `*foo` with `dereference(foo, mRLBox)`, however, such approaches changes the natural syntax expected by a C++ programmer.

Implementing context free swizzling. RLBox instead implements swizzling without knowledge of the sandbox base address by leveraging the structure of pointer representations. This approach is applicable for any sandbox whose base memory is sufficiently aligned, as is the case in our SFI

implementation which is aligned to 4GB. Converting from an application pointer representation to sandboxed pointer representation required on line 4 can be easily performed by clearing the top 32-bits of pointer `bar` in the overloaded equality operator. However, the second case which is the conversion of a sandboxed pointer representation to an application pointer representation on line 5 is more challenging.

To solve this, we make the observation that we can successfully learn the sandbox base address by inspecting the first 32-bits of any *example pointer* which is (1) not null (2) is in the same sandbox as the pointer we wish to convert and (3) is stored in the application pointer format (as a 64-bit pointer). The implementation of swizzling below shows how to find such an *example pointer* below.

```
#define top32 0xFFFFFFFF00000000
#define bot32 0xFFFFFFFF
tainted<T>& operator=(tainted_volatile<T2>& rhs) {
  uintptr_t exampleAppPtr = &(rhs.val);
  uintptr_t base = exampleAppPtr & top32;
  this->val = base + (rhs.val & bot32);
  return *this;
}
```

Line 4 constructs an *example pointer* in the body of **`operator`**=, with the expression `&(rhs.field)`. This expression points to the same memory location as the variable `foo` on line 5 in the original program. Importantly, `foo` may be used as an example pointer as it is located in the same sandbox as `*foo`, is stored in the application pointer representation, and is not null as we successfully dereferenced it on line 5, prior to the call of the **`operator`**= also on line 5. This solution is also used to solve similar problems when supporting pointer arithmetic on `tainted` types.

2.10.2 Native Client Tooling Changes

NaCl was originally designed for sandboxing mobile code in the browser, rather than libraries. We thus had to make significant modifications to the compiler, optimization passes, ELF loader, machine model and runtime prior to it being usable for library sandboxing. We list these changes below.

Loader and Runtime. The NaCl loader does not support libraries. We thus modify the loader and runtime to save the ELF symbol table from when the library is loaded, to provide symbol resolution.

Threading. On 64-bit platforms, NaCl relies on each sandbox having unique OS threads due to some design choices in the use of thread-local storage. However, for library sandboxing, the same application thread may invoke code from different sandboxes; therefore we had to modify the NaCl to operate without this requirement.

Function Invocation. NaCl uses a "springboard"—a piece of code which enables memory isolation prior to starting sandbox code—to invoke `main` with no function parameters. For library sandboxing, we modify the springboard to allow arbitrary function invocations with any parameters. We implemented callback support using the default syscall mediation mechanism in NaCl to correctly redirect callbacks .

SIMD. NaCl custom compilation toolchain does not support handwritten SIMD and assembly instructions found in libraries such as libjpeg-turbo. To mitigate this, we modify the NASM assembler to assemble this hand-written assembly to object files that pass NaCl verifier checks. We measured that this allowing handwritten SIMD instructions provided a 48% speedup for the SFI based sandboxing on a simple benchmark rendering a libjpeg image.

Machine Model. The machine model of a platform dictates the size of various types such as **`int`**, **`long`**, or pointers. To maximize performance, NaCl uses a custom machine model—for example, NaCl uses 32-bit pointers, even in its 64-bit machine model. However, this can introduce

incompatibilities during library sandboxing when data structures are transferred between the application and library

To remedy this, we modified the NaCl tooling to ensure the machine model is consistent with the Firefox application code. This required making significant changes to the NaCl Clang toolchain, the NaCl instruction set in LLVM, NaCl's implementation of libc and the NaCl runtime. We implemented these changes carefully such that NaCl Verifier (the tool used to determine the safety of a NaCl binary) would not require any modifications, giving us confidence that our changes do not affect security.

The Wasm toolchains also exhibit similar differences in the machine model. Rather than relying on similar large changes in the Wasm compiler toolchain, we use RLBox feature to automatically adjust for machine model differences (§2.8) to ensure compatibility.

2.11 Chapter acknowledgments

Chapter 2, in full, is a reprint of the material as it appears in the USENIX Security Symposium 2020. Narayan, Shravan; Disselkoen, Craig; Garfinkel, Tal; Froyd, Nathan; Rahm, Eric; Lerner, Sorin; Shacham, Hovav; Stefan, Deian. USENIX, 2020. The dissertation author was the primary investigator and author of this material.

Chapter 3

VeriWasm: Hardening against sandboxing compiler bugs

WebAssembly (Wasm) is a modern, platform-independent bytecode that was originally designed to be embedded in the browser, and was therefore designed with isolation in mind. Ironically, the fact that Wasm's design was tied to the browser also unshackled it *from* the browser: different "embedding environments" are using Wasm as a general-purpose software-based fault isolation (SFI) mechanism.

For example, the Fastly and Cloudflare content-delivery networks (CDNs) use Wasm to isolate different tenants on their edge clouds [Var18, Byt20a]. They run many clients' Wasm code within a single process, using Wasm-based SFI to protect clients from each other—and to protect the embedding host process from clients. Any break in Wasm's isolation guarantees could allow a malicious client to steal or corrupt data that belongs to another client (or the host). Similarly, Mozilla uses Wasm to sandbox third party C libraries in the Firefox renderer [NDG+20, Fro20]. The Wasm compiler—in this case, the Bytecode Alliance Lucet compiler [Byt20a]—essentially serves to instrument the library with inline SFI checks that restrict the library control and data flows to its own sandbox. But if this Wasm-based SFI fails, everyday web users are vulnerable to

remote attackers, who could exploit such "sandbox escape" bugs to do anything from steal data to fully compromise machines.

In general, all systems that use Wasm for isolation implicitly trust the Wasm compiler with their users' safety—the compiler is solely responsible for enforcing isolation by inserting checks into the native code it generates. However, compilers are complex software artifacts and code optimization passes are especially notorious for introducing unintended consequences [YCER11, RCC+12, TLR20, YJO+17, LSS15, SLS16]. Wasm compilers perform optimizations *after* inserting memory safety checks,[1] so any bugs in optimization could allow a carefully generated Wasm program to bypass Wasm's safety checks. These bugs actually happen: a bug in the Lucet loop invariant code motion pass broke SFI-safety, allowing maliciously-crafted Wasm code to escape the sandbox to, say, read arbitrary memory [Han19a]. Though there's an active research community focused on building verified compilers (e.g., [Ler09, Chl10])—even a verified Wasm compiler [BLP20]!—it seems likely that defects in industrial compilers will persist for some time.

Given this practical reality, we propose an alternative method for ensuring the safety of Wasm code: *verifying* the isolation of native-compiled Wasm modules using a verifier that operates directly on the generated machine code. This approach has a number of benefits. First, by working with native binaries, the verifier engages directly with the "final form" of a module and does not need to consider any further transformations to the instruction stream. Second, the verifier is much simpler than a compiler or run-time system, which not only shrinks the trusted computing base, but also makes the verifier easier to prove correct using formal methods. Finally, the memory-safety verifier runs once per Wasm module, and thus imposes no run-time overhead.

Our work makes the following contributions:

- A framework that describes sufficient conditions for reasoning about the memory-safety properties of native-compiled Wasm modules at the x86-64 instruction level.

[1] Note that this choice is on purpose, as subsequent optimization passes may allow redundant or unnecessary checks to be removed.

- The design and implementation of VeriWasm, a static offline verifier that uses this framework to check the memory safety of x86-64 binaries produced by the Lucet Wasm compiler.
- A formal proof that our verifier is *sound*, i.e., it only labels programs safe if they are indeed safe. We mechanically verify our proof using the Coq theorem prover.
- An empirical assessment of VeriWasm against both benign and faulty native-compiled Wasm modules, including two Wasm-sandboxed libraries used in Firefox and 103 client binaries deployed on Fastly's production edge cloud.

Our code—both VeriWasm and our Coq model and proofs—and data are available under an open source licence [JTA+20].

3.1 A brief intro to Wasm and its ancestors

The promise of safe, portable binary code is one that has driven well over 25 years of research. We briefly summarize this history to place Wasm in context, give an overview of Wasm's design, and finally explain how VeriWasm can help.

3.1.1 Ancestral systems

The idea of creating portable execution formats, independent of high-level language or machine architecture, is at least sixty years old—going back to Conway's UNCOL effort in the late 1950s [Con58]. But, in spite of a range of interesting systems (e.g., notably UCSD's p-System [Cam83]), the idea did not take off until the 1990s, which saw the introduction of the Open Software Foundation's Architecture Neutral Distribution Format [Mac92], Sun's Java Virtual Machine [LYBB14], and Microsoft's Common Language Runtime [MG01].

Of these, Java was one of the first systems to address the need for safety. Java included both a load-time verifier and run-time checks designed to ensure that maliciously constructed

Java bytecode could not bypass the language's type-and memory-safety guarantees. This proved challenging in practice, in part because Java's complex type system in turn demanded a complex verifier—and, unfortunately, verifier bugs led to a broad range of "sandbox escapes" [BD18].

In 1994, Wahbe et al. introduced the notion of *Software-based Fault Isolation* (SFI) [WLAG93], a family of binary instrumentation techniques for ensuring coarse-grained memory safety. While Wahbe and colleagues integrated their original SFI concept into the Omniware mobile code format [ATLLW96], most subsequent efforts focused on developing SFI in the context of existing instruction set architectures (e.g., x86, x86-64) [SS98, ES00, MM06, YSD+09, SMB+10]. Several of these, including the original SFI scheme, were designed with load-time verification in mind, i.e., they properly instrumented binaries to not only satisfy all memory-safety invariants, but make it easy to verify these invariants. Google's NativeClient (NaCl) is perhaps the most popular of these systems—in part due to its integration in the Chrome browser—and its PNaCl variant included a portable intermediate binary format [DMCS10], like the original Omniware.

WebAssembly, first announced by the W3C in 2015, is an effort to mainstream much of this past work and produce a standard high-performance machine-independent bytecode that is also safe [HRS+17].

3.1.2 Wasm: Fast, modern SFI

Wasm was explicitly designed to be easy to sandbox—after all, browsers need to parse, validate, and compile Wasm within a page load. Indeed, many of Wasm's design choices—from its static type system to its structured control flow and memory hierarchy—are in service of sandboxing: they make it easy for a compiler to generate native code that is SFI-safe, i.e., code whose data- and control-flow are isolated to the sandbox. These design choices are also the reason organizations like Fastly and Mozilla are starting to use Wasm to sandbox potentially untrusted code: by compiling to Wasm, they get SFI for free.

Control-flow safety. Wasm compilers must ensure that the control flow of the generated native

code is safely restricted to the sandbox. Two Wasm design features simplify this task. First, Wasm exposes structured control instructions (e.g., `if` blocks and `loops`) that preserve Wasm's type safety; Wasm does not expose unstructured control flow instructions like `goto` that would make this task harder. Second, the bytecode only allows indirect control transfers via static look-up tables: `call_indirect`, for example, is used to call functions registered in the module indirect-function table; `br_table` is used to jump to local branch-table blocks. Both of these instructions perform dynamic checks to ensure control is transferred to a valid function or block of the correct type, respectively. They also make it easy to ensure the module's indirect control flow is safe when compiled to native code. For example, when compiling `call_indirect idx` to x86-64, the Lucet compiler simply ensures that the index (`idx`) corresponds to a registered function, checks the type of the function at this index, and then calls the function.

Memory isolation. Wasm compilers must also ensure memory isolation. Again, deliberate language design choices simplify this task: Wasm exposes three distinct isolated memory regions—the stack, global variables, and a linear memory region—which must be accessed with different type-safe instructions. This makes it easy for a compiler to ensure that memory accesses are safe when compiling to native code. Instructions that access stack variables (`local.get/set idx`) or global variables (`global.get/set idx`) can simply be compiled to native code that access memory at constant offsets (`idx`) from the stack pointer and global variable memory base, respectively. Similarly, linear memory accesses (`load/store offset`) can be compiled into native loads and stores that access memory at `offset` off the linear memory base, after appropriately ensuring the `offset` is within the linear memory bound.

3.1.3 Trust but verify

While Wasm makes it easy for compilers to enforce SFI, we still have to *trust* compilers to do so correctly. Correct compilation is easier said than done. Modern optimizing compilers are complex, and a single bug in an optimization pass could result in a sandbox escape. On a Wasm

edge-cloud, this could, for example, allow an attacker to steal or corrupt data sensitive to other clients or the cloud provider (e.g., SSL keys). In the browser, where Wasm is used to sandbox libraries, it could allow an attacker to compromise the renderer to, again, steal or corrupt sensitive data.

We need a way to *verify* that Wasm compilers preserve SFI even across optimization passes. One way to do this is to prove that the compiler is correct (e.g., like the Certified CompCert C compiler [Ler09]). Alas, verifying an industrial optimizing compilers is notoriously hard (e.g., it took CompCert 100,000 lines of Coq and 6 person-years for the proof alone [LBK+16]). Luckily, we don't need to prove a compiler correct to ensure that the programs it produces are safe. Instead, like NaCl, we can verify that the compiler inserts the necessary checks to enforce SFI safety in each binary it produces. In this paper, we describe such a verifier for the Lucet compiler.

3.2 VeriWasm overview

VeriWasm is a static SFI verifier for native-compiled Wasm. VeriWasm takes as input a possibly buggy or malicious native-compiled Wasm module, and uses a sound static analysis to determine if the module is safe. In this section, we give a brief overview of VeriWasm's design and the four local safety properties it verifies. In Section 3.3 we describe the static analysis passes that verify these properties.

VeriWasm design. VeriWasm verifies binaries produced by the Lucet WebAssembly compiler. Fig. 3.1 gives an overview of VeriWasm's different stages. The tool first recursively disassembles the native-compiled Wasm binary and produces a control-flow graph for every function exposed in the symbol table. As a part of this process, VeriWasm also resolves all indirect jumps in the control-flow graph (see Section 3.3.4), and ensures that all direct and indirect calls target functions present in the symbol table. Then, VeriWasm checks the disassembled code against a list of safe native instructions—the instructions the Lucet compiler emits—and rejects binaries

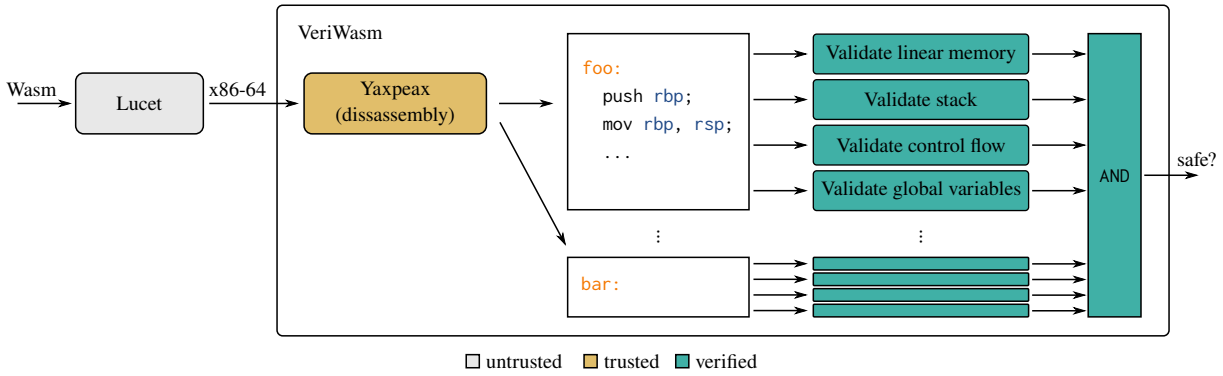


Figure 3.1: VeriWasm takes as input a malicious or buggy natively-compiled Wasm module and uses a trusted disassembler (Yaxpeax [yax20]) to create a CFG for each function in the module. VeriWasm then runs its verified analysis passes on each function's CFG to determine if the binary preserves SFI-safety.

with potentially unsafe instructions (like `int` and `syscall`). Finally, VeriWasm analyzes each function to verify local safety properties; if all functions are safe, VeriWasm then declares the module safe.

VeriWasm's local safety properties. To verify if a compiled function is safe—that its control- and data- flow is isolated to the Wasm module—VeriWasm verifies *four local safety properties*:

- *Linear memory isolation*: All linear memory accesses must fall within the linear memory bounds (or surrounding guard pages). This ensures that linear memory operations cannot span beyond the sandbox boundary.

- *Stack isolation and integrity*: All stack accesses must fall within the stack region (or surrounding guard pages) and all stack writes must be restricted to local variables in the current stack frame. This ensures that stack accesses cannot extend past the sandbox and prevents a function from writing past the local variables stored on the stack (e.g., return addresses, arguments, and other frames).

- *Global variable isolation*: All global variable accesses must fall within the global variable memory region. This ensures global variable accesses cannot extend past the sandbox.

Feature	Safety property	Description
Linear memory	Linear memory isolation	All linear memory reads and writes fall within the 4GB linear memory space (or surrounding guard pages).
Stack	Stack isolation	Stack reads fall within the stack region (or surrounding guard pages).
	Stack-frame integrity	Stack writes are to local variables in the current stack frame.
Global variables	Global variable isolation	Global variable accesses fall within the global variable memory region.
Control flow	Jump target validity	All indirect jumps target valid code blocks.
	Call target validity	All indirect calls target valid functions.
	Return target validity	Functions return to their respective call sites.

Figure 3.2: The safety properties VeriWasm verifies to prove SFI-safety. For clarity the stack and control flow safety properties are broken down into sub-properties.

- *Control flow safety*: All indirect jumps must target valid code blocks, all indirect calls must target the start of valid functions, and all returns must return to the calling function. This coarse-grained control flow integrity (CFI) ensures that only VeriWasm-verified code runs.

These safety properties are sufficient to prove that native-compiled Wasm binaries are safe. But these safety properties are also less restrictive than Wasm itself. For example, Wasm's CFI restricts indirect calls to target functions of the right type; our control flow safety property, on the other hand, only requires indirect calls to target valid functions (see §3.3.4 for further discussion). Wasm similarly restricts functions from reading beyond their stack frame, while our stack isolation property allows functions that read the whole stack. This difference is important: it simplifies the analysis (e.g., by not requiring it to perform type inference to enforce Wasm's finer-grain, type-based CFI). We describe our analysis—which exploits Lucet's implementation choices for additional simplicity—next.

3.3 VeriWasm's analysis

VeriWasm uses abstract interpretation to verify the isolation of Wasm modules compiled with Lucet. Abstract interpretation is a static analysis technique that infers information about a program by overapproximating its behavior. It executes the code similar to a standard interpreter, but describes a program's variables as abstract values representing a set of concrete values relevant

to a safety property of the variable. For example, VeriWasm's linear memory safety analysis tracks whether variables in registers and on the stack are less than 2^{32} which represent valid offsets into the linear memory.

Each of VeriWasm's safety analyses have different abstract semantics; however, they all have some characteristics in common. The first is that they only track variables in registers and on the stack, but not in the linear memory. VeriWasm can validate modules without tracking variables across the linear memory because Lucet's optimizations also do not track variables across the linear memory. The second characteristic is that all of VeriWasm's analysis passes analyze each function independently. When VeriWasm encounters a call instruction, it skips over the call, but acts as if the callee modified every register—they must be checked again before the code uses them in an SFI-sensitive operation. This ensures that even if the callee modifies callee-saved registers, the caller function is still safe. The caller's stack frame and stack pointer are preserved as a consequence of stack frame integrity (§3.3.2). The third characteristic is that all of VeriWasm's analyses are sound: all functions VeriWasm labels as safe are indeed safe.

Unlike previous SFI systems (e.g. [WLAG93, YSD+09, MM06]), Wasm was not designed with verification in mind—and this necessarily makes our analysis more complex. The NaCl compiler, for example, compiles code to *bundles* [YSD+09, MM06][2], and ensures that all safety checks are local to a bundle and cannot be to moved (outside the bundle) or optimized away. This allows the verifier to perform a single pass analysis, at the bundle-level, to ensure both memory and control flow safety [YSD+09, MTT+12]. In contrast, Wasm compilers perform optimizations *after* inserting SFI safety checks—and thus checks can be moved (e.g., checks can be lifted outside of a loop as part of a loop invariant code motion, as mentioned in Section 3) or, when redundant, removed. This is crucial for performance, but it unfortunately means that the verifier must account for such optimizations when checking SFI safety. In the rest of this section, we describe VeriWasm's analysis passes, ending with a description of how VeriWasm validates

[2]A bundle is a 32-byte aligned group of instructions.

safety in the presence of compiler optimizations.

3.3.1 Linear memory isolation

VeriWasm verifies that the module's linear memory is isolated—that all linear memory accesses fall within the module's linear memory region. Lucet gives each module a contiguous 8GB region above a linear memory base; the region is composed of 4GB of usable memory followed by a 4GB guard page. Lucet compiles all linear memory accesses to x86-64 instructions with an effective address of the form `LinearMemBase + Offset1 + Offset2`, where `Offset1` and `Offset2` are 64-bit values that should be less than 2^{32}. Addresses of this form should be at most `LinearMemBase` + 8GB, which will either be in the linear memory or the following guard page. VeriWasm ensures that all linear memory accesses are constrained to the linear memory region by verifying that: (1) `LinearMemBase` points to the linear memory base and (2) that `Offset1` and `Offset2` are less than 2^{32}.

Tracking linear memory safety. VeriWasm verifies linear memory isolation by tracking which *variables*—registers or stack slots—have a concrete value less than 2^{32} (`Bounded`), and which variables point to the start of the linear memory (`LinearMemBase`). All variables start with an abstract value of `Unknown`, except the linear memory base register (`rdi`), which starts with the value `LinearMemBase`. Other variables only become `LinearMemBase` when they're assigned from a variable with abstract value `LinearMemBase`. Variables become `Bounded` when they are truncated or assigned a 32-bit value. For example, after `mov eax, eax`, register `rax` is now `Bounded`, and after `mov rbx, 0x1337`, `rbx` is now `Bounded`.

Verifying linear memory isolation. VeriWasm applies two checks to the results of this analysis to prove that linear memory is properly isolated. VeriWasm first checks that, for all linear memory accesses, one argument is `LinearMemBase`, and the other arguments are `Bounded`. VeriWasm also verifies that at each function call, `rdi` is `LinearMemBase`. This is required by Lucet's calling convention, and violating that convention could break linear memory isolation.

The following code shows how VeriWasm verifies a function, `foo`, that reads an element in linear memory and then calls a function `bar`:

```
1  foo:
2  ; ASSUME: rdi is LinearMemBase
3  ; TRACK: rax, rbx, ... are Unknown
4    ...
5    mov eax, eax;
6    ; TRACK: rax Bounded
7    mov rsi, [rdi + rax + 0x48];
8    ; ASSERT: rdi is LinearMemBase
9    ; ASSERT: rax and 0x48 are Bounded
10   ...
11   call bar;
12   ; ASSERT: rdi is LinearMemBase
13   ...
```

The read on line 7 is safe since `rdi` is `LinearMemBase` and `rax` and `0x48` are `Bounded` (since `rax` has been truncated to 32 bits on line 5). This safety check would fail if `rdi` had been altered to point to something other than `LinearMemBase`, or if `rax` had any possibility of being greater than or equal to 2^{32}. It would also fail if `rdi` were not `LinearMemBase` in the call to bar (line 11).

3.3.2 Stack isolation and stack-frame integrity

VeriWasm verifies that native-compiled code cannot break isolation by misusing the stack. We do this by verifying:

- *Stack isolation*: stack reads and writes fall within the module's stack region (or surrounding guard pages, described below).

- *Stack-frame integrity*: stack writes are additionally bounded by the base of the current stack frame, i.e., stack writes cannot clobber return addresses, spilled arguments, or other function stack frames.

Before we describe how VeriWasm verifies these properties, we describe how Lucet compiles stack operations, and describe the layout of the stack region.

Native-Wasm stack instructions. Lucet compiles Wasm stack accesses into three kinds of native operations:

`rsp = rsp ± c`	stack adjustments
`x = mem[rsp ± c]`	stack loads
`mem[rsp ± c] = x`	stack stores

In this syntax, `c` is a constant: Lucet adjusts the stack by a constant amount, stores variables at a constant offset from the stack pointer, and loads variables from a constant offset from the stack pointer. VeriWasm takes advantage of the constant offsets to statically infer what part of the stack region any particular read or write falls within.

Native-Wasm stack layout. Lucet allocates a contiguous region—typically 128K—for the module stack region. Both ends of the stack region, as shown in Fig. 3.3, are guarded. Lucet uses a 4K guard at the end the stack region, and an 8K guard below the start of the stack region. The 4K guard region prevents most functions—all functions with fewer than 4K local variables—from growing the program stack past the stack region: at worst, accessing a local variable will land in the 4K region and trap. Some functions use more than 4K of local variables, though. For these, Lucet adds a dynamic check `probestack(k)` in the function prologue before growing the stack by `k`. The probestack ensures that growing the stack is safe, i.e., within the 128K stack region, and traps otherwise. The 8K guard at the stack region base just fits the maximum number of spilled arguments (8,000 bytes) and any control data; we require this guard region to simplify our

binary analysis.[3]

Our stack safety verification is based on the key observation that a function will (1) read *at most* 8K above (towards bottom of stack region) the stack frame pointer—the function's spilled arguments; and (2) read and write *at most* 4K (or `k` if the function prologue has a `probestack(k)`) below (towards the top of the stack region) the stack frame pointer—the function's local variables. Since stack operations are in terms of the stack pointer, though, our analysis must accordingly track the difference between the stack pointer and frame pointer to verify if any particular read or write is safe.

Tracking the stack growth. VeriWasm tracks the stack growth at each point in a function. The growth, `StackGrowth`, is the difference between the stack frame pointer (the stack pointer before the function start executing) and the stack pointer after each instruction, i.e., `StackGrowth` = $\texttt{rsp}_{\texttt{current}}$ - $\texttt{rsp}_{\texttt{start}}$. At the start of each function, `StackGrowth` is zero. Then, whenever an instruction modifies the stack, VeriWasm accordingly adjusts `StackGrowth`. For example, pushing a value to the stack decreases `StackGrowth` by eight, popping does the converse. At each merge point (e.g., the block after an `if-else`), VeriWasm checks that `StackGrowth` is the same for all incoming paths; if not, VeriWasm sets `StackGrowth` to `Unknown`.

Verifying stack isolation. To verify stack isolation, VeriWasm checks all stack accesses of the form `mem[rsp + c]` access memory within the stack region. Specifically, for functions without a probestack, this amounts to checking:

$$-4\text{K} < \texttt{StackGrowth} + \texttt{c} < 8\text{K}.$$

For functions with probestack, we instead check:

$$\min(-4\text{K}, \texttt{k}) < \texttt{StackGrowth} + \texttt{c} < 8\text{K},$$

[3]We're working with Bytecode Alliance to integrate this patch into Lucet.

where k is the argument to the probestack(k) call.

Verifying stack frame integrity. To verify stack frame integrity, VeriWasm checks that for all stack writes of the form mem[rsp + c] = x fall within the 4K read-write region, i.e.,

$$-4\text{K} < \texttt{StackGrowth} + \texttt{c} < 0.$$

This ensures that the function cannot write past the stack frame pointer to, for example, clobber return addresses.

The following code shows a function bar which writes the value of rdi to a stack slot, then reads a spilled argument into the rax register.

```
1  bar:
2    ; TRACK: StackGrowth 0
3    push rbp;
4    ; TRACK: StackGrowth -8
5    mov rbp, rsp;
6    sub rsp, 0x10;
7    ; TRACK: StackGrowth -24
8    mov [rsp + 0x8], rdi;
9    ; ASSERT: -4K < StackGrowth + 0x8 < 0
10   mov rax, [rsp + 0x20];
11   ; ASSERT: -4K < StackGrowth + 0x20 < 8K
12   ...
```

Lines 3 and 6 decrease StackGrowth: the push on line 3 decreases StackGrowth by eight and the allocation of stack space on line 6 decreases StackGrowth by sixteen. On line 8, the stack write is checked to verify the frame's integrity. This check passes because StackGrowth must be -24 at this point, so the address mov rsp + 0x8 must be a local stack variable. On line 10, the stack read is checked to ensures the read is within the stack region. This check also passes because it reads from the 8K before the stack frame, accessing a spilled argument.

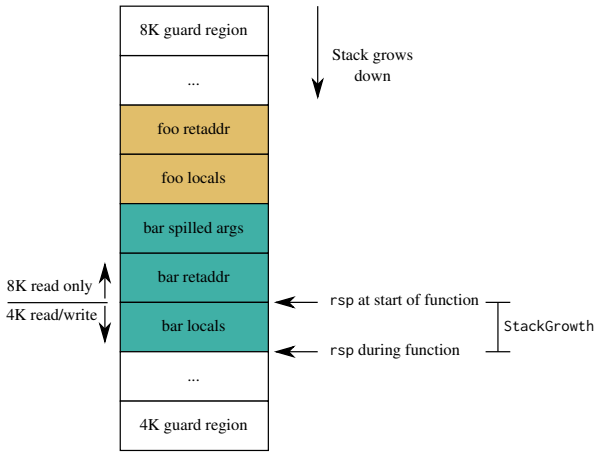


Figure 3.3: Stack layout of function `bar` called by function `foo`. VeriWasm ensures that `bar` can safely read and write to local variables (at most 4K) in its own stack frame but not beyond the stack frame pointer—to clobber its return address or `foo`'s stack frame. Moreover, VeriWasm ensures that `bar` can also read its spilled arguments (at most 8K).

3.3.3 Global variable isolation

VeriWasm validates that all global variable accesses are contained within the global memory region; the region consists of one or more 4K pages, and Lucet statically declares the region's size within the binary. Lucet compiles all global variable accesses to constant offsets from the base of the global memory region. This makes validation simple: VeriWasm tracks the base of the global memory region, and validates that it is only accessed with a constant offset that is within the declared size of the region.

Tracking global variables. VeriWasm tracks which variables point to the start of the global memory region (`GlobalsBase`). Initially, Lucet stores `GlobalsBase` just below the base of the linear memory, at `LinearMemBase - 0x10`. VeriWasm figures out which variables point to `GlobalBase` similarly to how linear memory safety analysis tracks `LinearMemBase` (§3.3.1).

Verifying global variable safety. VeriWasm checks that all global variables accesses are contained within the global variable memory region. At each global variable access, it checks that the offset is less than `GlobalSize`, the size of the globals region. The following code shows how

VeriWasm validates a function that increments a global variable:

```
1 ; ASSUME GlobalSize 4096
2 ; TRACK: rax, rbx, ... are Unknown
3 ...
4 mov rax, [rdi - 0x10];
5 ; ASSERT: rdi is LinearMemBase
6 ; TRACK: rax GlobalsBase
7 mov rbx, [rax + 0x18];
8 ; ASSERT: rax is GlobalsBase
9 ; ASSERT: 0x18 < GlobalSize
10 add rbx, 1;
11 mov [rax + 0x18], rbx;
12 ; ASSERT: rax is GlobalsBase
13 ; ASSERT: 0x18 < GlobalSize
14 ...
```

First, line 4 loads `GlobalsBase`, and VeriWasm validates that it's loaded from the correct location; this would fail if `GlobalsBase` were loaded from anywhere besides `[LinearMemBase - 0x10]`, since anything else below `LinearMemBase` is illegal. Next, VeriWasm checks that the offset of the global variable is less than `GlobalSize`: this ensures that the access stays in the allocated region (line 7). VeriWasm similarly checks the write offset on line 11. All accesses are to offsets less than the `GlobalSize` of 4096, so the checks pass.

3.3.4 Control flow safety

VeriWasm verifies the control flow safety of native-compiled Wasm binaries. Specifically, we verify (1) *jump target validity*—that indirect jumps target valid code blocks; (2) *call target validity*—that indirect calls target the start of valid functions; and (3) *return target validity*—that function returns actually return to their call sites.

Jump target validity

To ensure that the control flow graph for each function is complete, VeriWasm resolves all indirect jumps within the CFG. While the problem of statically deciding the targets of computed jumps is notoriously difficult [WZH+11] (if possible at all), Wasm's language level restrictions on indirect jumps make it tractable for native-compiled Wasm modules. In particular, Wasm does not support arbitrary computed jumps. The only computed jump Wasm supports is the branch or jump table instruction `br_table`. Lucet compiles these jump table instructions as follows:

```
1  ; TRACK: rax, rbx, ... are Unknown
2  ...
3  cmp rax, 0x7; compare jump index to table size
4  jae default_case;
5  ; TRACK: rax is Checked(0x7)
6  mov rdx, 0x4000; load of a jump table
7  ; TRACK: rdx is JumpTableBase(0x4000)
8  mov rbx, [rdx + rax * 4];
9  ; TRACK: rbx is JumpOffset
10 add rdx, rbx; jump to the target
11 ; TRACK: rdx is JumpTarget
12 jmp rdx;
```

Here, the native-compiled Wasm first checks that the jump index (`rax`) is less than the total number of table entries (lines 3 and 4). If it is, the code performs a lookup into the jump table and jumps to the appropriate case (lines 6-12).

VeriWasm exploits the structure of the jump table check to determine all possible jump targets for each jump table.[4] Then, VeriWasm verifies that every jump target is safe by running the verifier on the CFG at the target site. (This is an overapproximation of possible jump targets, since some jump table targets may remain unused.) VeriWasm does this using a fix-point algorithm:

[4] If VeriWasm encounters a computed jump that does not follow this pattern, the code is not a valid Lucet compilation output and the safety check fails.

while a single pass of this analysis may resolve all indirect jumps with respect to a particular CFG, resolving these jumps may also reveal additional indirect jumps which require further disassembly. Because of this interdependence between the CFG and the indirect jump analysis, VeriWasm alternates between disassembling passes and jump-resolution passes. It stops when it reaches a fixed point in the CFG generation process, i.e., when the analysis no longer resolves new jumps.

Tracking jump targets. To verify that indirect jump targets are safe, VeriWasm must identify all potential jump targets. To this end, VeriWasm tracks which variables within a function represent intermediate steps of a valid indirect jump target lookup. On line 6 in the above code snippet, for example, `rdx` takes on the abstract value `JumpTableBase(0x4000)` since it was assigned the constant `0x4000`. In this same block, `rax` has the abstract value `Checked(0x7)`; it was checked to be less than `0x7` on lines 3 and 4.

Verifying jump targets. VeriWasm uses the abstract values from the analysis phase to identify all possible targets for the indirect jump on line 12. In particular, VeriWasm uses the jump table bounds check—the abstract `Checked(·)` value—to identify the number of jump target entries from the jump table base—the abstract `JumpTableBase(·)` value—it needs to verify. For example, in the above code, VeriWasm identifies that the possible targets are the first seven entries at `0x4000`, and then disassembles and verifies the code at those locations.

Call target validity

To verify that a function is safe, VeriWasm validates that every function that it calls is also safe. Indirect calls make this hard: VeriWasm cannot necessarily determine the target of the call. Instead, VeriWasm overapproximates the set of possible targets of the indirect call, and validates that every possible target is safe. VeriWasm can do this because Wasm treats indirect calls as a lookup into a statically known function pointer table.

Lucet performs a dynamic safety check on all indirect calls to ensure each call target

is present in the indirect function table. By verifying that all indirect calls enforce this check, and that all function pointers in the module indirect-function table point to the start of verified functions, we inductively verify that indirect calls are safe.

While Wasm requires all indirect calls to be well-typed—and Lucet enforces this with a runtime check—VeriWasm does not attempt to verify this. Doing so would be hard. Computing the number of arguments to each function alone is difficult—it requires us to infer the number of arguments passed at each call site. This is harder once we consider indirect calls to the function—to infer the number of arguments passed to a function we would have to analyze all indirect call sites that could potentially call the function. Happily, calling a function with an incorrect type for Lucet compiled Wasm does not affect SFI-safety. First, Lucet validates arguments before they are used in SFI-sensitive operations (e.g., loads and stores), regardless of their type. Second, the 8K guard region below the stack region (§3.3.2) ensures that reading stack-spilled arguments can never result in stack accesses outside the stack region.

Below we give an example of a function `baz` correctly loading a function pointer from the indirect function table and calling that function.

```
1  ; TRACK: rax, rbx, ... are Unknown
2  baz:
3    push rbp;
4    mov rpb, rsp;
5    mov r9, 0x2000;
6    ; TRACK: r9 MetadataBase
7    mov r9, [r9 + 0x8];
8    ; ASSERT r9 is MetadataBase
9    ; TRACK: r9 TableSize
10   cmp r9, rax;
11   jb case2;
12
13 case1:
```

```
14   ud2;
15
16 case2:
17   mov r9, 0x3000; base of indirect func table
18   ; TRACK: rax Bounded
19   ; TRACK: r9 FuncTableBase
20   shl rax, 0x4;
21   ; TRACK: rax PtrOffset
22   mov rax, [r9 + rax + 0x8];
23   ; ASSERT: r9 is FunctionTable
24   ; ASSERT: rax is PtrOffset
25   ; TRACK: rax FnPtr
26   call rax;
27   ; ASSERT rax is FnPtr
28   ret;
```

Lines 5-11 load the indirect function table size from the module metadata and check that `rax` is less than the total number of entries in the table, and thus that `rax` holds a valid index. This check ensures that loading the entry corresponding to the function index is in-bounds and cannot be used to read memory beyond the module boundary. In particular, if the index in `rax` is out-of-bounds (greater than the number of entries in the indirect function table) the code proceeds to `case1` which triggers an illegal instruction exception. If the index is in-bounds, lines 17-26 use the checked `rax` to lookup a function pointer in the indirect function table, and then call it. VeriWasm's abstract analysis tracks the steps of this process to ensure that all steps have been performed correctly.

Tracking call targets. The call safety analysis tracks which variables within a function are used as intermediate values in the function pointer checking process. The intermediate values of the call check are `MetadataBase` (`r9` on line 5), `TableSize` (`r9` on line 7), `Bounded` (`rax` on line 17), `PtrOffset` (`rax` after line 20) and FnPtr (`rax` after line 22). If VeriWasm cannot verify that a

variable is any of these intermediate values, then the value is `Unknown` and should not be used to create a valid function pointer. If any of the operands to the two memory operations (table size lookup and function table lookup) are `Unknown`, VeriWasm triggers a safety violation, since this access could potentially break isolation. The analysis is designed in such a way that the abstract value representing a step of the lookup process can only be produced by the correct operation applied to the results of two correctly computed previous steps. For example, the abstract value `FnPtr` representing a valid function pointer can only be produced by dereferencing the base of the indirect table added to a valid entry offset in that table, which in turn must have already been bounds checked to be within the table.

Checking call check integrity. Checking call safety is as simple as checking that the register acting as the target for each indirect call has the abstract value `FnPtr`, representing a properly computed call target. This check on line 26 succeeds because `rax` has been computed through the correct sequence of steps. VeriWasm would have triggered a safety violation if the value of `rax` was not checked, or if the lookup was not performed correctly (e.g., if the pointer was modified after the code loaded it from the table).

Checking function table integrity. While validating our call check integrity ensures that the target of each indirect call check is properly loaded from the indirect call table, VeriWasm also validates that every function pointer present in the indirect call table points to the start of a valid function. VeriWasm checks that the indirect function table is located in a read-only section, and performs a one-time check when the module is loaded that all pointers present in the table target the start of validated functions.

Return target validity

VeriWasm validates that all functions return to their call site. To do this, VeriWasm validates that at each return site in the function the stack pointer points to the return address pushed by the calling function. We do this by verifying that the stack growth—the difference

between the `rsp` and `rbp`—of Section 3.3 is zero at each return site.

3.3.5 Making VeriWasm robust to compiler optimizations

Lucet performs optimizations after it inserts safety checks, which means that optimizations can modify these checks. Lucet can, for example, split checks across basic blocks, modify checks, or reorder checks. If VeriWasm is not precise enough, it can falsely label these modifications unsafe; below, we describe how VeriWasm handles optimizations without triggering false positives.

VeriWasm allows safety check modifications as long as all necessary checks have been performed by the time the checked variable is used in an SFI-sensitive operation (e.g., a load or a jump). One example of a safety check modification is how Lucet reorders the steps of indirect call checks. Normally, it generates indirect call checks that load the function pointer in three steps:

1. Check the function index lies within the call table.
2. Create the `PtrOffset` into the indirection function table for this index. This `PtrOffset` points to a valid entry in the call table, since the function index has already been checked.
3. Dereference the `PtrOffset` to retrieve the function pointer.

Lucet's optimizations often reorder steps one and two of the check:

1. Create a `PtrOffset` from a potentially unsafe index. This `PtrOffset` points to a valid entry in the indirect function table *only if the base index is shown to be safe before* `PtrOffset` *is dereferenced*.
2. Check the function index lies within the call table.
3. Dereference the `PtrOffset` to retrieve the function pointer.

This reordered check is safe because the function index has been checked (and therefore the `PtrOffset` must be valid) before the `PtrOffset` generated from the index is dereferenced. Crucially, it's not possible to validate this check's safety using the strict formulation in Section 3.3.4, since the strict safety check requires pointer offsets to be generated from a known-valid base index. Therefore, the naive version of VeriWasm triggers false positives.

To make VeriWasm's analyses robust to reordering, we use *dependent abstract variables* (DAV). A DAV is a variable whose safety depends on the safety of another variable. VeriWasm uses DAVs in both call safety analysis and indirect jump analysis. In the example above, VeriWasm uses a `DependentPtrOffset` instead of a `PtrOffset`, since the base index has not yet been checked. If the `DependentPtrOffset` is used before its base index has been checked, validation will fail. Concretely, consider the following (correct) function that performs an indirect call with a reordered safety check:

```
1  ; TRACK: rax, rbx, ... are Unknown
2  baz:
3    push rbp;
4    mov rpb, rsp;
5    mov r9, 0x2000;
6    ; TRACK: r9 MetadataBase
7    mov r9, [r9 + 0x8];
8    ; ASSERT: r9 is MetadataBase
9    ; TRACK: r9 TableSize
10   shl rax, rdx, 0x4;
11   ; TRACK: rax DependentPtrOffset(rdx,6)
12   cmp r9, rdx;
13   jb case2;
14
15 case1:
16   ud2;
17
```

```
18 case2:
19   mov r9, 0x3000;
20   ; TRACK: rdx Bounded
21   ; TRACK: rax PtrOffset
22   ; TRACK: r9 FuncTableBase
23   mov rax, [r9 + rax + 0x8];
24   ; ASSERT: r9 is FunctionTable
25   ; ASSERT: rax is PtrOffset
26   ; TRACK: rax FnPtr
27   call rax;
28   ; ASSERT: rax is FnPtr
29   ret;
```

On line 10 `rax` is a `DependentPtrOffset` that is dependent upon the base index (`rdx`). If `rax` were immediately used to lookup the indirect function pointer, validation would fail. However, if `rdx` were checked first (as on lines 12 and 13), validation would succeed, since `rax` must necessarily be valid by the time it is used.

VeriWasm uses reaching definitions [ASU86] to resolve DAV constraints. Reaching definition analysis records the set of locations that could have written to registers and stack locations at every point in the program as well as the expression assigned to these locations. If two registers or stack slots have identical reaching definitions, they must necessarily have the same value i.e. they are aliased. This means that if a register or stack slot is declared safe, all other variables with the same set of reaching definitions are safe as well. Whenever a variable is checked, VeriWasm also resolves all dependent abstract values with the same set of reaching definitions as the variable being checked (and any variables derived from them).

3.4 Evaluation

We evaluate VeriWasm by asking three questions:

- Does VeriWasm find SFI breaking bugs—can it discover compilation bugs in Wasm binaries that allow accesses outside sandbox memory?
- Does VeriWasm have a low false positive rate—does it avoid incorrectly flagging Wasm binaries that are actually safe?
- Is VeriWasm fast enough—can it validate Wasm modules as part of the compilation process for browsers and edge cloud providers?

To check if VeriWasm is able to find SFI breaking bugs, we evaluate it on a suite of known bugs from previous SFI systems and Wasm compilers (§3.4.1); it's able to identify every bug in this suite. We also set up a fuzz harness for Lucet; while this did not reveal any bugs on the current version of Lucet, it helped us eliminate false positives in the VeriWasm tool. To evaluate VeriWasm's false positive rate and performance, we validate four sets of benchmarks—SPEC2006, the Lucet compiler's Shootout microbenchmark suite, two Wasm sandboxed libraries shipped by the Firefox browser, and 103 Wasm modules from the edge-cloud provider Fastly for a total of 119 Wasm modules. VeriWasm reports no false positives and validates most of these applications in less than twenty seconds; this latency, while not sufficient for just-in-time applications, allows Firefox and Fastly to run VeriWasm on nightly re-builds of Wasm binaries.

Table 3.1: Average function verification times, max function verification times, and total module verification times of SPEC2006 applications.

	astar	**bzip2**	**gobmk**	**h264ref**	**lbm**	**libquantum**	**mcf**	**milc**	**namd**	**povray**	**sjeng**	**soplex**	**sphinx**
Average Function Validation Time (s)	0.02	0.05	0.01	0.02	0.08	0.05	0.04	0.02	0.04	0.01	0.05	0.0	0.02
Max Function Validation Time (s)	6.23	5.82	5.63	5.78	10.59	10.55	6.08	6.42	5.73	5.59	9.59	10.14	9.91
# Functions in Module	334	170	2638	695	142	233	153	394	363	1901	279	3895	490
Total Validation Time (s)	7.27	7.74	13.6	16.57	11.83	10.78	6.72	7.17	15.66	19.31	13.83	19.25	11.01

Experimental setup. We run all experiments (with the exception of verifying Fastly's binaries) on a 2.1GHz Intel Xeon Platinum 8160 machine with 96 cores and 1 TB of RAM running Arch Linux 5.8.14. We evaluate the Fastly modules on a 5.5GHz Intel quad core i7-8559U machine

Table 3.2: Average function verification times, max function verification times, and total module verification times of Lucet's microbenchmarks.

	shootout
Average Function Validation Time (s)	0.63
Max Function Validation Time (s)	71.45
# Functions in Module	124
Total Validation Time (s)	78.04

Table 3.3: Average function verification times, max function verification times, and total module verification times of Firefox libraries currently deployed as native-compiled Wasm.

	libgraphite	**libogg**
Average Function Validation Time (s)	0.01	0.0
Max Function Validation Time (s)	7.61	0.03
# Functions in Module	674	270
Total Validation Time (s)	7.89	0.12

with 8GB of RAM. All experiments run on a single core, and no experiment uses more than 6GB of RAM. We compile the SPEC2006 and Shootout benchmarks using the Clang compiler (version 10.0.0) to go from C/C++ to Wasm, then compile the results to x86-64 using the Lucet compiler (version 0.7). The Firefox Wasm sandboxed libraries come from the Firefox nightly build (version 78.0a1 on May 4th, 2020). Fastly customers compiled their applications from the source language to Wasm, and Fastly compiles from Wasm to x86-64 on their own servers using the Lucet compiler (version 0.7).

Implementation. We implement VeriWasm in ≈3000 lines of Rust. VeriWasm uses the Yaxpeax disassembler for disassembly [yax20].

3.4.1 Does VeriWasm find SFI breaking bugs

Testing. We test if VeriWasm finds SFI-breaking errors by creating a suite of 11 bugs from other SFI toolchains like NaCl [YSD+09], miSFIt [SS98], and PittSFIeld [MM06], as well as old bugs

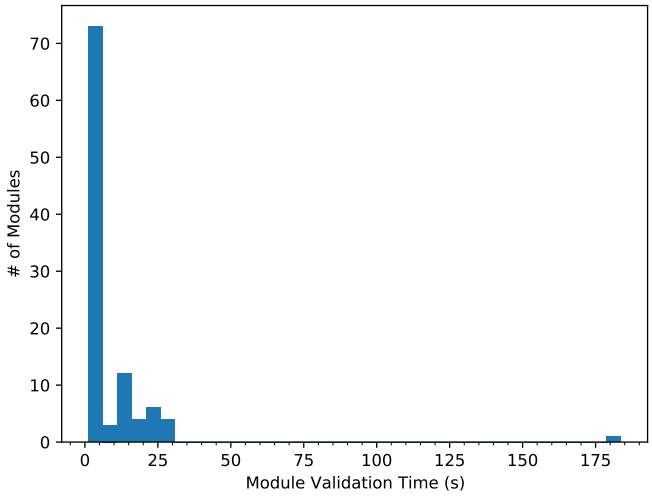


Figure 3.4: Total verification time for each of Fastly's client applications

from the Lucet compiler. These bugs fall into different categories: two violate call safety, three violate stack safety, two violate linear memory safety, one violates jump safety, and three use illegal instructions. The most interesting bugs include:

- A stack out-of-bounds write where the SFI scheme does not prevent the stack pointer from being moved outside the allocated stack range. This bug was found in the MiSFIt [SS98] SFI system by McCamant et al. [MM06]. Malicious code can exploit the bug by repeatedly allocating stack space until the stack pointer points to a different memory region (e.g., a different module's address space). VeriWasm catches this bug by verifying stack isolation and stack frame integrity.

- An unchecked memory access during indirect jump target lookup. A Lucet optimization re-ordered instructions so that bounds checks for indirect jump indices occurred after jump table lookups. VeriWasm catches this bug as part of its control flow safety checking, which ensures that jump table lookups only occur after the indirect jump index has been bounds checked.

- A bug where the memory safety checks and the control flow safety checks are accidentally mixed up, allowing for control flow through a register which has only been checked to be safe for memory access. Unlike the previous bugs in SFI compilers, this bug was discovered in the *verifier* for the PittSFIeld SFI scheme by Kroll et. al [KD09]. Specifically, the verifier marked certain binaries as safe, when they were actually unsafe. VeriWasm successfully classifies binaries with such unsafe pattern as unsafe.

We run VeriWasm on code that uncovers a given vulnerability by by either (1) compiling proof-of-concept code from an original bug report or (2) handwriting assembly with the vulnerability (when the proof-of-concept is not available). In some cases, we have to translate the bug to Wasm; for example, one bug unsafely manipulates a function pointer that has already been checked by NaCl. Since Lucet uses a different kind of function pointer check based on an indirect function table, we modify this bug use Lucet's function pointer check.

Fuzzing. We also integrated VeriWasm into a fuzz testing pipeline. Specifically, we (1) use Csmith [YCER11] to randomly generate valid C/C++ programs, then compile them to Wasm with Clang and (2) directly generate random Wasm modules using Binaryen [bin15]. We then compile these Wasm modules with Lucet and run VeriWasm on the resulting binaries. We run the fuzzing infrastructure on over 2 million Csmith-generated programs and 20 million Binaryen-generated programs, but do not find bugs in current version of Lucet.

The fuzzing infrastructure did, however, reveal early bugs in VeriWasm—VeriWasm declared some safe programs unsafe because Lucet optimizations reordered and manipulated some dynamic safety checks in a safe but unexpected way. We used these results to improve VeriWasm's precision (see §3.3.5).

3.4.2 Can VeriWasm validate correct programs?

VeriWasm cannot classify an unsafe program as safe; still, VeriWasm *can* flag safe code as unsafe. This may happen, say, if Lucet runs an optimization that VeriWasm cannot precisely reason about (e.g., the range-based loop check hoisting optimization Zeng et al. describe [ZTM11]). To check if VeriWasm's analysis is precise enough to avoid false positives on real code, we test it on binaries from four sources.

VeriWasm validates all four benchmark sets—a total of 119 Wasm binaries from SPEC2006 [Hen06], Lucet's Shootout microbenchmarks, Wasm sandboxed libraries in Firefox [NDG+20], and Wasm modules deployed in Fastly's edge cloud—with no false positives. SPEC2006 presents realistic workloads on applications like video compression, speech recognition, and ray tracing. We run VeriWasm on thirteen of nineteen SPEC benchmarks (all written in C and C++) and, like [JPBG19], we exclude six because they contain constructs that aren't representable in Wasm (e.g., `longjmp`, exceptions, and unsupported system calls). Shootout, which Lucet uses for performance testing, consists of a single Wasm module containing benchmarks like matrix multiplication and AES encryption; several benchmarks are particularly interesting for VeriWasm, since they contain some uncommon code patterns (e.g., very large switch tables). We also evaluate VeriWasm on two Wasm sandboxed libraries that ship with Firefox nightly [Fro20], and 103 Wasm modules currently deployed by Fastly [Byt20a]. Both of these benchmarks are of deployed real-world applications of Wasm sandboxing.

3.4.3 Is VeriWasm fast enough?

To evaluate if VeriWasm is fast enough to run on Firefox and Fastly's nightly builds of Wasm modules we use the four sources of benchmarks from the previous section. Verification of real modules takes on average 8.6 seconds and a median time of 1.7 seconds; while this is sufficiently fast for our intended use case of validation before a nightly build, it is too slow for

low latency applications like just-in-time compilers.

SPEC2006 performance. VeriWasm takes between 6.7 and 19.4 seconds to validate each SPEC module (Table 3.1). On average, the top 1% of functions account for 87% of execution time. We observe this for all benchmarks: a few large complex functions use most of the total verification time.

Shootout performance. VeriWasm requires 78 seconds to validate the Shootout module. A single function—a function that contains a huge switch table with 4096 cases (see Table 3.2)—dominates the verification time: it take takes 71 seconds to validate. When compiling this function, Lucet uses separate local variables for each switch case. This results in code with over 10,000 local variables that are maintained on the stack. So, VeriWasm takes a long time to validate this code—it must track information about all of these variables across a large, complex function.

Firefox performance. VeriWasm requires between 0.12 and 7.9 seconds to validate the Firefox libraries (see Table 3.3). This overhead is reasonable for our use case—verifying nightly builds—even with many libraries. Indeed, this is even cheap enough to run in the Firefox continuous integration tool.

Fastly client performance. VeriWasm takes between 1 and 183.7 seconds to verify each Fastly Wasm application, The median verification time is 1.85 seconds (see Figure 3.4). These applications have between 501 and 3123 functions, with a median of 621. Because the Fastly application code is confidential, we unfortunately can't diagnose and thus report why one of the applications takes three minutes to verify.

Performance breakdown. For each of our 119 modules, VeriWasm spends, on average, 75.2% of the verification time performing (CFG generation and) indirect jump analysis; 21.9% checking call safety; and, 2.9% verifying stack and heap safety. Indirect jump analysis is more expensive than other analyses for two reasons. First, this analysis records the reaching definitions of each value in the program in case they are needed; the other analyses only track values that have been computed by code that resembles safety checks. Second, because the analysis is inherently

coupled with CFG generation (§3.3.4), VeriWasm sometimes needs need to perform the jump analysis multiple times to resolve all the jumps in a particular function CFG.

When verifying a module, we find that VeriWasm spends, on average, 70.8% of its time verifying roughly 1% of the functions in the module. These functions are large and complex—e.g., the median number of basic blocks in one of these functions is 521, which is roughly 47× larger than the median number of blocks of function across all the modules we verify (11).

3.5 Limitations and Future Work

Unverified disassembly. VeriWasm relies on an unverified disassembler; bugs here can lead to bugs in VeriWasm's verification. However, this is not a fundamental limitation of the technique since verified disassembly has been demonstrated in prior work [MTT+12] and VeriWasm can be modified to adopt such approaches.

VeriWasm's speed. Although VeriWasm is sufficiently fast for its intended use case of checking nightly builds of Wasm code in browsers and CDN deployments, it is currently not fast enough to perform online verification in low latency applications like Wasm JIT compilers. This is because VeriWasm's control flow analyses seem to scale worse to large, complex functions than its other analyses (as discussed in VeriWasm 3.4.3). To address this issue and enable online verification in the future, we plan to: (1) optimize the performance of these control flow analyses, and (2) work with the Lucet team to make Lucet generated code more easily verifiable without compromising on performance.

Overfitting to Lucet. VeriWasm currently only checks x86-64 code generated by the Lucet Wasm compiler. While we have not extended VeriWasm to other platforms or compilers, we designed VeriWasm to be extensible and general: our verifier takes advantage of Wasm properties instead of Lucet's compilation of Wasm whenever possible. For example, VeriWasm verifies indirect calls by checking that the target of the call is loaded from the indirect function table.

The indirect function table is a Wasm concept, not a Lucet detail. But, the exact layout of the indirect call table (in native code) is compiler-specific and we would have to extend VeriWasm to accommodate layouts that differ from Lucet.

We would similarly have to extend VeriWasm to account for potentially more complex optimization passes that affect SFI checks. While Lucet's optimizations of SFI safety checks are the most advanced today, both Lucet and other Wasm compilers may implement more complex SFI optimizations in the future (e.g., following [ZTM11]) We are working to integrate VeriWasm and verification into the Wasm ecosystem and hope to co-evolve the verifier with the bytecode.

Unverified Wasm runtime. Wasm modules require a Wasm runtime that creates and manages memory regions for the module as well as provides secure access to syscalls. VeriWasm does not verify any part of this runtime and bugs in the runtime are out of scope of this work. However, bugs in SFI runtimes have been found in the past [Nat09] and their verification remains an open problem.

3.6 Related Work

In this section, we put VeriWasm in context of related work.

SFI system and verifier safety. Before Wasm, most SFI systems [WLAG93, MM06, YSD+09, SMB+10] were designed with binary validation in mind and came packaged with the corresponding verifier tools. These systems did not allow optimizations to move or eliminate SFI checks, allowing for extremely compact verifiers that can ensure binary safety by analyzing just a few instructions at a time [MM06]. Rocksalt [MTT+12] implements a fully verified version of such an SFI verifier, verifying both x86 disassembly and correctness of the analysis of Native Client [YSD+09] binaries. Tools like ERIM [VOED+19] that enforce fault isolation via hardware features can simplify validation more; for instance, the ERIM's verifier only has to check that the sandboxed component never uses byte sequences that decode to privileged hardware

instructions that disable sandboxing. Unlike these tools, VeriWasm validates natively-compiled Wasm modules where optimizations may have eliminated and moved security checks in order to improve performance, making validation more challenging. Thus, the VeriWasm verifier checks for security properties that are comparatively more complex (§3.2) at a per-function level, and relies on a verified correct analysis to provide a safe verifier; unlike RockSalt, VeriWasm does not verify correct x86 disassembly.

Other SFI systems also use schemes that require more complex validation. For instance, Strato [ZTE13] allows optimizations on SFI security checks, and provides a verifier that uses range analysis to ensure that pointers are correctly bounds checked (an approach other verifiers also use [ZTM11]). While VeriWasm employs a similar analysis (§3.3.1), it must also account for Wasm's trusted stack; for instance, the example in Section 3.3.5 shows how VeriWasm handles a case where security checks are elided when loading multiple times from the same location in the trusted stack.

Besson et al. focus on validating SFI binaries with a trusted stack [BJL18]. They model the semantics of and implement an SFI verifier using an abstract interpretation to validate stack and heap safety, an approach VeriWasm also follows. In addition, VeriWasm identifies and addresses several more requirements necessary for Wasm verification; in particular, VeriWasm validates in the presence of indirect calls and jumps, compiler optimizations, and dynamic values for heap-base address etc. by leveraging knowledge about Wasm and Wasm compilers.

Necula et al. present a more general approach for verifying complex security properties called proof carrying code [Nec97]. In this approach, compilers include a proof of safety along with a binary. While this allows validation of complex security properties, VeriWasm must work with the existing eco-system of Wasm modules, which does not include safety proofs.

Tan [Tan17] provides a more detailed history and analysis of earlier SFI tools and their validation toolchains.

Compilation and translation safety. An alternate approach to ensuring safety is to verify the

compiler itself. For instance, Kroll et al. [KSA14] implement a verified SFI system on top of the Compcert verified compiler [Ler09], and there is ongoing work to build a verified Wasm compiler [BLP20]. More generally, CompCert and its variations [Ler06, vVZN+13, SBCA15] and CakeML [KMNO14] are examples of foundationally verified compilers, and there are many examples of verified optimizations for these compilers [MZTG16, BDMT19].

Translation validation [Sam75, PSS98] is a different approach to compiler correctness; it proves that some property—most extremely, equivalence—holds before and after compiler optimizations. There's translation validation work for optimization passes in industrial compilers like gcc [Nec00] and LLVM [TLR20, TGM11, STL11]; there's also work on formally verified translation validation for CompCert in Coq [RL10, TL08, TL09, TL10].

Another approach is to provide a domain-specific language for writing verified compiler optimizations [LMNR15, LMC03, LMRC05, KTL09, BRN+20]; this doesn't verify the whole compiler, but verifies that a single optimization (or other) pass is correct.

The DSL approach can verify that optimization passes are correct, while translation validation can verify that a single compilation of a given program is correct; whole-compiler verification offers the strongest guarentees of end-to-end correctness. VeriWasm's approach trades-off completeness for easier proof burden and maintenance of the compiler toolchain. Specifically, since VeriWasm only verifies the verifier—a much simpler tool than a full compiler or optimization pass—it has a comparatively smaller proof burden. Futhermore, VeriWasm's approach allows changes to the compiler and optimization passes without changes (or with minor changes) to the verifier or the verifier proofs.

Retrofitting security checks. VeriWasm's analysis techniques are similar to some tools that retrofit security checks in binaries. For instance, ARMor [ZLDSR11] analyzes and rewrites ARM binaries to ensure SFI which is verified via abstract interpretation; Zhang et al.[ZS13] analyze and modify binary executables to apply control flow integrity without source code; StackArmor [CSA+15] rewrites the stack usage in binaries to prevent stack based attacks by

enforcing the safe stack policy [KSP+14]. Unlike these systems, which may conservatively add checks when analyses proves too complicated or does not terminate, VeriWasm operates without modifying the generated assembly. Additionally, VeriWasm leverages properties of Wasm to further simplify analysis—for instance, to ensure safety even in the presence of indirect jumps.

Abstract interpretation. Abstract interpretation [CC77] has been verifying program properties and finding bugs for over forty years. The Astrée static analyzer [CCF+05] has verified absence of certain errors in space vehicles [BCC+10], and many works [RRW05, VYY10] use abstract interpretation for everything from verification to synthesis. There are even verified static analysis passes and frameworks [BLMP13, JLB+15, BLP16]. Like these works, VeriWasm uses a verified abstract interpretation passes but specifically focuses on showing that binaries are safely sandboxed.

Bug finding. Bug finding tools can also identify security flaws. They use techniques like symbolic execution [CDE+08], concolic execution [GLM12, YLX+18, SMA05], fuzzing [YCER11, afl13], and binary instrumentation [NS07]. These tools find several classes of security bugs like use-after-frees [SBPV12], race conditions [SBN+97, SI09], stack overflows [RRW05], and more; some use fast but unsound analysis to quickly find bugs with low false positives [BBC+10]. In contrast, VeriWasm cannot check for general classes of security bugs and instead only validates the Wasm security properties; it uses a sound analysis and may produce false positives (§3.4).

3.7 Conclusion

Complex software systems have bugs, and compilers—with their enormous attack surface—are no exception. Luckily, real-world exploits organically resulting from flaws in code generation are rare because they require byzantine inputs atypical of real programs. However, once the compiler takes on the role of protecting the execution environment from the behavior of compiled software, this dynamic is reversed—any compiler flaw is available for an attacker to exploit. This

problem, in the context of Wasm, motivates our work.

Wasm offers the promise of high-performance, portable, safe code. But this promise of safety requires us to trust that the compiler inserts SFI checks in all necessary places and that these checks are correctly handled in all subsequent optimization passes. When this trust fails, so do all safety guarantees.

In this paper, we advocate an alternative approach—trust the compiler to do its job but, just in case, verify the safety properties of the native code it has compiled. We present VeriWasm, a tool to validate that Wasm binaries produced by the Lucet compiler do not have missing security checks that can break isolation. VeriWasm uses simple abstract interpretation passes to establish local per-function properties, which is sufficient to prove SFI safety of an entire Wasm binary. VeriWasm has validated over 22 million auto generated Wasm binaries with no false positives (so far), and is also able to validate real world Wasm binaries used in Fastly's edge cloud and in the Firefox browser.

3.8 Chapter acknowledgments

Chapter 3, contains material reprinted from the Network and Distributed System Security Symposium (NDSS) 2021. Johnson, Evan; Alhessi, Yousef; Thien, David; Narayan, Shravan; Brown, Fraser; Lerner, Sorin; McMullen, Tyler; Savage, Stefan; Stefan, Deian. NDSS, 2021. The dissertation author was one of the co-authors of this material.

The dissertation author made co-equal contributions to the design of VeriWasm and was responsible for the early prototypes of VeriWasm. Material from the NDSS publication pertaining to the formalization of VeriWasm are not included in this dissertation chapter.

Chapter 4

Swivel: Spectre-resistant sandboxing

WebAssembly (Wasm) is a portable bytecode originally designed to safely run native code (e.g., C/C++ and Rust) in the browser [HRS+17]. Since its initial design, though, Wasm has been increasingly used to sandbox untrusted code outside the browser. For example, Fastly and Cloudflare use Wasm to sandbox client applications running on their edge clouds—where multiple client applications run within a single process [Pat19, Var17]. Mozilla uses Wasm to sandbox third-party C/C++ libraries in Firefox [NDG+20, Fro20]. Yet others use Wasm to isolate untrusted code in serverless computing [HR19], IoT applications [Byt19b], games [Mic20b], trusted execution environments [Ena20], and even OS kernels [Sne18].

In this paper, we focus on hardening Wasm against Spectre attacks—the class of transient execution attacks which exploit control flow predictors [KHF+19]. Transient execution attacks which exploit features within the memory subsystem (e.g., Meltdown [LSG+18], MDS [SLM+19, CGG+19, vSMÖ+19], and Load Value Injection [VBMS+20]) are limited in scope and have already been fixed in recent microarchitectures [Int20c] (see Section 4.5.2). In contrast, Spectre can allow attackers to bypass Wasm's isolation boundary on almost all superscalar CPUs [AMD19, Int18a, App18]—and, unfortunately, current mitigations for Spectre cannot be implemented entirely in hardware [MST+19, YYK+19, BBZ+19, KSK+20, SZOC19,

Car18, JAS+20, TVT19].

On multi-tenant serverless, edge-cloud, and function as a service (FaaS) platforms, where Wasm is used as *the* way to isolate mutually distursting tenants, this is particulary concerning:[1] A malicious tenant can use Spectre to break out of the sandbox and read another tenant's secrets in two steps (§4.4.4). First, they *mistrain* different components of the underlying control flow prediction—the conditional branch predictor (CBP), branch target buffer (BTB), or return stack buffer (RSB)—to speculatively execute code that accesses data outside the sandbox boundary. Then, they *reconstruct* the secret data from the underlying microarchitectural state (typically the cache) using a side channel (cache timing).

One way to mitigate such Spectre-based *sandbox breakout attacks* is to partition mutually distrusting code into separate processes. By placing untrusted code in a separate process we can ensure that the attacker cannot access secrets. Chrome and Firefox, for example, do this by partitioning different *sites* into separate processes [Goo18, RMO19, Moz18b]. On a FaaS platform, we could similarly place tenants in separate processes.

Unfortunately, this would still leave tenants vulnerable to cross-process *sandbox poisoning attacks* [Int18b, CVBS+19, KKSAG18]. Specifically, attackers can poison hardware predictors to coerce a victim sandbox to speculatively execute gadgets that access secret data—from their own memory region—and leak it via the cache (e.g., by branching on the secret). Moreover, using process isolation would sacrifice Wasm's scalability (running many sandboxes within a process) and performance (cheap startup times and context switching) [Pat19, Var17, NDG+20, HR19].

The other popular approach, removing speculation within the sandbox, is also unsatisfactory. For example, using pipeline fences to restrict Wasm code to sequential execution imposes a 1.8×–7.3× slowdown on SPEC 2006 (§4.4). Conservatively inserting pipeline fences before every dynamic load—an approach inspired by the mitigation available in Microsoft's Visual Studio compiler [Mic20a]—is even worse: it incurs a 7.3×–19.6× overhead on SPEC (§4.4).

[1]Though our techniques are general, for simplicity we henceforth focus on Wasm as used on FaaS platforms.

In this paper, we take a compiler-based approach to hardening Wasm against Spectre, without resorting to process isolation or the use of fences. Our framework, Swivel, addresses not only sandbox breakout and sandbox poisoning attacks, but also *host poisoning attacks*, i.e., Spectre attacks that coerce the process hosting the Wasm sandboxes into leaking sensitive data. That is, Swivel ensures that a malicious Wasm tenant cannot speculatively access data outside their sandbox nor coerce another tenant or the host to divulge secrets of other sandboxes via poisoning. We develop Swivel via three contributions:

1. Software-only Spectre hardening (§4.2.2) Our first contribution, Swivel-SFI, is a software-only approach to hardening Wasm against Spectre. Swivel-SFI eliminates sandbox breakout attacks by compiling Wasm code to *linear blocks (LBs)*. Linear blocks are straight-line x86 code blocks that satisfy two invariants: (1) all transfers of control, including function calls, are at the block boundary—to (and from) other linear blocks; and (2) all memory accesses within a linear block are masked to the sandbox memory. These invariants are necessary to ensure that the speculative control and data flow of the Wasm code is restricted to the sandbox boundary. They are not sufficient though: Swivel-SFI must also be tolerant of possible RSB underflow. We address this by (1) not emitting `ret` instructions and therefore completely bypassing the RSB and (2) using a *separate stack* for return addresses.

To address poisoning attacks, Swivel-SFI must still account for a poisoned BTB or CBP. Since these attacks are more sophisticated, we evaluate two different ways of addressing them, and allow tenants to choose between them according to their trust model. The first approach uses address space layout randomization (ASLR) to randomize the placement of each Wasm sandbox and flushes the BTB on each sandbox boundary crossing. This does not eliminate poisoning attacks; it only raises the bar of Wasm isolation to that of process isolation. Alternately, tenants can opt to eliminate these attacks altogether; to this end, our deterministic Swivel-SFI rewrites conditional branches to indirect jumps—thereby completely bypassing the CBP (which cannot be

directly flushed) and relying solely on the BTB (which can).

2. Hardware-assisted Spectre hardening (§4.2.3) Our second contribution, Swivel-CET, restores the use of all predictors, including the RSB and CBP, and partially obviates the need for BTB flushing. It does this by sacrificing backwards compatibility and using new hardware security extensions: Intel's Control-flow Enforcement Technology (CET) [Int20a] and Memory Protection Keys (MPK) [Int20a].

Like Swivel-SFI, Swivel-CET relies on linear blocks to address sandbox breakout attacks. But Swivel-CET does not avoid `ret` instructions. Instead, we use Intel® CET's hardware *shadow stack* to ensure that the RSB cannot be misused to speculatively return to a location that is different from the expected function return site on the stack [Int20a].

To eliminate host poisoning attacks, we use both Intel® CET and Intel® MPK. In particular, we use Intel® MPK to partition the application into two domains—the host and Wasm sandbox(es)—and, on context switch, ensure that each domain can only access its own memory regions. We use Intel® CET forward-edge control-flow integrity to ensure that application code cannot jump, sequentially or speculatively, into arbitrary sandbox code (e.g., due to a poisoned BTB). We do this by inserting `endbranch` instructions in Wasm sandboxes to demarcate valid jump targets, essentially partitioning the BTB into two domains. Our use of Intel's MPK and CET ensures that even if the host code runs with poisoned predictors, it cannot read—and thus leak—sandbox data.

Since Intel® MPK only supports 16 protection regions, we cannot use it to similarly prevent sandbox poisoning attacks: serverless, edge-cloud, and FaaS platforms have thousands of co-located tenants. Hence, to address sandbox poisoning attacks, like for Swivel-SFI, we consider and evaluate two designs. The first (again) uses ASLR to randomize the location of each sandbox and flushes the BTB on sandbox entry; we don't flush on sandbox exit since the host can safely run with a poisoned BTB. The second is deterministic and not only allows using

conditional branches but also avoids BTB flushes. It does this using a new technique, *register interlocking*, which tracks the control flow of the Wasm sandbox and turns every misspeculated memory access into an access to an empty guard page. Register interlocking allows a tenant to run with a poisoned BTB or CBP since any potentially poisoned speculative memory accesses will be invalidated.

3. Implementation and evaluation (§4.3–4.4) We implement both Swivel-SFI and Swivel-CET by modifying the Lucet compiler's Wasm-to-x86 code generator (Cranelift) and runtime. To evaluate Swivel's security we implement proof of concept breakout and poisoning attacks against stock Lucet (mitigated by Swivel). We do this for all three Spectre variants, i.e., Spectre attacks that abuse the CBP, BTB, and RSB.

We evaluate Swivel's performance against stock Lucet and several fence-insertion techniques on several standard benchmarks. On the Wasm compatible subset of the SPEC 2006 CPU benchmarking suite we find the ASLR variants of Swivel-SFI and Swivel-CET impose little overhead—they are at most 10.3% and 6.1% slower than Lucet, respectively. Our deterministic implementations, which eliminate all three categories of attacks, incur modest overheads: Swivel-SFI and Swivel-CET are respectively 3.3%–86.1% (geomean: 47.3%) and 8.0%–240.2% (geomean: 96.3%) slower than Lucet. These overheads are smaller than the overhead imposed by state-of-the-art fence-based techniques.

Open source and data We make all source and data available under an open source license at: https://swivel.programming.systems.

4.1 A brief overview of Wasm and Spectre

In this section, we give a brief overview of WebAssembly's use in multi-tenant serverless and edge-cloud computing platforms—and more generally function as a service (FaaS) platforms.

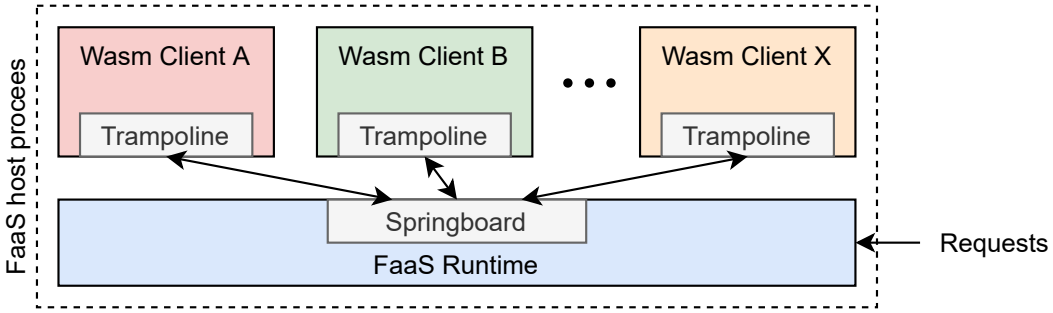


Figure 4.1: FaaS platform using Wasm to isolate mutually distrusting tenants.

In particular, we describe how FaaS platforms use Wasm as an intermediate compilation layer for isolating different tenants today. We then briefly review Spectre attacks and describe how today's approach to isolating Wasm code is vulnerable to this class of attacks.

4.1.1 WebAssembly

Wasm is a low-level 32-bit machine language explicitly designed to embed C/C++ and Rust code in host applications. Wasm programs (which are simply a collection of functions) are (1) deterministic and well-typed, (2) follow a structured control flow discipline, and (3) separate the heap—the *linear memory*—from the well-typed stack and program code. These properties make it easy for compilers like Lucet to sandbox Wasm code and safely embed it within an application [HRS$^+$17].

Control flow safety Lucet's code generator, Cranelift, ensures that the control flow of the compiled code is restricted to the sandbox and cannot be bent to bypass bounds checks (e.g., via return-oriented programming). This comes directly from preserving Wasm's semantics during compilation. For example, compiled code preserves Wasm's safe stack [HRS$^+$17, KSP$^+$14], ensuring that stack frames (and thus return values on the stack) cannot be clobbered. The compiled code also enforces Wasm's coarse-grained CFI and, for example, matches the type of each indirect call site with the type of the target.

Memory isolation When Lucet creates a Wasm sandbox, it reserves 4GB of virtual memory for the Wasm heap and uses Cranelift to bound all heap loads and stores. To this end, Cranelift (1) explicitly passes a pointer to the base of the heap as the first argument to each function and (2) masks all pointers to be within this 4GB range. Like previous software-based isolation (SFI) systems [WLAG93], Cranelift avoids expensive bounds check operations by using guard pages to trap any offsets that may reach beyond the 4GB heap space.

Embedding Wasm An application with embedded Wasm code will typically require context switching between the Wasm code and host—e.g., to read data from a socket. Lucet allows safe control and data flow across the host-sandbox boundary via *springboards* and *trampolines*. Springboards are used to enter Wasm code—they set up the program context for Wasm execution—while trampolines are used to restore the host context and resume execution in the host.

Using Wasm in FaaS platforms WebAssembly FaaS platforms like Fastly's Terrarium allow clents to deploy scalable function-oriented Web and cloud applications written in any langauge (that can be compiled to Wasm). Clients compile their code to Wasm and upload the resulting module to the platform; the platform handles scaling and isolation. As shown in Figure 4.1, FaaS platforms place thousands of client Wasm modules within a single host process, and distribute these processes across thousands of servers and multiple datacenters. This is the key to scaling—it allows any host process, in any datacenter, to spawn a fresh Wasm sandbox instance and run any client function in response to a user request. By using Wasm as an intermediate layer, FaaS platforms isolate the client for free [WLAG93, Pat19, NGL$^+$19, BLP20, McM20, NDG$^+$20]. Unfortunately, this isolation does not hold in the presence of Spectre attacks.

4.1.2 Spectre attacks

Spectre attacks exploit hardware predictors to induce mispredictions and speculatively execute instructions—*gadgets*—that would not run sequentially [KHF+19]. Spectre attacks are classified by the hardware predictor they exploit [CVBS+19]. We focus on the three Spectre variants that hijack control flow:

- **Spectre-PHT** Spectre-PHT [KHF+19] exploits the *pattern history table* (PHT), which is used as part of the *conditional branch predictor* (CBP) to guess the direction of a conditional branch while the condition is still being evaluated. In a Spectre-PHT attack, the attacker (1) pollutes entries in the PHT so that a branch is mispredicted to the wrong path. The attacker can then use this wrong-path execution to bypass memory isolation guards or control flow integrity.
- **Spectre-BTB** Spectre-BTB [KHF+19] exploits the *branch target buffer* (BTB), which is used to predict the target of an indirect jump [ZKE20]. In a Spectre-BTB attack, the attacker pollutes entries in the BTB, redirecting speculative control flow to an arbitrary target. Spectre-BTB can thus be used to speculatively execute gadgets that are not in the normal execution path (e.g., to carry out a ROP-style attack).
- **Spectre-RSB** Spectre-RSB [MR18, KKSAG18] exploits the *return stack buffer* (RSB), which memorizes the location of recently executed `call` instructions to predict the targets of `ret` instructions. In a Spectre-RSB attack, the attacker uses chains of `call` or `ret` instructions to over- or underflow the RSB, and redirect speculative control flow in turn.

Spectre can be used *in-place* or *out-of-place* [CVBS+19]. In an in-place attack, the attacker mistrains the prediction for a victim branch by repeatedly executing the victim branch itself. In an out-of-place attack, the attacker finds a secondary branch that is *congruent* to the victim branch—predictor entries are indexed using a subset of address bits—and uses this secondary branch to mistrain the prediction for the victim branch.

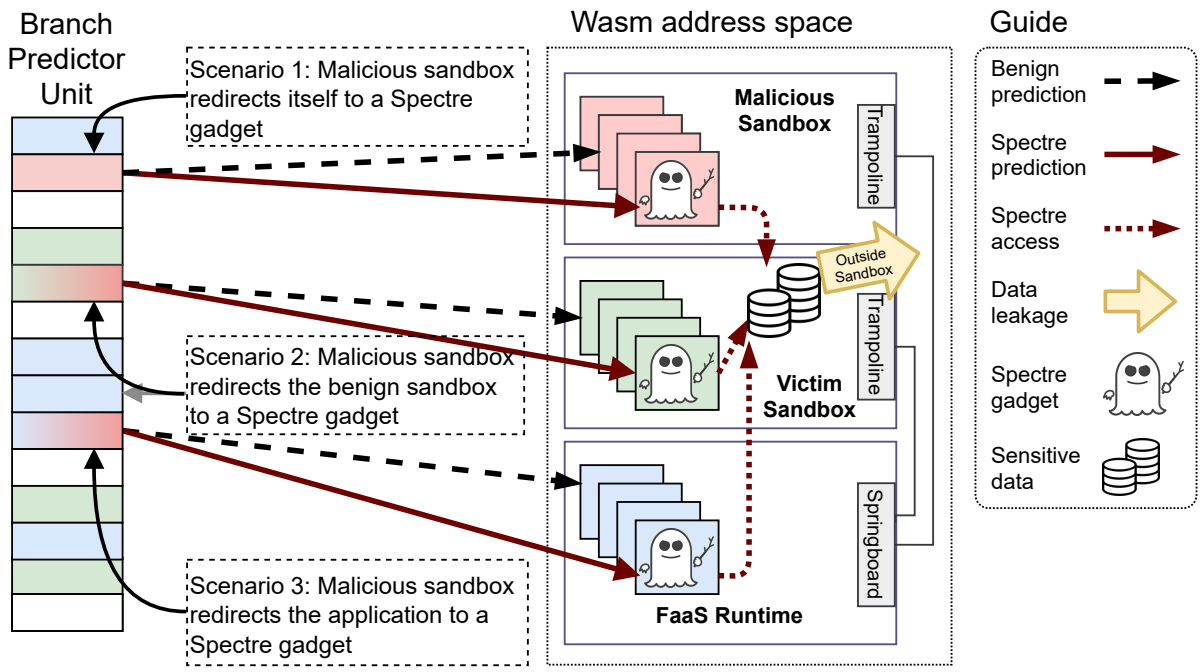


Figure 4.2: A malicious tenant can fill branch predictors with invalid state (red). In one scenario, the attacker causes its own branches to speculatively execute code that access memory outside of the sandbox. In the second and third scenarios, the attacker uses Spectre to respectively target a victim sandbox or the host runtime to misspeculate and leak secret data.

4.1.3 Spectre attacks on FaaS platforms

A malicious FaaS platform client who can upload arbitrary Wasm code can force the Wasm compiler to emit native code which is safe during sequential execution, but uses Spectre to bypass Wasm's isolation guarantees during speculative execution. We identify three kinds of attacks (Figure 4.2):

- **Scenario 1: Sandbox breakout attacks** The attacker bends the speculative control flow of their own module to access data outside the sandbox region. For example, they can use Spectre-PHT to bypass conditional bounds checks when accessing the indirect call table. Alternatively, they can use Spectre-BTB to transfer the control flow into the middle of instructions to execute unsafe code (e.g., code that bypasses Wasm's implicit heap bounds checks).

```
1 mov    rdx,QWORD PTR [fn_table_len]  ; get fn table length
2 cmp    rcx,rdx  ; check that rcx is in-bounds
3 jb     index_ok
4 ud2  ; trap otherwise
5 index_ok:
6 lea    rdx,[fn_table]
7 mov    rcx,QWORD PTR [rdx+rcx*4]
8 call   rcx
```

Figure 4.3: A simplified snippet of the vulnerable code from our Spectre-PHT breakout attack. This code is safe during sequential execution (it checks the index `rcx` before using it to load a function table entry). But, during speculative execution, control flow may bypass this check and access memory outside the function table bounds.

- **Scenario 2: Sandbox poisoning attacks** The attacker uses an out-of-place Spectre attack to bend the control flow of a victim sandbox and coerce the victim into leaking their own data. Although this attack is considerably more sophisticated, we were still able to implement proof of concept attacks following Canella et al. [CVBS+19]. Here, the attacker finds a (mispredicted) path in the victim sandbox that leads to the victim leaking data, e.g., through cache state. They then force the victim to mispredict this path by using a congruent branch within their own sandbox.
- **Scenario 3: Host poisoning attacks** Instead of bending the control flow of the victim sandbox, the attacker can use an out-of-place Spectre attack to bend the control flow of the host runtime. This allows the attacker to speculatively access data from the host as well as any other sandbox.

Figure 4.3 gives an example sandbox breakout gadget. The gadget is in the implementation of the Wasm `call_indirect` instruction, which is used to call functions indexed in a module-level function table. This code first compares the function index `rcx` to the length of the function table (to ensure that `rcx` points to a valid entry). If `rcx` is valid, it then jumps to `index_ok`, loads the function from the corresponding entry in the table, and calls it; otherwise the code traps.

An attacker can mistrain the conditional branch on line 3 and cause it to speculatively jump to `index_ok` even when `rcx` is out-of-bounds. By controlling the contents of `rcx`, the attacker

Table 4.1: Effectiveness of Swivel against different Spectre variants. A full circle indicates that Swivel eliminates the attack while a half circle indicates that Swivel only mitigates the attack.

Attack variant		Swivel-SFI		Swivel-CET	
		ASLR	Det	ASLR	Det
Spectre-PHT	in-place	●	●	●	●
	out-of-place	◐	●	◐	●
Spectre-BTB	in-place	●	●	●	●
	out-of-place	●	●	●	●
Spectre-RSB	in-place	●	●	●	●
	out-of-place	●	●	●	●

can thus execute arbitrary code locations outside the sandbox. In Section 4.4.4 we demonstrate several proof of concept attacks, including a breakout attack that uses this gadget. These attacks serve to highlight the importance of hardening Wasm against Spectre.

4.2 Swivel: Hardening Wasm against Spectre

Swivel extends Lucet—and the underlying Cranelift code generator—to address Spectre attacks on FaaS Wasm platforms. We designed Swivel with several goals in mind. First, *performance*: Swivel minimizes the number of pipeline fences it inserts, allowing Wasm to benefit from speculative execution as much as possible. Second, *automation*: Swivel does not rely on user annotations or source code changes to guide mitigations; we automatically apply mitigations when compiling Wasm. Finally, *modularity*: Swivel offers configurable protection, ranging from probabilistic schemes with high performance to thorough mitigations with strong guarantees. This allows Swivel users to choose the most appropriate mitigations (see Table 4.1) according to their application domain, security considerations, or particular hardware platform.

In the rest of this section, we describe our attacker model and introduce a core abstraction: *linear blocks*. We then show how linear blocks, together with several other techniques, are used to address both sandbox breakout and poisoning attacks. These techniques span two Swivel designs: Swivel-SFI, a software-only scheme which provides mitigations compatible with existing CPUs;

and Swivel-CET, which uses hardware extensions (Intel® CET and Intel® MPK) available in the 11th generation Intel® CPUs.

Attacker model We assume that the attacker is a FaaS platform client who can upload arbitrary Wasm code which the platform will then compile and run alongside other clients using Swivel. The goal of the attacker is to read data sensitive to another (victim) client using Spectre attacks. In the Swivel-CET case, we only focus on exfiltration via the data cache—and thus assume an attacker who can only exploit gadgets that leak via the data cache. We consider transient attacks that exploit the memory subsystem (e.g., Meltdown [LSG+18], MDS [SLM+19, CGG+19, vSMÖ+19], and LVI [VBMS+20]) out of scope and discuss this in detail in Section 4.5.2.

We assume that our Wasm compiler and runtime are correct. We similarly assume the underlying operating system is secure and the latest CPU microcode updates are applied. We assume hyperthreading is disabled for any Swivel scheme except for the deterministic variant of Swivel-CET. Consistent with previous findings [ZKE20], we assume BTBs predict the lower 32-bits of target addresses, while the upper 32-bits are inferred from the instruction pointer.

Swivel addresses attackers that intentionally extract information using Spectre. We do not prevent clients from accidentally leaking secrets during sequential execution and, instead, assume they use techniques like constant-time programming to prevent such leaks [CDG+20]. For all Swivel schemes except the deterministic variant of Swivel-CET, we assume that a sandbox cannot directly invoke function calls in other sandboxes, i.e., it cannot control the input to another sandbox to perform an in-place poisoning attack. We lastly assume that host secrets can be protected by placing them in a Wasm sandbox, and discuss this further in Section 4.5.1.

4.2.1 Linear blocks: local Wasm isolation

To enforce Wasm's isolation sequentially and speculatively, Swivel-SFI and Swivel-CET compile Wasm code to linear blocks (LBs). Linear blocks are straight-line code blocks that

do not contain *any* control flow instructions except for their terminators—this is in contrast to traditional basic blocks, which typically do not consider function calls as terminators. This simple distinction is important: It allows us to ensure that *all* control flow transfers—both sequential and speculative—land on linear block boundaries. Then, by ensuring that individual linear blocks are *safe*, we can ensure that whole Wasm programs, when compiled, are confined and cannot violate Wasm's isolation guarantees.

A linear block is safe if, independent of the control flow into the block, Wasm's isolation guarantees are preserved. In particular, we cannot rely on safety checks (e.g., bounds checks for memory accesses) performed across linear blocks since, speculatively, blocks may not always execute in sequential order (e.g., because of Spectre-BTB). When generating native code, Swivel ensures that a linear block is safe by:

Masking memory accesses Since we cannot make any assumptions about the initial contents of registers, Swivel ensures that unconditional heap bounds checks (performed via masking) are performed in the same linear block as the heap memory access itself. We do this by modifying the Cranelift optimization passes which could lift bounds checks (e.g., loop invariant code motion) to ensure that they don't move masks across linear block boundaries. Similarly, since we cannot trust values on the stack, Swivel ensures that any value that is unspilled from the stack and used in a bounds check is masked again. We use this *mask-after-unspill* technique to replace Cranelift's unsafe mask-before-spill approach.

Pinning the heap registers To properly perform bounds checks for heap memory accesses, a Swivel linear block must determine the correct value of the heap base. Unfortunately, as described above, we cannot make any assumptions about the contents of any register or stack slot. Swivel thus reserves one register, which we call the *pinned heap register*, to store the address of the sandbox heap. Furthermore, Swivel prevents any instructions in the sandbox from altering the pinned heap register. This allows each linear block to safely assume that the pinned heap register

holds the correct value of the heap base, even when the speculative control flow of the program has gone awry due to misprediction.

Hardening jump tables Wasm requires bounds checks on each access to indirect call tables and switch tables. Swivel ensures that each of these bounds checks is local to the linear block where the access is performed. Moreover, Swivel implements the bounds check using *speculative load hardening* [Car18], masking the index of the table access with the length of the table. This efficiently prevents the attacker from speculatively bypassing the bounds check.

Swivel does not check the indirect jump targets beyond what Cranelift does. At the language level, Wasm already guarantees that the targets of indirect jumps (i.e., the entries in indirect call tables and switch tables) can only be the tops of functions or switch-case targets. Compiled, these correspond to the start of Swivel linear blocks. Thus, an attacker can only train the BTB with entries that point to linear blocks, which, by construction, are safe to execute in any context.

Protecting returns Wasm's execution stack—in particular, return addresses on the stack—cannot be corrupted during sequential execution. Unfortunately, this does not hold speculatively: An attacker can write to the stack (e.g., with a buffer overflow) and speculatively execute a return instruction which will divert the control flow to their location of choice. Swivel ensures that return addresses on the stack cannot be corrupted as such, even speculatively. We do this using a *separate stack* or *shadow stack* [BZP19], as we detail below.

4.2.2 Swivel-SFI

Swivel-SFI builds on top of linear blocks to address all three classes of attacks.

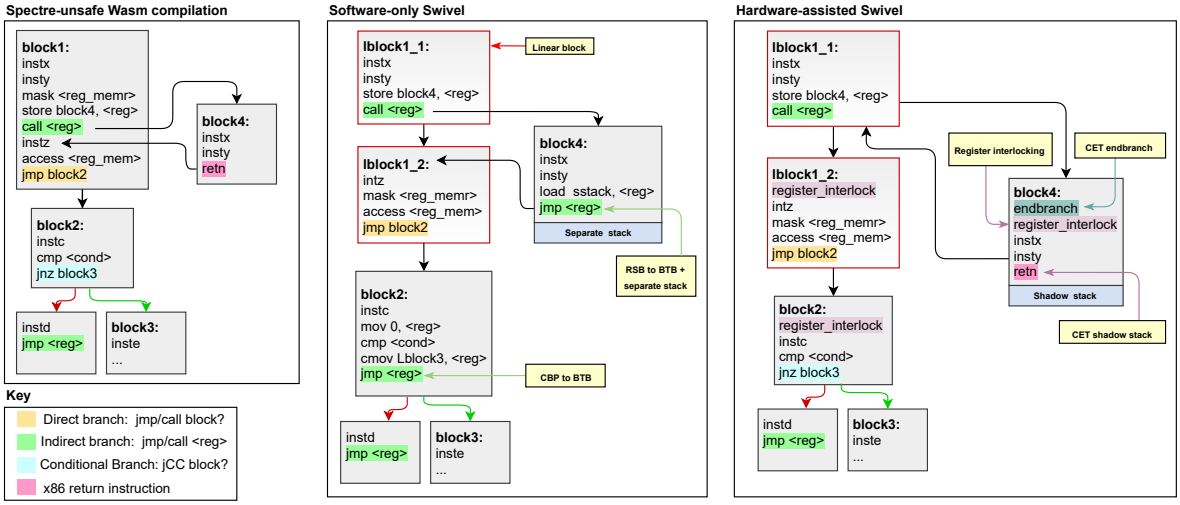


Figure 4.4: Swivel hardens Wasm against spectre via compiler transformations. In Swivel-SFI, we convert basic blocks to linear blocks. Each linear block (e.g., `lblock1_1` and `lblock1_2`) maintains local security guarantees for speculative execution. Then, we protect the backward edge (`block4`) by replacing the return instructions and using a separate return stack. To eliminate poisoning attacks, in the deterministic version of Swivel-SFI, we further encode conditional branches as indirect jumps. In Swivel-CET, we similarly use linear blocks, but we allow return instruction, and protect returns using the hardware shadow stack. To reduce BTB flushes, we additionally use Intel® CET's `endbranch` to ensure that targets of indirect branches land at the beginning of linear blocks. In the deterministic version, we avoid BTB flushing and instead use register interlocking to prevent leakage on misspeculated paths.

Addressing sandbox breakout attacks

Compiling Wasm code to linear blocks eliminates most avenues for breaking out of the sandbox. The only way for an attacker to break out of the sandbox is to speculatively jump into the middle of a linear block. We prevent this with:

The separate stack We protect returns by preserving Wasm's safe return stack during compilation. Specifically, we create a separate stack in a fixed memory location, but outside the sandbox stack and heap, to ensure that it cannot be overwritten by sandboxed code. We replace every call instruction with an instruction sequence that stores the address of the subsequent instruction—the return address—to the next entry in this separate stack. Similarly, we replace every return instruction with a sequence that pops the address off the separate stack and jumps to

that location. To catch under- and over-flows, we surround the separate stack with guard pages.

BTB flushing The other way an attacker can jump into the middle of a linear block is via a mispredicted BTB entry. Since all indirect jumps inside a sandbox can only point to the tops of linear blocks, any such entries can only be set via a congruent entry outside any sandbox—i.e., an attacker must orchestrate the host runtime into mistraining a particular jump. We prevent such attacks by flushing the BTB on transitions into and out of the sandbox.[2]

Addressing sandbox and host poisoning attacks

There are two ways for a malicious sandbox to carry out poisoning attacks: By poisoning CBP or BTB entries. Since we already flush the BTB to address sandbox breakout attacks, we trivially prevent all BTB poisoning. Addressing CBP poisoning is less straightforward. We consider two schemes:

Mitigating CBP poisoning To mitigate CBP-based poisoning attacks, we use ASLR to randomize the layout of sandbox code pages. This mitigation is not sound—it is theoretically possible for an attacker to influence a specific conditional branch outside of the sandbox. As we discuss in Section 4.2.4, this raises the bar to (at least) that of process isolation: The attacker would would have to (1) de-randomize the ASLR of both their own sandbox and the victim's and (2) find useful gadgets, which is itself an open problem (§4.6).

Eliminating CBP poisoning Clients that are willing to tolerate modest performance overheads (§4.4) can opt to eliminate poisoning attacks. We eliminate poisoning attacks by removing conditional branches from Wasm sandboxes altogether. Following [LSG+17], we do this by using

[2] In practice, BTB predictions are not absolute (as discussed in our attacker model), instead they are 32-bit offsets relative to the instruction pointer [ZKE20]. To ensure that this does not result in predictions at non linear block boundaries, we restrict the sandbox code size to 4GB.

the `cmov` conditional move instruction to encode each conditional branch as an indirect branch with only two targets (Figure 4.4).

4.2.3 Swivel-CET

Swivel-SFI avoids using fences to address Spectre attacks, but ultimately bypasses all but the BTB predictors—and even then we flush the BTB on every sandbox transition. Swivel-CET uses Intel® CET [Int20a] and Intel® MPK [Int20a] to restore the use of the CBP and RSB, and avoid BTB flushing.[3]

Addressing sandbox breakout attacks

Like Swivel-SFI, we build on linear blocks to address sandbox breakout attacks (Figure 4.4). Swivel-CET, however, prevents an attacker from speculatively jumping into the middle of a linear block using:

The shadow stack Swivel-CET uses Intel® CET's *shadow stack* to protect returns. Unlike Swivel-SFI's separate stack, the shadow stack is a hardware-maintained stack, distinct from the ordinary data stack and inaccessible via standard load and store instructions. The shadow stack allows us to use call and return instructions as usual—the CPU uses the shadow stack to check the integrity of return addresses on the program stack during both sequential and speculative execution.

Forward-edge CFI Instead of flushing the BTB, Swivel-CET uses Intel® CET's coarse-grained control flow integrity (CFI) [ABEL09] to ensure that sandbox code can only jump to the top of a linear block. We do this by placing an `endbranch` instruction at the beginning of every linear block that is used as an indirect target (e.g., the start of a function that is called indirectly).

[3] Appendix 4.8.1 gives a brief introduction to these new hardware features.

During speculative execution, if the indirect branch predictor targets an instruction other than an `endbranch` (e.g., inside the host runtime), the CPU stops speculating [Int20a].

Conditional BTB flushing When using ASLR to address sandbox poisoning attacks, we still need to flush the BTB on transitions into each sandbox. Otherwise, one sandbox could potentially jump to a linear block in another sandbox. Our deterministic approach to sandbox poisoning (described below), however, eliminates BTB flushes altogether.

Addressing host poisoning attacks

To prevent host poisoning attacks, Swivel-CET uses Intel® MPK. Intel® MPK exposes new user mode instructions that allow a process to partition its memory into sixteen linear regions and to selectively enable/disable read/write access to any of those regions. Swivel-CET uses only two of these protection domains—one for the host and one shared by all sandboxes—and switches domains during the transitions between host and sandbox. When the host creates a new sandbox, Swivel-CET allocates the heap memory for the new sandbox with the sandbox protection domain, and then relinquishes its own access to that memory so that it is no longer accessible by the host. This prevents host poisoning attacks by ensuring that the host cannot be coerced into leaking secrets from another sandbox. We describe how we safely copy data across the boundary later (§4.3).

Addressing sandbox poisoning attacks

By poisoning CBP or BTB entries, a malicious sandbox can coerce a victim sandbox into executing a gadget that leaks sensitive data. As with Swivel-SFI, we consider both a probabilistic and deterministic design to addressing these attacks. Since the probabilistic approach is like Swivel-SFI's, we describe only the deterministic design.

Preventing leaks under poisoned execution Swivel-CET does *not* eliminate cross-sandbox CBP or BTB poisoning. Instead, we ensure that a victim sandbox cannot be coerced into leaking data via the cache when executing a mispredicted path. To leak secrets through the cache, the attacker must maneuver the secret data to a gadget that will use it as an offset into a memory region. In Cranelift, any such gadget will use the heap, as stack memory is always accessed at constant offsets from the stack pointer (which itself cannot be directly assigned). We thus need only prevent leaks that are via the Wasm heap—we do this using *register interlocks*.

Register interlocking Our register interlocking technique tracks the control flow of a Wasm program and prevents it from accessing its stack or heap when the speculative path diverges from the sequential path. We first assign each non-trivial linear block a unique 64-bit *block label*. We then calculate the expected block label of every direct or indirect branch and assign this value to a reserved *interlock register* prior to branching. At the beginning of each linear block, we check that the value of the interlock register corresponds to the static block label using `cmov` instructions. If the two do not match, we zero out the heap base register as well as the stack register. Finally, we unmap pages from the address space to ensure that any access from a zero heap or stack base will fault—and thus will not affect cache state.

The register interlock fundamentally introduces a data dependency between memory operations and the resolution of control flow. In doing so, we prevent any memory operations that would result in cache based leaks, but do not prevent all speculative execution. In particular, any arithmetic operations may still be executed speculatively. This is similar to hardware taint tracking [YYK+19], but enforced purely through compiler changes.

Finally, Wasm also stores certain data (e.g., globals variables and internal structures) outside the Wasm stack or heap. To secure these memory accesses with the register interlock, we introduce an artificial heap load in the address computation for this data.

Table 4.2: Breakdown of Swivel's individual protection techniques which, when combined, address the thee different class of attacks on Wasm (§4.1.3). For each technique we also list (in brackets) the underlying predictors.

Swivel protection and technique	Swivel-SFI		Swivel-CET	
	ASLR	Det	ASLR	Det
Sandbox breakout protections				
- Linear blocks [CBP, BTB, RSB]	✓	✓	✓	✓
- BTB flush in springboard [BTB]	✓	✓		
- Separate control stack [RSB]	✓	✓		
- CET endbranch [BTB]			✓	✓
- CET shadow stack [RSB]			✓	✓
Sandbox poisoning protections				
- BTB flush in springboard [BTB]	✓	✓	✓	
- Code page ASLR [CBP]	✓		✓	
- Direct branches to indirect [CBP]		✓		
- Register interlock [CBP, BTB]				✓
Host poisoning protections				
- Separate control stack [RSB]	✓	✓		
- Code page ASLR [CBP]	✓			
- BTB flush in trampoline [BTB]	✓	✓		
- Direct branches to indirect [CBP]		✓		
- Two domain MPK [CBP]			✓	✓

4.2.4 Security and performance trade-offs

Swivel offers two design points for protecting Wasm modules from Spectre attacks: Swivel-SFI and Swivel-CET. For each of these schemes we further consider probabilistic (ASLR) and deterministic techniques. In this section, we discuss the performance and security trade-offs when choosing between these various Swivel schemes.

Probabilistic or deterministic?

Table 4.1 summarizes Swivel's security guarantees. Swivel's deterministic schemes eliminate Spectre attacks, while the probabilistic schemes eliminate Spectre attacks that exploit the BTB and RSB, but trade-off security for performance when it comes to the CBP (§4.4): Our probabilistic schemes only *mitigate* Spectre attacks that exploit the CBP.

To this end, (probabilistic) Swivel hides branch offsets by randomizing code pages.

Previously, similar fine-grain approaches to address randomization have been proposed to mitigate attacks based on return-oriented programming [CVS+15, GAS+17]. Specifically, when loading a module, Swivel copies the code pages of the Wasm module to random destinations, randomizing all but the four least significant bits (LSBs) to keep 16-byte alignment. This method is more fine-grained than page remapping, which would fail to randomize the lower 12 bits for 4KB instruction pages.

Unfortunately, only a subset of address bits are typically used by hardware predictors. Zhang et. al [ZKE20], for example, found that only the 30 LSBs of the instruction address are used as input for BTB predictors. Though a similar study has not been conducted for the CBP, if we pessimistically assume that 30 LSBs are used for prediction then our randomization offers at least 26 bits of entropy. Since the attacker must de-randomize both their module and the victim module, this is likely higher in practice.

As we show in Section 4.4, the ASLR variants of Swivel are faster than the deterministic variants. Using code page ASLR imposes less overhead than the deterministic techniques (summarized in Table 4.2). This is not surprising: CBP conversion (in Swivel-SFI) and register interlocking (in Swivel-CET) are the largest sources of performance overhead.

For many application domains, this security-performance trade-off is reasonable. Our probabilistic schemes use ASLR only to mitigate sandbox poisoning attacks—and unlike sandbox breakout attacks, these attacks are significantly more challenging for an attacker to carry out: They must conduct an out-of-place attack on a specific target while accounting for the unpredictable mapping of the branch predictor. To our knowledge, such an attack has not been demonstrated, even without the additional challenges of defeating ASLR.

Furthermore, on a FaaS platform, these attacks are even harder to pull off, as the attacker has only a few hundred milliseconds to land an attack on a victim sandbox instance before it finishes—and the next victim instance will have entirely new mappings. Previous work suggests that such an attack is not practical in such a short time window [ERAGP18].

For other application domains, the overhead of the deterministic Swivel variants may yet be reasonable. As we show in Section 4.4, the average (geometric) overhead of Swivel-SFI is 47.3% and that of Swivel-CET is 96.3%. Moreover, users can choose to use Swivel-SFI and Swivel-CET according to their trust model—Swivel allows sandboxes of both designs to coexist within a single process.

Software-only or hardware-assisted?

Swivel-SFI and Swivel-CET present two design points that have different trade-offs beyond backwards compatibility. We discuss their trade-offs, focusing on the deterministic variants.

Swivel-SFI eliminates Spectre attacks by allowing speculation only via the BTB predictor and by controlling BTB entries through linear blocks and BTB flushing. Swivel-CET, on the other hand, allows the other predictors. To do this safely though, we use register interlocking to create data dependencies (and thus prevent speculation) on certain operations after branches. Our interlock implementation only guards Wasm memory operations—this means that, unlike Swivel-SFI, Swivel-CET only prevents *cache-based* sandbox poisoning attacks. While non-memory instructions (e.g., arithmetic operations) can still speculatively execute, register interlocks sink performance: Indeed, the overall performance overhead of Swivel-CET is higher than Swivel-SFI (§4.4).

At the same time, Swivel-CET can be used to handle a more powerful attacker model (than our FaaS model). First, Swivel-CET eliminates poisoning attacks even in the presence of attacker-controlled input. This is a direct corollary of being able to safely execute code with poisoned predictors. Second, Swivel-CET (in the deterministic scheme) is safe in the presence of hyperthreading; our other Swivel schemes assume that hyperthreading is disabled (§4.2). Swivel-CET allows hyperthreading because it doesn't rely on BTB flushing; it uses register interlocking to eliminate sandbox poisoning attacks. In contrast, our SFI schemes require the

BTB to be isolated for the host and each Wasm sandbox—an invariant that may not hold if, for example, hyperthreading interleaves host application and Wasm code on sibling threads.

4.3 Implementation

We implement Swivel on top of the Lucet Wasm compiler and runtime [McM20, Pat19]. In this section, we describe our modifications to Lucet.

We largely implement Swivel-SFI and Swivel-CET as passes in Lucet's code generator Cranelift. For both schemes, we add support for pinned heap registers and add direct `jmp` instructions to create linear block boundaries. We modify Cranelift to harden switch-table and indirect-call accesses: Before loading an entry from either table, we truncate the index to the length of the table (or the next power of two) using a bitwise mask. We also modify Lucet's stack overflow protection: Lucet emits conditional checks to ensure that the stack does not overflow; these checks are rare and we simply use `lfence`s.

We modify the springboard and trampoline transition functions in the Lucet runtime. Specifically, we add a single `lfence` to each transition function since we must disallow speculation from crossing the host-sandbox boundary.

The deterministic defenses for both Swivel-SFI and Swivel-CET—CBP conversion and register interlocks—increase the cost of conditional control flow. To reduce the number of conditional branches, we thus enable explicit loop unrolling flags when compiling the deterministic schemes.[4] This is not necessary for the ASLR-based variants since they do not modify conditional branches. Indeed, the ASLR variants are straightforward modifications to the dynamic library loader used by the Lucet runtime: Since all sandbox code is position independent, we just copy a new sandbox instance's code and data pages to a new randomized location in memory.

We also made changes specific to each Swivel scheme:

[4] For simplicity, we do this in the Clang compiler when compiling applications to Wasm and not in Lucet proper.

Swivel-SFI We augment the Cranelift code generation pass to replace `call` and `return` instructions with the Swivel-SFI separate stack instruction sequences and we mask pointers when they are unspilled from the stack. For the deterministic variant of Swivel-SFI, we also replace conditional branches with indirect jump instructions, as described in Section 4.2.2.

To protect against sandbox poisoning attacks (§4.1.3), we flush the BTB during the springboard transition into any sandbox. Since this is a privileged operation, we implement this using a custom Linux kernel module.

Swivel-CET In the Swivel-CET code generation pass, we place `endbranch` instructions at each indirect jump target in Wasm to enable Intel® CET protection. These indirect jump targets include switch table entries and functions which may be called indirectly. We also use this pass to emit the register interlocks for the deterministic variant of Swivel-CET.

We adapt the springboard and trampoline transition functions to ensure that all uses of `jmp`, `call` and `return` conform to the requirements of Intel® CET. We furthermore use these transition functions to switch between the application and sandbox Intel® MPK domains.

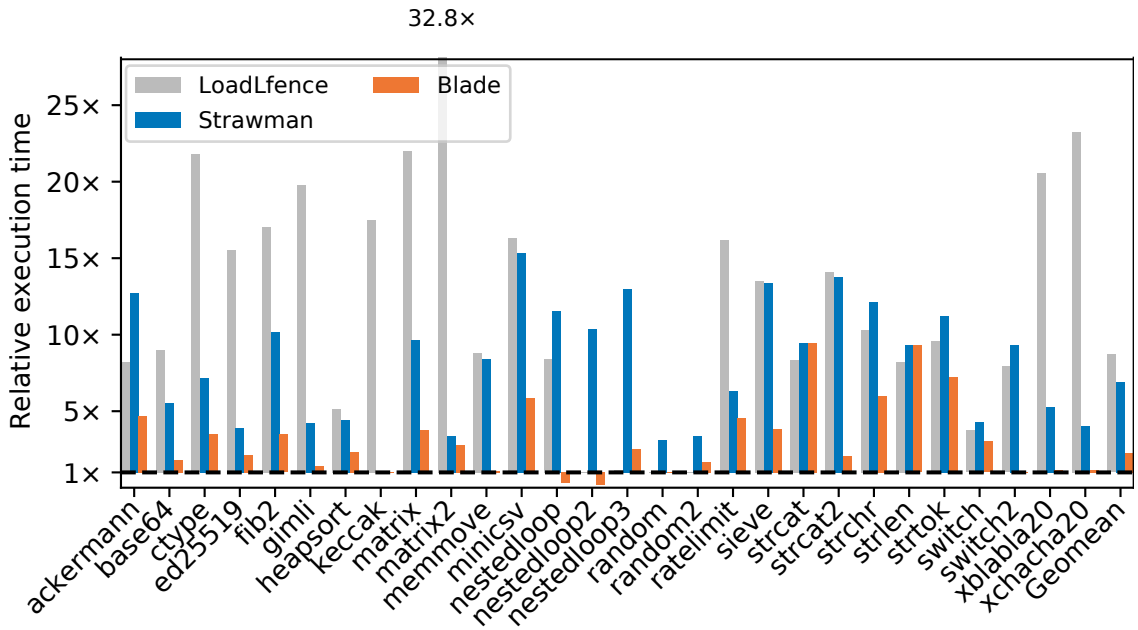
Since Intel® MPK blocks the application from accessing sandbox memory, we add primitives that briefly turn off Intel® MPK to copy memory into and out of sandboxes. We implement these primitives using the `rep` instruction prefix instead of branching code, ensuring that the primitives are not vulnerable to Spectre attacks during this window.

Finally, we add Intel® CET support to both the Rust compiler—used to compile Lucet—and to Lucet itself so that the resulting binaries are compatible with the hardware.

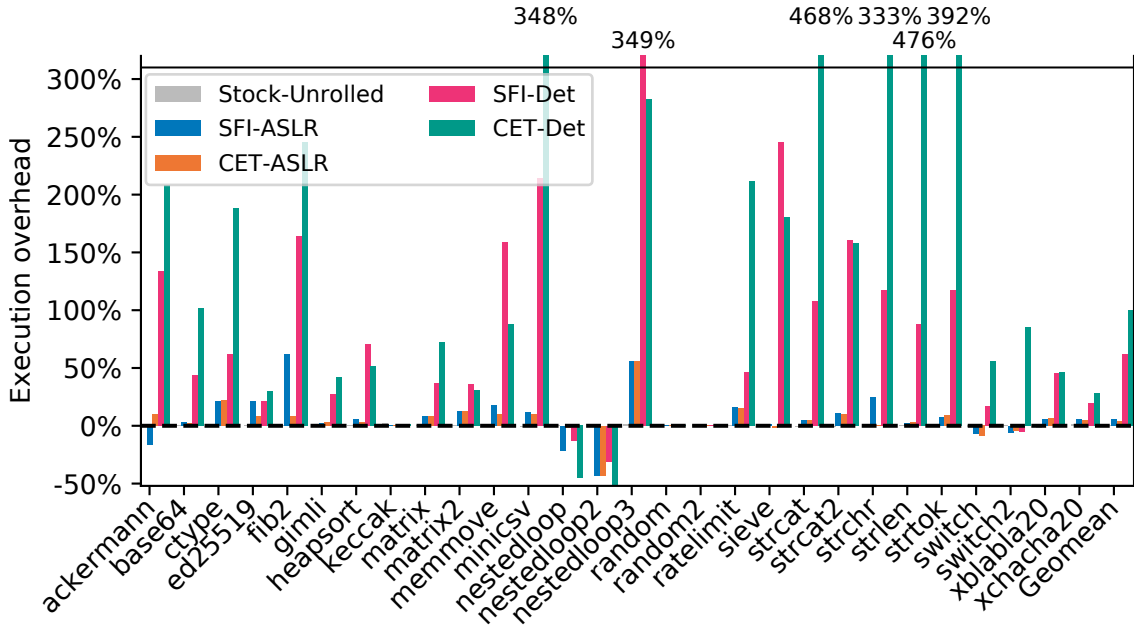
4.4 Evaluation

We evaluate Swivel by asking four questions:

- **What is the overhead of Wasm execution? (§4.4.1)** Swivel's hardening schemes make

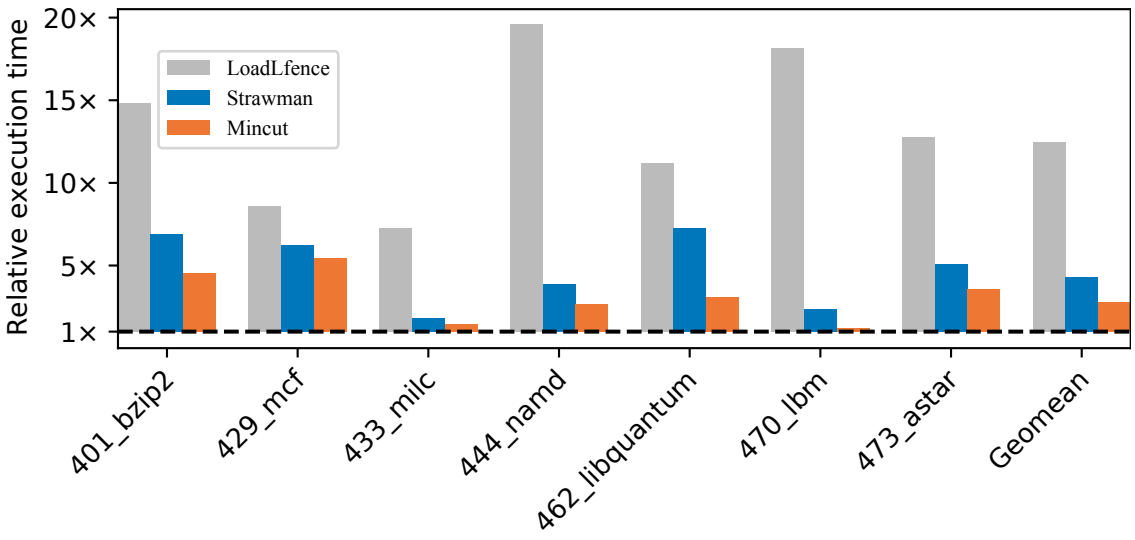


(a) Fence scheme overhead on Sightglass

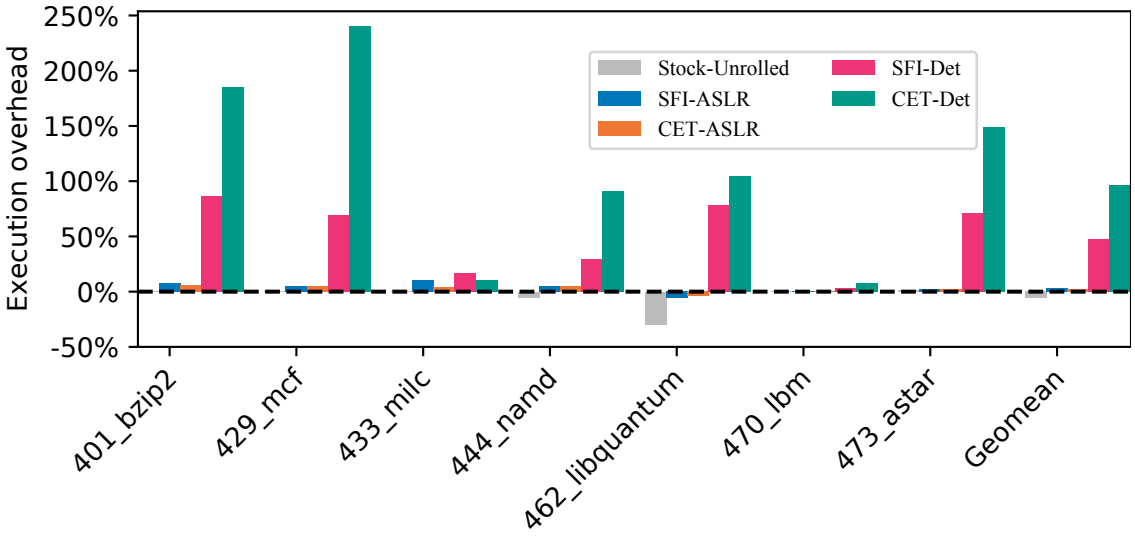


(b) Swivel scheme overhead on Sightglass

Figure 4.5: Performance overhead of Swivel on the Sightglass benchmarks. (a) On Sightglass, the baseline schemes LoadLfence, Strawman, and Mincut incur geomean overheads of 8.7×, 6.9×, and 2.4× respectively. (b) In contrast, the Swivel schemes perform much better where the ASLR versions of Swivel-SFI and Swivel-CET incur geomean overheads of 5.5% and 4.2% respectively. With deterministic sandbox poisoning mitigations, these overheads are 61.9% and 99.7%.



(a) Fence scheme overhead on SPEC 2006



(b) Swivel scheme overhead on SPEC 2006

Figure 4.6: Performance overhead of Swivel on SPEC 2006 benchmarks. (a) On SPEC 2006, the baseline schemes LoadLfence, Strawman, and Mincut incur overheads of 7.3×–19.6×, 1.8×–7.3×, and 1.2×–5.4× respectively. (b) In contrast, the Swivel schemes perform much better where the ASLR versions of Swivel-SFI and Swivel-CET incur overheads of at most 10.3% and 6.1% respectively. With deterministic sandbox poisoning mitigations, these overheads are 3.3%–86.1% and 8.0%–240.2% respectively.

changes to the code generated by the Lucet Wasm compiler. We examine the performance impact of these changes on Lucet's Sightglass benchmark suite [Byt19a] and Wasm-compatible SPEC 2006 [Hen06] benchmarks.

- **What is the overhead of transitions? (§4.4.2)** Swivel modifies the trampolines and springboards used to transition into and out of Wasm sandboxes. The changes vary across our different schemes—from adding `lfence`s, or flushing the BTB during one or both transition directions, to switching Intel® MPK domains. We measure the impact of these changes on transition costs using a microbenchmark.
- **What is the end-to-end overhead of Swivel? (§4.4.3)** We examine the impact of Wasm execution overhead and transition overhead on a webserver that runs Wasm services. We measure the impact of Swivel protections on five different Wasm workloads running on this webserver.
- **Does Swivel eliminate Spectre attacks? (§4.4.4)** We evalute the security of Swivel, i.e., whether Swivel prevents sandbox breakout and poisoning attacks, by implementing several proof-of-concept Spectre attacks.

Machine setup We run our benchmarks on a 4-core, 8-thread Tigerlake CPU software development platform (2.7GHz with a turbo boost of 4.2GHz) supporting the Intel® CET extension. The machine has 16 GB of RAM and runs 64-bit Fedora 32 with the 5.7.0 Linux kernel modified to include Intel® CET support [Int17]. Our Swivel modifications are applied to Lucet version `0.7.0-dev`, which includes Cranelift version `0.62.0`. We perform benchmarks on standard SPEC CPU 2006, and Sightglass version `0.1.0`. Our webserver macrobenchmark relies on the Rocket webserver version `0.4.4`, and we use `wrk` version `4.1.0` for testing.

4.4.1 Wasm execution overhead

We measure the impact of Swivel's Spectre mitigations on Wasm performance in Lucet using two benchmark suites:

- The Sightglass benchmark suite [Byt19a], used by the Lucet compiler, includes small functions such as cryptographic primitives (`ed25519`, `xchacha20`), mathematical functions

(`ackermann`, `sieve`), and common programming utilities (`heapsort`, `strcat`, `strtok`).

- SPEC CPU 2006 is a popular performance benchmark that includes various integer and floating point workloads from typical programs. We evaluate on only the subset of the benchmarks from SPEC 2006 that are compatible with Wasm and Lucet. This excludes programs written in Fortran, programs that rely on dynamic code rewriting, programs that require more than 4GB of memory which Wasm does not support, and programs that use exceptions, `longjmp`, or multithreading.[5] We note that Swivel does *not* introduce new incompatibilities with SPEC 2006 benchmarks; all of Swivel's schemes are compatible with the same benchmarks as stock Lucet.

Setup We compile both Sightglass and the SPEC 2006 benchmarks with our modified Lucet Wasm compiler and run them with the default settings. Sightglass repeats each test at least 10 times or for a total of 100ms (whichever occurs first) and reports the median runtime. We compare Swivel's performance overhead with respect to the performance of the same benchmarks when using the stock Lucet compiler. For increased measurement consistency on short-running benchmarks, while measuring Sightglass we pin execution to a single core and disable CPU frequency scaling.[6]

Baseline schemes In addition to our comparison against Stock Lucet, we also implement three known Spectre mitigations using `lfence`s and compare against these as a reference point. First, we implement LoadLfence, which places an `lfence` after every load, similar to Microsoft's Visual Studio compiler's "Qspectre-load" mitigation [Mic20a]. Next, we implement Strawman, a scheme which restricts code to sequential execution by placing an `lfence` at all control flow targets (at the start of all linear blocks)—this is similar to the Intel compiler's "all-fix-lfence" mitigation [Int20b]. Finally, we implement Mincut, an `lfence` insertion algorithm suggested by

[5] Some Web-focused Wasm platforms support some of these features. Indeed, previous academic work evaluates these benchmarks on Wasm [JPBG19], but non-Web Wasm platforms including Lucet do not support them.

[6] Tests were performed on June 18, 2020; see testing disclaimer 4.8.2.

Vassena et al. [VDG+21] which uses a min-cut algorithm to minimize the number of required `lfence`s. We further augment Mincut's `lfence` insertion with several of our own optimizations, including (1) only inserting a single `lfence` per linear block; and (3) unrolling loops to minimize branches, as we do for the register interlock scheme and CBP conversions (§4.2.2). Finally, to ensure that unrolling loops does not provide an unfair advantage, we also present results for `Stock-Unrolled`, which is stock Lucet with the same loop unrolling as used in Swivel's schemes.

Results We present the Wasm execution overhead of the various protection options on the Sightglass benchmarks in Figure 4.5b, and on the SPEC 2006 benchmarks in Figure 4.6b. The overheads of the ASLR versions of Swivel-SFI and Swivel-CET are small: 5.5% and 4.2% geomean overheads respectively on Sightglass, and at most 10.3% (geomean: 3.4%) and at most 6.1% (geomean: 2.6%) respectively on SPEC. The deterministic versions of Swivel introduce modest overheads: 61.9% and 99.7% on Sightglass, and 3.3%–86.1% (geomean: 47.3%) and 8.0%–240.2% (geomean: 96.3%) on SPEC. All four configurations outperform the baseline schemes by orders of magnitude: Strawman incurs geomean overheads of 6.9× and 4.3× on Sightglass and SPEC respectively, LoadLfence incurs 8.7× and 12.5× overhead respectively, while Mincut incurs 2.4× and 2.8× respectively.

Breakdown Addressing CBP poisoning (CBP-to-BTB conversion in Swivel-SFI and register interlocks in Swivel-CET) dominates the performance overhead of our deterministic implementations. We confirm this hypothesis with a microbenchmark: We measure the overheads of these techniques individually on stock Lucet (with our loop unrolling flags). We find that the average (geomean) overhead of CBP conversion is 52.9% on Sightglass and 38.5% on SPEC. The corresponding overheads for register interlocks are 93.2% and 53.4%.

Increasing loop unrolling thresholds does not significantly improve the performance of stock Lucet (e.g., the speedup of our loop unrolling on stock Lucet is 0.0% and 5.9% on Sightglass and SPEC, respectively). It does impact the performance of our deterministic Swivel variants

though (e.g., we find that it contributes to a 15%-20% speed up). This is not surprising since loop unrolling results in fewer conditional branches (and thus reduces the effect of CBP conversions and register interlocking).

To understand the outliers in Figure 4.5 and Figure 4.6, we inspect the source of the benchmarks. Some of the largest overheads in Figure 4.5 are on Sightglass' string manipulation benchmarks, e.g., `strcat` and `strlen`. These microbenchmarks have lots of *data-dependent loops*—tight loops with data-dependent conditions—that cannot be unrolled at compile-time. Since our register interlocking inserts a data dependence between the pinned heap base register and the loop condition, this prevents the CPU from speculatively executing instructions from subsequent iterations. We believe that similar data-dependent loops are largely the cause for the slowdowns on SPEC benchmarks, including `429.mcf` and `401.bzip2`. Some of the other large overheads in Sightglass (e.g., `fib2` and `nestedloop3`) are largely artifacts of the benchmarking suite: These microbenchmarks test simple constructs like loops—and CBP-to-BTB conversion naturally makes (almost empty) loops slow.

4.4.2 Sandbox transition overhead

We evaluate the overhead of context switching. As described in Section 4.2, Swivel adds an `lfence` instruction to host-sandbox transitions to mitigate sandbox breakout attacks. In addition to this: Swivel-SFI flushes the BTB during each transition; Swivel-CET, in deterministic mode, switches Intel® MPK domains during each transition; and Swivel-CET, in ASLR mode, flushes the BTB in one direction and switches Intel® MPK domain in each transition.

We measure the time required for the host application to invoke a simple no-op function call in the sandbox, as well as the time required for the sandboxed code to invoke a permitted function in the application (i.e., perform a callback). We compare the time required for Wasm code compiled by stock Lucet with the time required for code compiled with our various protection schemes. We measure the average performance overhead across 1000 such function call

Table 4.3: Time taken for transitions between the application and sandbox—for function calls into the sandbox and callback invocations from the sandbox. Swivel overheads are generally modest, with the deterministic variant of Swivel-CET in particular imposing very low overheads.

Transition Type	Function Invoke	Callback Invoke
Stock	2.14μs	0.07μs
Swivel-SFI (`lfence` + BTB flush both ways)	4.5μs	1.26μs
Swivel-CET ASLR (`lfence` + BTB flush one way + MPK)	4.08μs	0.79μs
Swivel-CET deterministic (`lfence` + MPK)	2.29μs	0.08μs

Table 4.4: Average latency (**ALat**), 99% tail latency (**TLat**), average throughput (**Tput**) in requests/second and binary files size (**Size**) for the webserver with different Wasm workloads (1k = 10^3, 1m = 10^6).

Swivel Protection	Templated HTML				XML to JSON			
	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size
Stock (unsafe)	20.8ms	42.1ms	4.81k	3.3MB	186ms	228ms	531	3.2MB
Swivel-SFI ASLR	124ms	137ms	803	3.9MB	213ms	281ms	459	3.8MB
Swivel-SFI Det	34.6ms	80.4ms	2.90k	4.2MB	279ms	322ms	350	4.1MB
Swivel-CET ASLR	111ms	123ms	898	3.4MB	197ms	252ms	498	3.3MB
Swivel-CET Det	28.7ms	66.3ms	3.50k	4.1MB	291ms	328ms	338	4.0MB

Swivel Protection	Change JPEG quality				Check SHA-256			
	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size
Stock (unsafe)	2.23s	2.93s	38.2	2.0MB	424ms	532ms	230	3.6MB
Swivel-SFI ASLR	2.31s	2.91s	36.9	2.2MB	449ms	608ms	215	4.2MB
Swivel-SFI Det	3.01s	4.13s	26.4	2.9MB	463ms	575ms	210	4.6MB
Swivel-CET ASLR	2.30s	2.88s	37.0	2.0MB	409ms	562ms	234	3.7MB
Swivel-CET Det	2.92s	3.81s	27.5	2.9MB	459ms	570ms	211	4.4MB

invocations.[7] These measurements are presented in Table 4.3.

First, we briefly note that function calls in stock Lucet take much longer than callbacks. This is because the Lucet runtime has not fully optimized the function call transition, as these are relatively rare compared to callback transitions, which occur during every syscall.

In general, Swivel's overheads are modest, with the deterministic variant of Swivel-CET in particular imposing very low overheads. Flushing the BTB does increase transition costs, but the overall effect of this increase depends on how frequently transitions occur between the application and sandbox. In addition, flushing the BTB affects not only transition performance but

[7]Tests were performed on June 18, 2020; see testing disclaimer 4.8.2.

Table 4.5: Average latency (**ALat**), 99% tail latency (**TLat**), average throughput (**Tput**) in requests/second and binary files size (**Size**) for the webserver for a long-running, compute-heavy Wasm workload (1k = 10^3, 1m = 10^6).

Swivel Protection	Image classification			
	ALat	**TLat**	**Tput**	**Size**
Stock (unsafe)	9.67s	13.1s	2.05	34.2MB
Swivel-SFI ASLR	9.78s	13.9s	2.03	34.3MB
Swivel-SFI Det	17.7s	28.3s	1.11	34.7MB
Swivel-CET ASLR	9.82s	12.8s	2.02	34.2MB
Swivel-CET Det	15.7s	24.9s	1.26	34.7MB

also the performance of both the host application and sandboxed code. Fully understanding these overheads requires that we evaluate the overall performance impact on real world applications, which we do next.

4.4.3 Application overhead

We now evaluate Swivel's end-to-end performance impact on a webserver which uses Wasm to host isolated web services.

Setup For this benchmark, we use the Rocket webserver [roc20], which can host web services written as Wasm modules. Rocket operates very similarly to webservers used in previous academic papers exploring Wasm modules [HR19, SP20] as well as frameworks used by CDNs such as Fastly. We measure the webserver's performance while hosting five different web services with varying CPU and IO profiles. These services perform the following five tasks respectively: (1) expanding an HTML template; (2) converting XML input to JSON output; (3) re-encoding a JPEG image to change image quality; (4) computing the SHA-256 hash of a given input; and (5) performing image classification using inference on a pretrained neural network. We measure the overall performance of the webserver by tracking the average latency, 99% tail-latency, and throughput for each of the five web services. We also measure the size of the Wasm binaries

produced.[8]

Results Tables 4.4 and 4.5 show results of the webserver measurements. From the table, we see any of sys's schemes only reduce geomean throughput (across all workloads) between 28.4% and 33.7%. Swivel also modestly increases Wasm binary sizes, particularly with its deterministic schemes, due to additional instructions added for separate stack, CBP-to-BTB, and interlock mechanisms.

For long-running, compute-heavy Wasm workloads such as JPEG re-encoding and image classification, Swivel's performance overhead is dominated by Wasm execution overhead measured in Section 4.4.1. Thus, on these workloads the ASLR versions of Swivel perform much better than the deterministic versions, as their Wasm execution overhead is lower. On the other hand, for short-running workloads such as templated HTML, we observe that the deterministic schemes outperform the ASLR schemes. This is because Swivel's ASLR implementation must remap and `memcpy` the sandbox code pages during sandbox creation, effectively adding a fixed overhead to each request. For short-running requests, this fixed per-request cost dominates overall overhead. In contrast, Stock Lucet and Swivel's deterministic schemes take advantage of shared code pages in memory to create sandboxes more rapidly, incurring lower overhead on short-running requests.

4.4.4 Security evaluation

To evaluate the security of Swivel, we implement several Spectre attacks in Wasm and compile this attack code with both stock Lucet and Swivel. We find that stock Lucet produces code that is vulnerable to Spectre, i.e., our proof of concept attacks (POCs) can be used to carry out both breakout and poisoning attacks, and that Swivel mitigates these attacks.

[8]Tests were performed on June 18, 2020; see testing disclaimer 4.8.2.

Attack assumptions Our attacks extend Google's Safeside [Goo20] suite and, like the Safeside POCs, rely on three low-level instructions: The `rdtsc` instruction to measure execution time, the `clflush` instruction to evict a particular cache line, and the `mfence` instruction to wait for pending memory operations to complete. While these instructions are not exposed to Wasm code by default, we expose these instructions to simplify our POCs. Additionally, for cross Wasm module attacks, we manually specify the locations where Wasm modules are loaded to simplify the task of finding partial address collisions in the branch predictor.

Our simplifications are not fundamental and can be removed in an end-to-end attack. Previous work, for example, showed how to construct precise timers [SMGM17, FGBR18], and how to control cache contents [VKM19] in environments like JavaScript where these instructions are not directly exposed. The effects of the `mfence` instruction can be achieved by executing `nop` instructions until all memory operations are drained. And, in the style of heap and JIT spraying attacks [Sin10], we can increase the likelihood of partial address collision by deploying hundreds to thousands of modules on the FaaS platform.

POC 1: Sandbox breakout via in-place Spectre-PHT Our first POC adopts the original Spectre-PHT bounds-check bypass attack [KHF+19] to Wasm. As mentioned in Section 4.1.3, in Wasm, indirect function calls are expressed as indices into a function table. Hence, the code emitted for the `call_indirect` instruction performs a bounds check, to ensure that the function index is within the bounds of the table, before performing the lookup and call. By inducing a misprediction on this check, our POC can read beyond the function table boundary and treat the read value as a function pointer. This effectively allows us to jump to any code location and speculatively bypass Wasm's CFI (and thus isolation). We demonstrate this by jumping to a host function that returns bytes of a secret array.

POC 2: Sandbox breakout and poisoning via out-of-place Spectre-BTB Our second POC adopts the out-of-place Spectre-BTB attack of Canella et al. [CVBS+19] to Wasm. Specifically,

we mistrain an indirect jump in a victim or attacker-controlled module by training a congruent indirect jump instruction in another attacker-controlled module. We train the jump to land on a gadget of our choice. To demonstrate the feasibility of a sandbox poisoning attack, we target a double-fetch leak gadget. To demonstrate a sandbox breakout attack, we jump in the middle of a basic block to a memory load, skipping Wasm's heap bounds checks.

POC 3: Poisoning via out-of-place Spectre-RSB Our third POC compiles the Spectre-RSB attack from the Google Safeside project [Goo20] to Wasm. This attack underflows the RSB to redirect speculative control flow. We use this RSB underflow behavior to speculatively "return" to a gadget that leaks module secrets. We run this attack entirely within a single Wasm module. However, on a FaaS platform this attack can be used across modules when the FaaS runtime interleaves the execution of multiple modules, similar to the Safeside cross-process Spectre-RSB attack.

Results We developed our POCs on a Skylake machine (Xeon Platinum 8160) and then tested them on both this machine and the Tiger Lake Intel® CET development platform we used for our performance evaluation. We found that stock Lucet on the Skylake machine was vulnerable to all three POCs while Swivel-SFI, both the ASLR and deterministic versions, were not vulnerable. On the Tiger Lake machine, we found that stock Lucet was vulnerable to POC 3 while Swivel-SFI and Swivel-CET, both the ASLR and deterministic versions, were not. Although the Tiger Lake CPU is documented to be vulnerable to all three Spectre variants [Int20c], we did not successfully reproduce POC 1 and POC 2 on this machine. Getting these attacks to work may require reverse engineering the branch predictors used on these new CPUs. We thus leave the extensions of our POCs to this microarchitecture to future work.

4.5 Limitations and discussion

In this section, we cover some of the current limitations of Swivel, briefly mention alternate design points, and address the generality of our solutions.

4.5.1 Limitations of Swivel

We discuss some limitations of Swivel, both in general and for our implementation in particular.

Implementation limitations For this paper, we have simplified some of the implementation details for Swivel-CET to reduce the engineering burden of modifying multiple compiler toolchains and standard libraries while still providing accurate performance evaluations. First, we do not ensure that interlock labels are unique to each linear block, but rather reuse interlock labels; while unique labels are critical for security, previous works have extensively demonstrated the feasibility of assigning unique labels [BCN+17]. Our goal was to measure the performance of the instruction sequences for interlock assignment (64-bit conditional moves) and checking (64-bit conditional checks).

Next, when disabling Intel® MPK protections in Swivel-CET (§4.2.3) in the host calls, we must avoid using indirect branches; while we follow this principle for hostcalls we expose (e.g., when marshaling data for web server requests), we do not modify existing standard library hostcalls. These additional modifications, while straightforward, would require significant engineering effort in modifying the standard library (libc) used by Wasm. Finally, we did not implement the required guard pages in the lower 4GB of memory in our prototype of deterministic Swivel-CET. Prior work [DZL15] has shown how to reserve the bottom 4GB of memory—and that this does not impact performance.

Secretless host Swivel assumes that the host (or runtime) doesn't contain secret information. This assumption is sensible for some applications: in the CDN use case, the CDN part of the process is lightweight and exists only to coordinate with the sandboxes. But not all. As a counter-example, the Firefox web browser currently uses Wasm to sandbox third-party libraries written in C/C++ [NDG+20, Fro20]. We could use Swivel to ensure that Firefox is secure from Spectre attacks conducted by a compromised third-party libraries. To protect secrets in the host (Firefox), we could either place the secrets into a separate Wasm sandbox, or apply one of our proposed CBP protections to the host (e.g., CBP-to-BTB or interlocks).

Hyperthreading The only scheme in Swivel that supports hyperthreading is the deterministic Swivel-CET. Alternately, instead of disabling hyperthreading, Intel suggests relying on single-threaded indirect branch predictors (STIBP) to prevent a co-resident thread from influencing indirect branch predictions [Int18a]. STIBP could allow any Swivel scheme to be used securely with hyperthreading.

4.5.2 Other leakages and transient attacks

Swivel-CET allows victim code to run with poisoned predictors but prevents exfiltration via the data cache. This, unfortunately, means that attackers may still be able to leak victim data through other microarchitectural channels (e.g., port contention or the instruction cache [KHF+19]). Swivel-SFI does not have this limitation; we can borrow techniques from Swivel-SFI to eliminate such leaks (e.g., flushing the BTB).

The CPU's memory subsystem may also introduce other transient execution attacks. Spectre-STL [Hor18] can leak stale data (which may belong to another security domain) before a preceding store could overwrite this data due to speculative dependency checking of the load and the preceding stores. Swivel does not address Spectre-STL. However, Spectre-STL can been mitigated through speculative store bypass disable (SSBD) [Int18c], which imposes a small

performance overhead (less than 5% on most benchmarks [Lar18]) and is already enabled by default on most systems.

Meltdown [LSG+18] could leak privileged kernel memory from userspace. Variants of the Meltdown attack, e.g., microarchitectural data sampling (MDS), can be used to leak stale data from several microarchitectural elements [Int19, CGG+19, SLM+19, MLSS20, vSMÖ+19]. Load Value Injection (LVI) exploits the same microarchitectural features as Meltdown to inject data into the microarchitectural state [VBMS+20]. More recent Intel CPUs (e.g., Tiger Lake) are designed to be resilient against this class of attacks [Int20c], and we believe that these attacks can efficiently be mitigated in hardware. For this reason, recent research into secure speculation and architectural defense against transient execution attacks are mostly focused on the Spectre class of issues [GKM+20, Car18, Tur18, WCG+19].

For legacy systems, users should apply the latest microcode and software patches to mitigate Meltdown and variants of MDS [ker20, Int20c]. For variants of MDS that abuse hyperthreading on legacy systems, Intel suggests safe scheduling of sibling CPU threads [Int19]. Since Wasm restricts what instructions are allowed in a Wasm module, this makes some MDS attacks more challenging to execute. For instance, Wasm modules cannot use Intel® TSX transactions or access non-canonical or kernel addresses that are inherent to some of the MDS variants [ker19]. LVI requires fine-grain control over inducing faults or microcode assists, which is not available at the Wasm level. Some legacy systems may still be vulnerable to LVI; however, the feasibility of LVI attacks outside the Intel SGX environment is an open research question [VBMS+20].

4.5.3 Alternate design points for Swivel

We next describe alternate designs for Swivel and discuss the trade-offs of our design choices.

CBP-to-BTB conversion in Swivel-CET Since register interlocking is more expensive than CBP-to-BTB conversion, a reader may wonder whether the deterministic Swivel-CET could more efficiently protect against sandbox poisoning using CBP-to-BTB conversion. Unfortunately, since Swivel-CET does not flush the BTB in both directions, CBP-to-BTB conversion is not sufficient to fully mitigate sandbox poisoning. In addition to CBP-to-BTB conversion, Swivel-CET would also need to use interlocking (without additional performance gain) or flush the BTB both ways. In the latter case, we might as well use Swivel-SFI, as the main advantage of using Intel® CET in Swivel is to avoid flushing the BTB.

Interlocking in Swivel-SFI Likewise, one may wonder about the benefits of using interlock in Swivel-SFI. Unfortunately, for interlock to be useful, it requires hardware support. First, we require Intel® MPK to ensure that a sandbox can't confuse the host into accessing (and then leaking) another sandbox's data. Second, we require the Intel® CET `endbranch` instruction to ensure that the sandbox cannot use BTB entries leftover from the host.

Partitioning shared resources A different approach to addressing Spectre would be to partition differ hardware structures to ensure isolation. For example, for the CBP, one approach would be to exploit the indexing mechanism of branch predictors such that each sandbox uses an isolated portion of the CBP. Unfortunately doing this on existing CPUs is hard: superscalar CPUs use complex predictors with multiple indexing functions, and without knowledge of the underlying microarchitecture, we were unable to experimentally find a way to partition the CBP.

Alternately, we could mitigate host poisoning—or even sandbox poisoning—attacks by partitioning CPU cores, and preventing an attacker from running on the same core as their victim. This approach protects against sandbox poisoning and host poisoning, since branch predictors are per-physical-core. We tried this. Specifically, we implemented this mitigation in the host poisoning context and measured its performance. Unfortunately, requiring a core transition during every springboard and trampoline is detrimental to performance, and this scheme was not

competitive with our chosen implementation.

4.5.4 Generalizing Swivel

Swivel's techniques are not specific to the Lucet compiler or runtime. Our techniques can be applied to other Wasm compilers and runtimes, including the just-in-time Wasm compilers used in the Chrome and Firefox browsers.

Our techniques can also be adopted to other software-based fault isolation (SFI) compilers [Tan17]. Adopting Swivel to the Native Client (NaCl) compiler [YSD+09, SMB+10], for instance, only requires only a handful of changes. For example, we wouldn't even need to add linear blocks: NaCl relies on instruction bundles—32-byte aligned blocks of instructions—which are more restrictive than our linear blocks (and satisfy our linear block invariants).

More generally, Swivel can be adopted to other sandboxed languages and runtimes. JavaScript just-in-time compilers are a particularly good fit. Though JavaScript JITs are more complex than Wasm compilers, they share a similar security model (e.g., JavaScript in the browser is untrusted) and, in some cases, even share a common compilation pipeline. For example, Cranelift—the backend used by Lucet and Swivel—was designed to replace Firefox's JavaScript and Wasm backend implementations [Goh18], and thus could transparently benefit from our mitigations. Beyond Cranelift, we think that adopting our linear blocks and code page ASLR is relatively simple (e.g., compared to redesigning the browser to deal with Spectre) and could make JavaScript Spectre attacks significantly more difficult.

4.5.5 Implementation bugs in Wasm

Lehmann et al. [LKP20] showed that some Wasm compilers and runtimes, like prior SFI toolchains [Tan17], contain implementation bugs.[9] For example, they showed that some Wasm

[9]They also show that C memory safety bugs are still present within the Wasm sandbox—this class of bugs is orthogonal and cannot alone be used to to bypass Wasm's isolation guarantees.

runtimes fail to properly separate the stack and heap. Though they did not identify such bugs in Lucet, these classes of bugs are inevitable—and, while identifying such bugs is important, this class of bugs is orthogonal and well-understood in the SFI literature (and addressed, for example, by VeriWasm [JTA+21]). We focus on addressing Spectre attacks, which can fundamentally undermine the guarantees of even bug-free Wasm toolchains.

4.5.6 Future work

Swivel's schemes can benefit from extensions to compiler toolchains as well as hardware to both simplify its mitigations and improve performance. We briefly discuss some possible extensions and their benefits below.

Compiler toolchain extensions

We describe two performance optimizations for the Swivel-CET deterministic scheme, and a way to improve the security of Swivel's ASLR schemes.

Data dependent loops As discussed in Section 4.4.1, the Swivel-CET deterministic scheme imposes the greatest overheads in programs with data-dependent loops—e.g., programs that iterate over strings or linked lists (which loop until they find a null element). Swivel effectively serializes iterations of such data-dependent loops. We expect that many other Spectre mitigation (see Section 4.6), like speculative taint tracking [YYK+19], would similarly slow down such programs.

One way to speed up such code is to replace the data-dependent loops with a code sequence that first counts the expected number of iterations (N), executes an `lfence`, and then runs the original loop body for N iterations. This would introduce only a single stall in the loop and eliminate the serialization between loop iterations.

Compiler secret tracking Swivel currently assumes all locations in memory contain potentially secret data. However, several works (e.g., [WCG+19]) have proposed tracking secrets in compiler passes. This information can be used to optimize the Swivel-CET deterministic scheme. In particular, any public memory access can be hoisted above the register interlock to allow the memory to be accessed (and "leaked") speculatively.

Software diversity Swivel's ASLR variants randomize code pages. We could additionally use software diversity to increase the entropy of our probabilistic schemes [FSA97, HNTC+12]. Software diversity techniques (e.g., `nop` insertion) are cheap [HNL+13], and since they do not affect the behavior of branches, they can be used to specifically mitigate out-of-place Spectre-BTB and Spectre-PHT attacks.

Hardware extensions

Hardware extensions can make Swivel faster and simpler.

CBP flushing Swivel-SFI schemes rely on ASLR or CBP-to-BTB conversion to protect the CBP. However, hardware support for CBP flushing could significantly speed up Swivel. Alternatively, hardware support for tagging predictor state (e.g., host code and sandbox code) would allow Swivel to isolate the CBP without flushing.

Dedicated interlock instructions The register interlocking used in deterministic Swivel-CET requires several machine instructions in each linear block in order to assign and check labels. Dedicated hardware support for these operations could reduce code bloat.

Explicit BTB prediction range registers The Swivel-CET deterministic scheme allocates unique 64-bit labels to each linear block, which do not overlap across sandbox instances. We could simplify and speed up this scheme with a hardware extension that can be used to limit

BTB predictions to a range of addresses. With such an extension, Swivel could set the prediction range during each transition into the sandbox (to the sandbox region) and ensure that the BTB could only predict targets inside the sandbox code pages. This would eliminate out-of-place BTB attacks—and, with linear blocks, it would eliminate breakout attacks in Wasm. Finally, this would reduce code size: it would allow us to to reduce block labels to, for example, 16 bits (since we only need labels to be unique within the sandbox).

4.6 Related work

We give an overview of related work on mitigating Spectre attacks by discussing microarchitectural proposals, software-based approaches for eliminating Spectre gadgets, and previous approaches based on CFI or Intel® MPK.

Thwarting covert channels Several works [KLA+18, KKS+19, BBZ+19, YCS+18, SQ19] propose making microarchitectural changes to block, isolate, or remove the covert channels used to transfer transient secrets to architectural states. For example, *SafeSpec* [KKS+19] proposes a speculation-aware memory subsystem which ensures that microarchitectural changes to the cache are not committed until predictions are validated. Similarly, *CleanupSpec* [SQ19] proposes an undo logic for the cache state. Although these approaches remove the attacker's data leakage channel, they do not address the root cause of Spectre vulnerabilities. In contrast, Swivel works with no hardware changes.

Safe speculation Intel has introduced hardware support to mitigate Spectre-BTB across separate address spaces [Int18a, Int20c]. Specifically, the Indirect Branch Predictor Barrier (IBPB) allows the BTB to be cleared across context switches, while Single Thread Indirect Branch Predictors (STIBP) ensure that one thread's BTB entries will not be affected by the sibling hyperthread. These mitigations can be used by the OS as a coarse-grained mechanism for safe speculation, but only

apply to Spectre-BTB and have not been widely adopted due to performance overhead [KSK+20].

Other works propose microarchitectural changes to allow the software to control speculation for security-critical operations [TVT19, WNL+19] or certain memory pages [SLC+20, LZH+19]. Separately, STT [YYK+19] proposes speculative taint tracking within the microarchitecture. However, unlike Swivel, these approaches require significant hardware changes and do not offer a way to safely run code on existing CPUs.

Eliminating Spectre gadgets Another way to mitigate Spectre attacks is by inserting a barrier instruction (e.g., `lfence`), which blocks speculative execution [Mic20a, Int18b, AMD18]. However, as we evaluated in Section 4.4.1, insertion of `lfence` has a performance impact on the entire CPU pipeline and undercuts the performance benefit of out-of-order and speculative execution. In contrast, Swivel makes little to no use of `lfence`.

An optimized approach is to replace control flow instructions with alternate code sequences that are safe to execute speculatively. For instance, speculative load hardening (SLH) replaces conditional bounds checks with an arithmetized form to avoid Spectre-PHT [Car18]. Indeed, Swivel uses SLH to protect the bounds checks for indirect call tables and switch tables (§4.2.1). Alternatively, Oleksenko et al. [OTR+18] propose inserting artificial data dependencies between secret operations and pipeline serialization instructions. Finally, the *retpoline* technique [Tur18] replaces indirect branches with a specific code sequence using the `ret` instruction to avoid Spectre-BTB. To reduce the overhead of such code transformations, researchers have proposed several techniques to automatically locate Spectre gadgets [WCG+19, GKM+20, CDG+20] and apply mitigations to risky blocks of code. However, these techniques have to handle potential false positives or negatives; in contrast, Swivel focuses on defending against all possible Spectre attacks from untrusted code by applying compile-time mitigations.

Speculative CFI *SpecCFI* [KSK+20] has proposed hardware support for speculative and fine-grained control-flow integrity (CFI), which can be used to protect against attacks on indirect

branches. In comparison, Swivel-CET uses Intel® CET, which only supports coarse-grained CFI with speculative guarantees [SGS19]. *Venkman* [SZOC19] uses a technique similar to Swivel's linear blocks to ensure that indirect branches always reach a barrier instruction (e.g., `lfence`) by applying alignment to bundles similar to classical software fault isolation [WLAG93]. In contrast, Swivel is a fence-free approach that preserves the performance benefits of speculative execution.

Intra-process isolation using Intel® MPK Jenkins et al. [JAS+20] propose to provide intra-process Spectre protection using Intel® MPK. They use Intel® MPK to create separate isolation domains and use the relationship between the code and secret data to limit speculative accesses. However, since Intel® MPK only provides 16 domains, relying fully on Intel® MPK to isolate many sandbox instances is infeasible for the CDN Wasm use case we consider.

4.7 Conclusion

This work proposes a framework, Swivel, which provides strong in-memory isolation for Wasm modules by protecting against Spectre attacks. We describe two Swivel designs: Swivel-SFI, a software-only approach which provides mitigations compatible with existing CPUs, and Swivel-CET, which leverages Intel® CET and Intel® MPK. Our evaluation shows that versions of Swivel using ASLR incur low performance overhead (at most 10.3% on compatible SPEC 2006 benchmarks), demonstrating that Swivel can provide strong security guarantees for Wasm modules while maintaining the performance benefits of in-process sandboxing.

4.8 Appendix

4.8.1 Brief introduction to CET and MPK

CET Intel® CET is an instruction set architecture extension that helps prevent Return-Oriented Programming and Call/Jmp-Oriented Programming via use of a *shadow stack*, and *indirect branch tracking* (IBT). The shadow stack is a hardware-maintained stack used exclusively to check the integrity of return addresses on the program stack. To ensure the shadow stack cannot be tampered with, it is inaccessible via standard load and store instructions. The IBT allows the enforcement of coarse-grained control flow integrity (CFI) [ABEL09] via a branch termination instruction, `endbranch`. Binaries that wish to use IBT place the `endbranch` at all valid indirect jump targets. If an indirect jump instruction lands on any other instruction, the CPU reports a control-flow protection fault. Additionally, the IBT also supports a *legacy bitmap*, which allows programs to demarcate which code pages have IBT checking enabled.

Importantly, Intel® CET guarantees that any shadow stack mismatches observed during speculative execution of `return` instruction immediately halts further speculative execution. Similarly, any indirect jump during speculative execution from an IBT enabled code page to a page with IBT disabled also halts speculation.

MPK Intel® MPK uses four bits in each page-table entry to assign one of sixteen "keys" to any given memory page, allowing for 16 different memory domains. User mode instructions `wrpkru` and `rdpkru` allow setting read and write permissions for each of these domains on a per-thread basis. Intel® MPK thus allows a process to partition its memory and selectively enable/disable read and write access to any of regions without invoking the kernel functions or switching page tables.

Importantly, `wrpkru` does not execute speculatively - memory accesses affected by the PKRU register will not execute (even speculatively) until all prior executions of `wrpkru` have

completed execution and updated the PKRU register and are also resistant to Meltdown style attacks [Int18b].

4.8.2 Testing Disclaimer

Since we use a software development platform provided by Intel, we include the following disclaimer from Intel:

> Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark (in this paper SPEC CPU 2006 and Sightglass), are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation.

4.9 Chapter acknowledgments

Chapter 4, in full, is a reprint of the material as it appears in the USENIX Security Symposium 2021. Narayan, Shravan; Disselkoen, Craig; Moghimi, Daniel; Cauligi, Sunjay; Johnson, Evan; Gang, Zhao; Vahldiek-Oberwagner, Anjo; Sahita, Ravi; Shacham, Hovav; Tullsen, Dean; Stefan, Deian. USENIX, 2021. The dissertation author was the primary investigator and author of this material.

Chapter 5

Zero-Cost: Fast transitions to sandboxed code

Memory safety bugs are the single largest source of critical vulnerabilities in modern software. Recent studies found that roughly 70% of all critical vulnerabilities were caused by memory safety bugs [Mil19, Chr20] and that malicious attackers are exploiting these bugs before they can be patched [Goo21a, MSS20]. Software sandboxing — or software-based fault isolation (SFI) — promises to reduce the impact of such memory safety bugs [Tan17, WLAG93]. SFI toolkits like Native Client (NaCl) [YSD+09] and WebAssembly (Wasm) allow developers to restrict untrusted components to their own *sandboxed* regions of memory thereby isolating the damage that can be caused by bugs in these components. Mozilla, for example, uses Wasm to sandbox third-party C libraries in Firefox [NDG+20, Fro20]; SFI allows the browser to use libraries like libgraphite (font rendering), libexpat (XML parsing), libsoundtouch (audio processing), and hunspell (spell checking) without risking whole-browser compromise due to library vulnerabilities. Others have used SFI to isolate code in OS kernels [EAV+06, CCM+09, HBG+09, SESS96], databases [FC08, For05, WLAG93], browsers [LSW95, YSD+09, HRS+17], language runtimes [STM10, NT14], and serverless clouds [McM20, Var18, GMP+20].

SFI toolkits enforce memory isolation by placing untrusted code into a sandboxed environment within which every memory access is dynamically checked to be safe. For example, NaCl and Wasm toolkits (e.g., Lucet [Byt20a] and WAMR [Byt20b]) instrument memory accesses to ensure they are within the sandbox region and add runtime checks to ensure that all control flow is confined to the sandboxed paths with instrumented memory accesses. There is a large body of work that ensures the runtime checks are *fast* on different architectures, e.g., x86 [MM06, YSD+09, FC08, PG11], x86-64 [SMB+10], SPARC® [ATLLW96], and ARM® [ZLDSR11, ZWCW14, SMB+10], as otherwise they incur unacceptable overheads on the code executing in the sandbox. Similarly, there is a considerable literature that establishes that the checks are *correct* [MTT+12, KSA14, JTA+21, BJL18, BBD+19], as even a single missing check can let the attacker escape the sandbox.

However, the security and overhead of software sandboxing also crucially depends on the correctness and cost of context switching — the *trampolines* and *springboards* used to transition into and out of sandboxes. Almost all SFI systems, from [WLAG93]'s original SFI implementation to recent Wasm SFI toolkits [Byt20a, Byt20b], use *heavyweight transitions* for context switching.[1] These transitions (1) switch protection domains by tying into the underlying memory isolation mechanism (e.g., by setting segment registers [YSD+09], memory protection keys [VOED+19, HGJ+19], or sandbox context registers [Byt20a, Byt20b]), and (2) save, scrub, and restore machine state (e.g. the stack pointer, program counter, and callee-save registers) across the boundary. This code is complicated and hard to get right, as it has to account for the particular quirks of different architectures and operating system platforms [AVBOP20]. Consequently, bugs in transition code have led to vulnerabilities in both NaCl and Wasm — from sandbox breakouts [Nac11a, Nac11b], to information leaks [Nac10, Nac12], and application state corruption [Ryd20]. Furthermore, in applications with high application-sandbox context switching rates, the cost of transitions dominates the overall sandboxing overhead. For example,

[1] The one exception is WasmBoxC [Zak20], discussed in Section 5.6.

heavyweight transitions prohibitively slowed down font rendering in Firefox, preventing Mozilla from shipping a sandboxed libgraphite [NDG+20].

In this paper, we develop the principles and pragmatics needed to implement SFI systems with near-zero-cost transitions, thereby realizing the three-decade-old vision of reducing the cost of SFI context switches to (almost) that of a function call. We do this via four contributions:

1. Zero-cost conditions for isolation (§5.1). Heavyweight transitions provide security by wrapping cross-domain calls and returns to ensure that sandboxed code cannot, for example, read secret registers or tamper with the stack pointer. While this wrapping is necessary when sandboxing arbitrary code, our insight is these wrappers can be made redundant when the code enjoys additional structure, not dissimilar to the additional structure typically imposed by most SFI systems to ensure memory isolation. For example, NaCl uses *coarse-grained* control-flow integrity (CFI) to restrict the sandbox's control flow to its own code region [Tan17, YSD+09, HRS+17].

We concretize this insight via our second contribution, a precise definition of *zero-cost conditions* that guarantee that sandboxed code can safely use zero-cost transitions. In particular, we show that transitions can be eliminated when sandboxed code follows a *type-directed* CFI discipline, has well-bracketed control flow, enforces local state (stack and register) encapsulation, and ensures registers and stack slots are initialized before use. Our notion of zero-cost conditions is inspired, in part, by techniques that use type- and memory-safe languages to isolate code via language-level enforcement of well-bracketed control flow and local state encapsulation [MMT10, MWC10, GSS+15, MSL+08, HL07, MCG+99]. However, instead of requiring developers to rewrite millions of lines of code in high-level languages [Tan17], our zero-cost conditions distill the semantic guarantees provided by high-level languages to allow retrofitting zero-cost transitions in the SFI setting. In other part, our work is inspired by [BJL18], who define a defensive semantics for SFI that captures a notion of sandboxing via simple function calls with a stack shared between the sandbox and host application. Our work builds on this

work by addressing two shortcomings: First, their definition does not account for confidentiality of application data, and implementations based on their system would thus need heavyweight transitions to prevent such attacks. Second, their defensive semantics makes fundamental use of guard zones, which limits the flexibility of the framework. Our definitions of zero-cost transitions have no such limitations and fully realize their goal of defining flexible, secure SFI with zero-cost transitions between application and sandbox.

2. Instantiating the zero-cost model (§5.2). We demonstrate the retrofitting of zero-cost transitions via our third contribution, an instantiation of our zero-cost model to two SFI systems: Wasm and `SegmentZero32`. Previous work has shown how Wasm can provide SFI by compiling untrusted C/C++ libraries to native code using Wasm as an IR [BLP20, Zak20, NGL+19, NDG+20]. We show that Wasm satisfies our zero-cost conditions, and replace the heavyweight transitions used by the industrial Lucet Wasm SFI toolkit with zero-cost transitions. Wasm imposes more structure than required by our zero-cost conditions (and Wasm compilers are still relatively new and slow [JPBG19]), so, in order to compare the overhead of our zero-cost model to the still fastest SFI implementation — NaCl [YSD+09] — we design a new prototype SFI system (`SegmentZero32`) that: (1) enforces our zero-cost conditions through LLVM-level transformations, and (2) enforces memory isolation in hardware, using 32-bit x86 segmentation.[2]

3. Verifying security at the binary level (§5.3). Our fourth contribution is a *static verifier*, VeriZero, that checks whether a potentially malicious binary produced by the Lucet toolkit satisfies our zero-cost conditions. This removes the need to trust the Lucet compiler when, for example, compiling third-party Firefox libraries [NDG+20] or untrusted tenant code running on Fastly's serverless cloud [McM20]. To prove the soundness of VeriZero, we develop a logical relation that captures when a compiled Wasm function is well-behaved with respect to our zero-cost conditions and use it to prove that the checks of VeriZero guarantee that the zero-cost conditions are met.

[2]While the prevalence of 32-bit x86 systems is declining, it nevertheless still constitutes over 20% of the Firefox web browser's user base (over forty million users) [Moz21]; `SegmentZero32` would allow for high performance library sandboxing on these machines.

We implement VeriZero by extending VeriWasm [JTA+21] and show that in just a few seconds, it can (1) verify sandboxed libraries that ship (or are in the process of being shipped) with Firefox, Wasm-compiled SPEC CPU® 2006 benchmarks, and 100,000 programs randomly generated by Csmith [YCER11], and (2) catch previous NaCl and Wasm vulnerabilities (§5.4.4). VeriZero is being integrated into the Lucet industrial Wasm compiler [Joh21].

4. Implementation and evaluation (§5.4). Our last contribution is an implementation of our zero-cost sandboxing toolkits, and an evaluation of how they improve the performance of a transition micro-benchmark and two macro-benchmarks — image decoding (libjpeg) and font rendering (libgraphite) in Firefox. First, we demonstrate the potential performance of a purpose-built zero-cost SFI system, by evaluating `SegmentZero32` on SPEC CPU® 2006 and our macro-benchmarks. We find that `SegmentZero32` imposes at most 25% overhead on SPEC CPU® 2006 (nc), and at most 24% on image decoding and 22.5% on font rendering. These overheads are lower than the state-of-art NaCl SFI system. On the macro-benchmarks, `SegmentZero32` even outperforms an idealized SFI system that enforces memory isolation for free but requires heavyweight transitions. Second, we find that zero-cost transitions speed up Wasm-sandboxed image decoding by (up to) 29.7% and font rendering by 10%. The speedup resulting from our zero-cost transitions allowed Mozilla to ship the Wasm-sandboxed libgraphite library in production.

Open source and data. Our code and data will be made available under an open source license.

5.1 Overview

In this section we describe the role of transitions in making SFI secure, give an overview of existing heavyweight transitions, and introduce our zero-cost model, which makes it possible for SFI systems to replace heavyweight transitions with simple function calls.

5.1.1 The Need for Secure Transitions

As an example, consider sandboxing an untrusted font rendering library (e.g., libgraphite) as used in a browser like Firefox:

```
1 void onPageLoad(int* text) {
2   ...
3   int* screen = ...; // stored in r12
4   int* temp_buf = ...;
5   gr_get_pixel_buffer(text, temp_buf);
6   memcpy(screen, temp_buf, 100);
7   ...
8 }
```

This code calls the libgraphite `gr_get_pixel_buffer` function to render text into a temporary buffer and then copies the temporary buffer to the variable `screen` to be rendered.

Using SFI to sandbox this library ensures that the browser's memory is isolated from `libgraphite` — memory isolation ensures that `gr_get_pixel_buffer` cannot access the memory of `onPageLoad` or any other parts of the browser stack and heap. Unfortunately, memory isolation alone is not enough: if transitions are simply function calls, attackers can violate the calling convention at the application-library boundary (e.g., the `gr_get_pixel_buffer` call and its return) to break isolation. Below, we describe the different ways a compromised libgraphite can do this.

Clobbering Callee-Save Registers. Suppose the `screen` variable in the above `onPageLoad` snippet is compiled down to the register `r12`. In the System V calling convention `r12` is a *callee-saved* register [LMG+18], so if `gr_get_pixel_buffer` clobbers `r12`, then it is also supposed to restore it to its original value before returning to `onPageLoad`. A compromised libgraphite doesn't have to do this; instead, the attacker can poison the register:

```
1 mov r12, ☣
2 ret
```

Since `r12` (`screen`) in our hypothetical example is then used on Line 6 to `memcpy` the `temp_buf` from the sandbox memory, this gives the attacker a write gadget that they can use to hijack Firefox's control flow. To prevent such attacks, we need *callee-save register integrity*, i.e., we must ensure that sandboxed code restores callee-save registers upon returning to the application.

Leaking Scratch Registers. Dually, *scratch registers* can potentially leak sensitive information into the sandbox. Suppose that Firefox keeps a secret (e.g., an encryption key) in a scratch register. Memory isolation alone would not prevent an attacker-controlled libgraphite from using uninitialized registers, thereby reading this secret. To prevent such leaks, we need *scratch register confidentiality*.

Reading and corrupting stack frames. Finally, if the application and sandboxed library share a stack, the attacker could potentially read and corrupt data (and pointers) stored on the stack. To prevent such attacks, we need *stack frame encapsulation*, i.e., we need to ensure that sandboxed code cannot access application stack frames.

5.1.2 Heavyweight Transitions

SFI toolchains — from NaCl [YSD+09] to Wasm native compilers like Lucet [Byt20a] and WAMR [Byt20b] — use *heavyweight transitions* to wrap calls and returns and address the aforementioned attacks. These heavyweight transitions are secure transitions. They provide:

1. Callee-save register integrity. The *springboard* — the transition code which wraps calls — saves callee-save registers to a separate stack stored in protected application memory. When returning from the library to the application, the *trampoline* — the code which wraps returns — restores the registers.

2. Scratch register confidentiality. Since any scratch register may contain secrets, the springboard clears *all* scratch registers before transitioning into the sandbox.

3. Stack frame encapsulation. Most (but not all) SFI systems provision separate stacks for

trusted and sandboxed code and ensure that the trusted stack is not accessible from the sandbox. The springboard and trampoline account for this in three ways. First, they track the separate stack pointers at each transition in order to switch stacks. Second, the springboard copies arguments passed on the stack to the sandbox stack, since sandboxed code cannot access arguments stored on the application stack. Finally, the trampoline tracks the actual return address on transition by keeping it in the protected memory, so that the sandboxed library cannot tamper with it.

The Cost of Wrappers. Heavyweight springboards and trampolines guarantee secure transitions but have two significant drawbacks. First, they impose an overhead on SFI — calls into the sandboxed library become significantly more expensive than simple application function calls (§5.4). Heavyweight transitions conservatively save and clear more state than might be necessary, essentially reimplementing aspects of an OS process switch and duplicating work done by well-behaved libraries. Second, springboards and trampolines must be customized to different platforms, i.e., different processors and calling conventions, and, in extreme cases such as in [VOED+19], even different applications. Implementation mistakes can — and have [Nat09, Nac12, Nac10, Nac11a, Nac11b, BD18] — resulted in sandbox escape attacks.

5.1.3 Zero-Cost Transitions

Heavyweight transitions are conservative because they make few assumptions about the structure (or possible behavior) of the code running in the sandbox. SFI systems like NaCl and Wasm *do*, however, impose structure on sandboxed code to enforce memory isolation. In this section we show that by imposing structure on sandboxed code we can make transitions less conservative. Specifically, we describe a set of *zero-cost conditions* that impose *just enough* internal structure on sandboxed code to ensure that it will behave like a high-level, compositional language while maintaining SFI's high performance. SFI systems that meet these conditions can safely elide almost all the extra work done by heavyweight springboards and trampolines, thus moving toward the ideal of SFI transitions as simple, fast, and portable function calls.

Zero-Cost Conditions. We assume that the sandboxed library code is split into functions and that each function has an expected number of arguments. We *formalize* the internal structure required of library code via a *safety monitor* that checks the zero-cost conditions, i.e., the local requirements necessary to ensure that calls-into and returns-from the untrusted library functions are "well-behaved" and, hence, that they satisfy the secure transition requirements.

1. Callee-save register restoration. First, our monitor enforces function-call level adherence to callee-save register conventions: our monitor tracks callee-save state and checks that it has been correctly restored upon returning. Importantly, satisfying the monitor means that application calls to a well-behaved library function do not require a transition which separately saves and restores callee-save registers, since the function is known to obey the standard calling convention.

2. Well-bracketed control-flow. Second, our monitor requires that the library code adheres to well-bracketed return edges. Abstractly, calls and returns should be well-bracketed: when `f` calls `g` and then `g` calls `h`, `h` ought to return to `g` and then `g` ought to return to `f`. However, untrusted functions may subvert the control stack to implement arbitrary control flow between functions. This unrestricted control flow is at odds with compositional reasoning, preventing *local* verification of functions. Further, subverting well-bracketing could enable an attacker to cause `h` to return directly to `f`. Then, even if `h` and `f` both restore their callee-save registers, those of `g` would be left unrestored. Accordingly, we require two properties of the library to ensure that calls and returns are well-bracketed. First, each jump must stay within the same function. This limits inter-function control flow to function calls and returns. Second, the (specification) monitor maintains a "logical" call stack, which is used to ensure that returns go only to the preceding caller.

3. Type-directed forward-edge CFI. Our monitor also requires that library code obeys type-directed forward-edge CFI. That is, for every call instruction encountered during execution, the jump target address is the start of a library function and the arguments passed match those expected by the called function. This ensures that each function starts from a (statically) known

stack shape, preventing a class of attacks where a benign function can be tricked into overwriting other stack frames or hijacking control flow because it is passed too few (or too many) arguments. If this were not the case, a locally well-behaved function that was passed too few arguments could write to a saved register or the saved return address, expecting that stack slot to be the location of an argument.

4. Local state encapsulation. Our monitor establishes *local state encapsulation* by checking that all stack reads and writes are within the current stack frame. This check allows us to *locally*, i.e., by checking each function in isolation, ensure that a library function correctly saves and restores callee-save registers upon entry and exit. To see why local state encapsulation is needed, consider the following idealized assembly function `library_func`:

```
1  library_func:          library_helper:
2    push r12               store sp - 1 := ☣
3    mov r12 ← 1            ret
4    load r1 ← sp - 1
5    add r1 ← r12
6    call library_helper
7    pop r12
8    ret
```

If `library_helper` is called it will overwrite the stack slot where `library_func` saved `r12`, and `library_func` will then "restore" `r12` to the attacker's desired value. Our monitor prohibits such cross-function tampering, thus ensuring that all subsequent reasoning about callee-save integrity can be carried out locally in each function.

5. Confidentiality. Finally, our monitor uses dynamic information flow control (IFC) tracking to define the confidentiality of scratch registers. The monitor tracks how (secret application) values stored in scratch registers flow through the sandboxed code, and checks that the library code does not leak this information. Concretely, our implementations enforce this by ensuring that, within each function's localized control flow, all register and local stack variables are initialized before

use.

The individual properties making up our zero-cost conditions are well-known to be beneficial to software security, and their enforcement in low-level code has been extensively studied (§5.6): our insightis that in conjunction these conditions are *sufficient* to eliminate heavyweight transitions in SFI systems, which can currently be a source of significant overhead when sandboxing arbitrary code. Indeed, in Section 5.2.1 we show that the Wasm type system is strict enough to ensure that a Wasm compiler generates native code that already meets these conditions. To increase the trustworthiness of this zero-cost compatible Wasm, in Section 5.3 we describe a verifier that statically checks that compiled Wasm code meets the zero-cost conditions. Further, in Section 5.2.2 we demonstrate how the zero-cost conditions can be used to design a new SFI scheme by combining hardware-backed memory isolation with existing LLVM compiler passes.

5.2 Instantiating Zero-Cost

We describe two isolation systems that securely support zero-cost transitions: they meet the overlay monitor zero-cost conditions. The first (§5.2.1) is an SFI system using WebAssembly as an IR before compiling to native code using the Lucet toolchain [Byt20a]. Here we rely on the language-level invariants of Wasm to satisfy our zero-cost requirements. To ensure that these invariants are maintained, in Section 5.3 we describe a verifier, VeriZero, that checks that compiled binaries meet the zero-cost conditions.

The second system, `SegmentZero32`, is our novel SFI system combining the x86 segmented memory model for memory isolation with several security-hardening LLVM compiler passes to enforce our zero-cost conditions. While WebAssembly meets the zero-cost conditions, it imposes additional restrictions that lead to unrelated slowdowns. `SegmentZero32` thus serves as a platform for evaluating the potential cost of enforcing the zero-cost conditions directly as

well as a proof-of-concept SFI implementation designed using the zero-cost framework.

5.2.1 WebAssembly

WebAssembly (Wasm) is a low-level bytecode with a sound, static type system. Wasm's abstract state includes global variables and heap memory, which are zero-initialized at start-up. All heap accesses are explicitly bounds checked, meaning that compiled Wasm programs inherently implement heap isolation. Beyond this, Wasm programs enjoy several language-level properties, which ensure compiled binaries satisfying the zero-cost conditions. We describe these below.

Control Flow. There are no arbitrary jump instructions in Wasm, only structured intra-function control flow. Functions may only be entered through a call instruction, and may only be exited by executing a return instruction. Functions also have an associated type; direct calls are type-checked at compile time while indirect calls are subject to a runtime type check. This ensures that compiled Wasm meets our type-directed forward-edge CFI condition.

Protecting the Stack. A Wasm function's type precisely describes the space required to allocate the function's stack frame (including spilled registers). All accesses to local variables and arguments are performed through statically known offsets from the current stack base. It is therefore impossible for a Wasm operation to access other stack frames or alter the saved return address. This ensures that compiled Wasm meets our local state encapsulation condition, and, in combination with type-checking function calls, guarantees that Wasm's control-flow is well-bracketed. We therefore know that compiled Wasm functions will always execute the register-saving preamble and, upon termination, will execute the register-restoring epilogue. Further, the function body will not alter the values of any registers saved to the stack, thereby ensuring restoration of callee-save registers.

Confidentiality. Wasm code may store values into function-local variables or a function-local

"value stack" similar to that of the Java Virtual Machine [jvm19]. The Wasm spec requires that compilers initialize function-local variables either with a function argument or with a default value. Further, accesses to the Wasm value stack are governed by a coarse-grained data-flow type system, with explicit annotations at control flow joins. These are used to check at compile-time that an instruction cannot pop a value from the stack unless a corresponding value was pushed earlier in the same function. This guarantees that local variable and value stack accesses can be compiled to register accesses or accesses to a statically-known offset in the stack frame.

When executing a compiled Wasm function without heavyweight transitions, confidential values from prior computations may linger in these spilled registers or parts of the stack. However, the above checks ensure that these locations will only be read if they have been previously overwritten during execution of the same function by a low-confidentiality Wasm library value.

5.2.2 SegmentZero32

To demonstrate that zero-cost conditions can be applied outside of highly structured languages such as Wasm, we demonstrate their enforcement in our novel SFI system for C code called `SegmentZero32`. As we mention in §5.1.3, our zero-cost conditions amalgamate a number of individual conditions which separately have well-studied enforcement mechanisms, and so we are able to compose a series of off-the-shelf Clang/LLVM security-hardening passes to form the core of `SegmentZero32`. The memory bounds checks are performed using the x86 segmented memory model [Int20a] (Similar to NaCl [YSD+09], however we use an additional segment to separate the sandboxed heap and stack).

Since `SegmentZero32` directly enforces the structure required for zero-cost transitions on C code (rather than relying on Wasm as an IR), it allows us to investigate the intrinsic cost of enforcing zero-cost (See Section 5.4.3), without suffering from irrelevant Wasm overheads. We additionally compare `SegmentZero32` against NaCl's 32-bit SFI scheme for the x86 architecture, which we believe is the fastest production-quality SFI toolchain currently available. Below we

discuss specific details `SegmentZero32` zero-cost condition enforcement.

Protecting the Stack. We apply the SafeStack [KSP+14, The21b] compiler pass to further split the sandboxed stack into a safe and unsafe stack. The safe stack contains only data that the compiler can statically verify is always accessed safely, e.g., return addresses, spilled registers, and allocations that are only accessed locally using verifiably safe offsets within the function that allocates them.[3] All other stack values are moved to the heap segment. This ensures that pointer manipulation of unsafe stack references cannot be used to corrupt the return address and saved context of the current call. We write a small LLVM pass to add additional support for tracking whether an access must be made through the heap segment or the stack segment, ensuring correct code generation.

These transformations ensure that malicious code cannot programmatically access anything stored in the stack segment, except through offsets statically determined to be safe by the SafeStack pass. This protects the stored callee-save registers and return address, guaranteeing the restoration of callee-save registers and well-bracketing *iff forward control flow is enforced*.

Control Flow. Fortunately, enforcing forward-edge CFI has been widely studied [BCN+17]. We use a CFI pass as implemented in Clang/LLVM [The21a, TRC+14] including flags to dynamically protect indirect function calls, ensuring forward control flow integrity. Further, `SegmentZero32` conservatively bans non-local control flow (e.g. `setjmp/longjmp`) in the C source code. A more permissive approach is possible, but we leave this for future work.

Confidentiality. To guarantee confidentiality we implement a small change in Clang to zero initialize all stack variables.[4] This ensures that scratch registers cannot leak secrets as all sandbox values are semantically written before use. In practice, many of these writes are statically known to be dead and therefore optimised away.

[3] We also use LLVM's stack-heap clash detection (-fstack-clash-protection) to prevent the stack growing into the heap.

[4] We can't use Clang's existing pass for variable initialization [The18] as it zero initializes data on the unsafe stack leading to poor performance

```
bad_func: [] → rax
  push r12
  ; TRACK: stack[0] = initial r12 value
  mov r12 ← 1
  ; TRACK: r12 initialized
  mov r11 ← r13 + r12
  ; TRACK: r11 uninitialized
  mov rdi ← 2
  ; TRACK: rdi initialized
  ; ASSERT: good_func arguments initialized
  call good_func
  ; TRACK: good_func return value initialized
  pop r12
  ; TRACK: r12 = initial r12 value
  ; ASSERT: callee-save registers restored
  gateret
```

```
good_func: [rdi] → rax
  mov rax ← rdi
  ret
```

Figure 5.1: Disassembled and lifted WebAssembly functions

5.3 Verifying Compiled WebAssembly

Instead of trusting the Wasm compiler, we build a *zero-cost verifier*, VeriZero, to check that the native, compiled output meets the zero-cost conditions and is thus safe to run without springboards and trampolines. VeriZero is a static x86 assembly analyzer that takes as input potentially untrusted native programs and verifies a series of local properties via abstract interpretation. Together these local properties guarantee that the monitor checks defined in oSFIasm are met.

VeriZero extends the VeriWasm SFI verifier [JTA+21]. Both operate over WebAssembly modules compiled by the Lucet Wasm compiler [Byt20a], first disassembling the native x86 code before computing a control-flow graph (CFG) for each function in the binary. The disassembled code is then lifted to a subset of SFIasm, which serves as the first abstract domain in our analysis. Unfortunately, the properties checked by VeriWasm, while sufficient to guarantee SFI security, are insufficient to guarantee zero-cost security. Below we will describe how VeriZero extends VeriWasm to guarantee the stronger zero-cost conditions are met.

5.3.1 The VeriZero Analyzers

VeriZero adds two new analyses to VeriWasm. The first extends VeriWasm's CFI analysis, which only captures coarse grained control-flow (i.e., that all calls target valid sandboxed functions), to also extract type information. Extracting type information from the binary code is possible without any complex type inference because Lucet leaves the type signatures in the compiled output (though we do not need to trust Lucet to get these type signatures correct since VeriZero would catch any deviations at the binary level). For direct calls, VeriZero simply extracts the WebAssembly type stored in the binary. For indirect calls we extend the VeriWasm indirect call analysis to track the type of each indirect call table entry, enabling us to resolve each indirect call to a statically known type. These types correspond to the input registers and stack slots, and the output registers (if any) used by a function. For example, in Figure 5.1 `bad_func` takes no input and outputs to `rax` and `good_func` takes `rdi` as input and outputs to `rax`.

The second analysis tracks dataflow in local variables, i.e., in registers and stack slots. Continuing with `bad_func` as our example this analysis captures that: in Line 2 stack slot 0 now holds the initial value of `r12`, in Line 4 `r12` holds an initialized (and therefore public) value, in Line 6 `r13` has not been initialized and therefore potentially contains confidential data so `r11` may also contain confidential data, etc. This analysis is used to check confidentiality, callee-save register restoration, local state encapsulation, and is combined with the previous analysis to check type-directed CFI.

5.3.2 The Dataflow Abstract Domain

To track local variable dataflow, VeriZero uses an abstract domain with three elements: `Uninitialized` which represents an uninitialized, potentially confidential value; `Initialized` which represents an initialized, public value; and `UninitializedCallee`(r) which represents a potentially confidential value which corresponds to the original value of the callee-save register

r. The domain forms a meet-semilattice with `Uninitialized` the least element and all other elements incomparable.

From here, analysis is straightforward, with a function's argument registers and stack slots initialized to `Initialized`, each callee-save register r initialized to `UninitializedCallee`(r), and everything else `Uninitialized`. Instructions are interpreted as expected, e.g., `mov` simply copies the abstract value of its source into the target, operations return the meet of their operands, and all constants and reads from the heap are treated as initialized. Across calls we assume that callee-save register conventions are followed (as we will be checking this), preserving the value of all callee-save registers and clearing all other registers' values. We extract the type information from the extended CFI analysis to determine the return register that is initialized after a function call.

5.3.3 Checking the Zero-Cost Conditions

The above two analyses, along with additional information from VeriWasm's existing analyses enable us to check the zero-cost conditions.

1. *Callee-save register restoration:* The `UninitializedCallee`(r) value enables straightforward checking that callee-save registers have been restored by checking that, at each `ret` instruction, each callee-save register r has the abstract value `UninitializedCallee`(r).

2. *Well-bracketed control-flow:* VeriWasm already implements a stack checker that guarantees that all writes to the stack are to local variables, ensuring that the saved return address on the stack cannot be tampered with. Further, it checks that the stack pointer is restored to its original location at the end of every function, ensuring the saved return address is used.

3. *Type-directed forward-edge CFI:* The dataflow analysis gives us the registers that are initialized when we reach a `call` instruction, enabling us to check that the input arguments of the target have been initialized. For example, when we reach Line 11 we know that `rdi` has the value

`Initialized`. The type-based CFI analysis tells us that `good_func` expects `rdi` as an input, so this call is marked as safe.

4. *Local state encapsulation:* To ensure SFI security, VeriWasm checks that no writes are below the current stack frame, ensuring that verified Wasm functions cannot tamper with other frames.

5. *Confidentiality:* We check confidentiality using the information obtained in our dataflow analysis, where the value `Initialized` ensures that a value is initialized with a public, non-confidential value. This enables us to check each of the confidentiality checks encoded in oSFIasm are met: for instance the type-safe forward-edge CFI check described above already ensures each argument is initialized. In Figure 5.1, the confidentiality checker will flag Line 6 as unsafe because `r13` still has the value `UninitializedCallee(r13)`, which potentially contains confidential information leaked from the application.

5.4 Evaluation

We evaluate our zero-cost model by asking four questions:

- **Q1**: What is the cost of a context switch? (§5.4.1)
- **Q2**: What is end-to-end performance gain of Wasm-based SFI due to zero-cost transitions? (§5.4.2)
- **Q3**: What is the performance overhead of purpose-built zero-cost SFI enforcement? (§5.4.3)
- **Q4**: Is the VeriZero verifier effective? (§5.4.4)

Since our zero-cost condition enforcement does incur some runtime overhead, **Q2** and **Q3** are heavily workload-dependent. The benefit a workload receives from the zero-cost approach will be in direct proportion to the frequency with which it performs domain transitions.

Systems. To investigate the first three questions, we consider two groups of SFI systems. The first group compares a number of different transition models for Wasm-based SFI for 64-bit binaries, built on top of the Lucet compiler [Byt20a]. All of these will have identical runtime overhead, meaning that the only variance between them will be due to transition overhead. The `WasmLucet` build uses the original heavyweight springboards and trampolines shipped with the Lucet runtime written in Rust. `WasmHeavy` adopts techniques from NaCl and uses optimized assembly to save and restore application context during transitions. `WasmZero` implements our zero-cost transition system, meaning transitions are simple function calls. To understand the overhead of register saving/restoring and stack switching, we also evaluate a `WasmReg` build which saves/restores registers like `WasmHeavy`, but shares the library and application stack like `WasmZero`.

The second group compares optimized SFI techniques for 32-bit binaries. Wasm-based SFI imposes overheads far beyond what is strictly necessary to enforce our zero-cost conditions, both because of the immaturity of the Lucet compiler in comparison to more established compilers such as Clang, and because Wasm inherently enforces additional restrictions on compiled code (e.g., structured intra-function control flow). We design `SegmentZero32` (§5.2.2) to enforce only our zero-cost-conditions and nothing more, aiming to benchmark it against the Native Client 32-bit isolation scheme (`NaCl32`) [YSD+09], arguably the fastest production SFI system available, which requires heavyweight transitions. Both systems make use of memory segmentation, a 32-bit x86-only feature for fast memory isolation. Unfortunately, we cannot make a uniform comparison between `NaCl32`, `SegmentZero32`, and `WasmZero` since Lucet only supports a 64-bit target.

Each group additionally uses unsandboxed, insecure native execution (`Vanilla`) as a baseline. To represent the best possible performance of schemes relying on heavyweight transitions, we also benchmark `IdealHeavy32` and `IdealHeavy64`, ideal hardware isolation schemes, which incur no runtime overhead but require heavyweight transitions. To simulate the performance of these ideal schemes, we simply measure the performance of native code with heavyweight

trampolines.

We integrate all of the above SFI schemes into Firefox using the RLBox framework [NDG+20]. Since RLBox already provides plugins for the `WasmLucet` and `NaCl32` builds, we only implement the plugins for the remaining system builds.

Benchmarks. We use a micro-benchmark to evaluate the cost of a single transition for our different transition models, using unsandboxed native calls as a baseline (**Q1**).

We answer questions **Q2–Q3** by measuring the end-to-end performance of font and image rendering in Firefox, using a sandboxed libgraphite and libjpeg, respectively. We use these libraries because they have many cross-sandbox transitions, which [NDG+20] previously observed to affect the overall browser performance. To evaluate the performance of libgraphite, we use Kew's benchmark[5], which reflows the text on a page ten times, adjusting the size of the fonts each time to negate the effects of font caches. When calling libgraphite, Firefox makes a number of calls into the sandbox roughly proportional to the number of glyphs on the page. We run this benchmark 100 times and report the median execution time below (all values have standard deviations within 1%).

To evaluate the performance of libjpeg, we measure the overhead of rendering images of varying complexity and size. Since the work done by the sandboxed libjpeg, per call, is proportional to the width of the image — Firefox executes the library in *streaming mode*, one row at a time — we consider images of different widths, keeping the image height fixed. This allows us to understand the benefits and limitations of zero-cost transitions, since the proportion of execution time spent context-switching decreases as the image width increases. We do this for three images, of varying complexity: a simple image consisting of a single color (`SimpleImage`), a stock image from the Image Compression benchmark suite[6] (`StockImage`), and an image of random pixels (`RandomImage`). We render each image 500 times and report the median time (standard deviations are all under 1%).

[5]Available at https://jfkthame.github.io/test/udhr_urd.html

[6]Online: https://imagecompression.info/test_images/. Visited Dec 9, 2020.

Finally, we use SPEC CPU® 2006 to partly evaluate the sandboxing overhead of our purpose-built `SegmentZero32` SFI system (**Q3**), and to measure VeriZero's verification speed (**Q4**).

Machine and Software Setup. We run all but the verification benchmarks on an Intel® Core™ i7-6700K machine with four 4GHz cores, 64GB RAM, running Ubuntu 20.04.1 LTS (kernel version 5.4.0-58). We run benchmarks with a shielded isolated cpuset [Tsa21] consisting of one core with hyperthreading disabled and the clock frequency pinned to 2.2GHz. We generate Wasm sandboxed code in two steps: First, we compile C/C++ to Wasm using Clang-11, and then compile Wasm to native code using the fork of the Lucet used by RLBox (snapshot from Dec 9, 2020). We generate NaCl sandboxed code using a modified version of Clang-4. We compile all other C/C++ source code, including `SegmentZero32` sandboxed code and benchmarks using Clang-11. We implement our Firefox benchmarks on top of Firefox Nightly (from August 22, 2020).

Summary of Results. We find that the performance of Wasm-based isolation can be significantly improved by adopting zero-cost transitions, but that Lucet-compiled WebAssembly's runtime overhead means that it does not outperform more optimised isolation schemes in end-to-end tests. The low performance overhead of `SegmentZero32` demonstrates that these runtime overheads are not inherent to the zero-cost approach, and that an optimised zero-cost SFI system can significantly outperform more traditional schemes, especially for workloads with a large number of transitions. Finally, we find that we can efficiently check zero-cost conditions at the binary level, for Lucet compiled code, with no false positives.

5.4.1 The Cost of Transitions

We measure the cost of different cross-domain transitions — direct and indirect calls into the sandbox, callbacks from the sandbox, and syscall invocations from the sandbox — for the different system builds described above. To expose overheads fully, we choose extremely fast payloads—either a function that just adds two numbers or the `gettimeofday` syscall, which

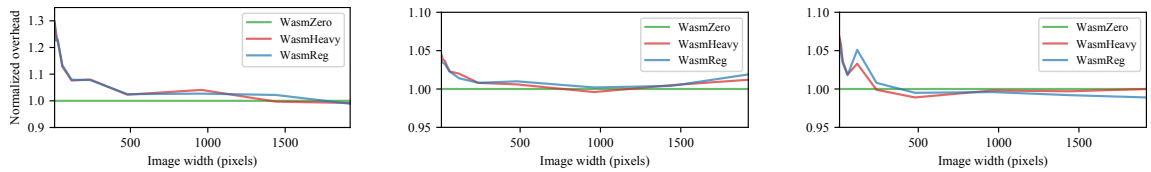
Table 5.1: Costs of transitions in different isolation models. Zero-cost transitions are shown in **boldface**. `Vanilla` is the performance of an unsandboxed C function call, to serve as a baseline.

Build	Direct call	Indirect call	Callback	Syscall
`Vanilla` (in C)	1ns	56ns	56ns	24ns
`WasmLucet`	—	1137ns	—	—
`WasmHeavy`	120ns	209ns	172ns	192ns
`WasmReg`	120ns	210ns	172ns	192ns
`WasmZero`	**7ns**	**66ns**	**67ns**	**60ns**
`Vanilla` (in C, 32-bit)	1ns	74ns	74ns	37ns
`NaCl32`	—	714ns	373ns	356ns
`SegmentZero32`	**24ns**	**108ns**	**80ns**	**88ns**

relies on Linux's vDSO to avoid CPU ring changes. The results are shown in Figure 5.1. All numbers are averages of one million repetitions, and repeated runs have negligible standard deviation.[7]

We make several observations. First, among Wasm-based SFI schemes, zero-cost transitions (`WasmZero`) are significantly faster than even optimized heavyweight transitions (`WasmHeavy`). Lucet's existing indirect calls written in Rust (`WasmLucet`) are significantly slower than both. Second, the cost of stack switching (the difference of `WasmHeavy` and `WasmReg`) is surprisingly negligible. Third, the performance of `Vanilla` and `WasmZero` should be identical but is not. This is *not* because our transitions have a hidden cost. Rather, it's because we are comparing code produced by two different compilers: `Vanilla` is native code produced by Clang, while `WasmZero` is code produced by Lucet, and Lucet's code generation is not yet highly optimized [Han19b]. For example, in the benchmark that adds two numbers, Clang eliminates the function prologue and epilogue that save and restore the frame pointer, while Lucet does not. We observe similar trends for hardware-based isolation. For example, we find that `SegmentZero32` transitions are much faster than `IdealHeavy32` and `NaCl32` transitions and only 23ns slower than `Vanilla` for direct calls. Finally, we observe that `SegmentZero32` is slower than `WasmZero`:

[7] Lucet and NaCl don't support direct sandbox calls; Lucet further does not support custom callbacks or syscall invocations.



(a) `SimpleImage` (b) `StockImage` (c) `RandomImage`

Figure 5.2: Performance of different Wasm transitions on rendering of (a) a simple image with one color, (b) a stock image, and (c) a complex image with random pixels, normalized to `WasmZero`. `WasmZero` transitions outperform other transitions. The difference diminishes with width, but narrower images are more common on the web.

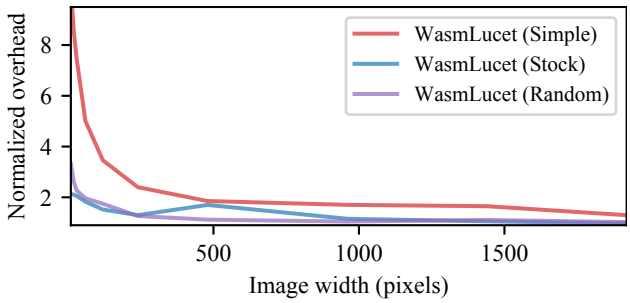


Figure 5.3: Performance of the `WasmLucet` heavyweight transitions included in the Lucet runtime on the image benchmarks in Section 5.4. Performance when rendering (a) a simple image with one color, (b) a stock image and (c) a complex image with random pixels. The performance is the overhead compared to `WasmZero`.

hardware isolation schemes like `SegmentZero32` and `NaCl32` execute instructions to enable or disable the hardware based memory isolation in their transitions.

5.4.2 End-to-End Performance Improvements of Zero-Cost Transitions for Wasm

We evaluate the end-to-end performance impact of the different transition models on Wasm-sandboxed font and image rendering as used in Firefox (see §5.4).

Font Rendering. We report the performance of libgraphite isolated with Wasm-based schemes on Kew's benchmark below:

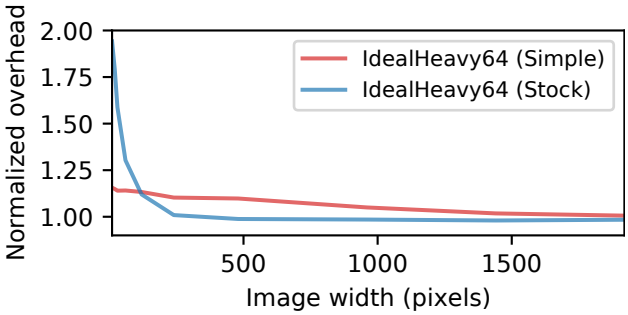


Figure 5.4: Performance of an ideal isolation scheme (no enforcement overhead) with heavy trampolines when rendering images. Wasm compilers whose enforcement overhead is lower than this can outperform even an ideal isolation scheme that uses heavy weight transitions.

	`WasmLucet`	`WasmHeavy`	`WasmReg`	`WasmZero`	`Vanilla`	`IdealHeavy64`
Font render	8173ms	2246ms	2230ms	2032ms	1116ms	1563ms

As expected, Wasm with zero-cost transitions (`WasmZero`) outperforms the other Wasm-based SFI transition models. Compared to `WasmZero`, Lucet's existing transitions slow down rendering by over 4×.[8] But, even the optimized heavyweight transitions (`WasmHeavy`) impose a 10% performance tax. This overhead is due to register saving/restoring; stack switching only accounts for 0.8% overhead.

While these results show that existing Wasm-based isolation schemes can benefit from switching to zero-cost transitions — and indeed the speed-up due to zero-cost transitions allowed Mozilla to ship the Wasm-sandboxed libgraphite — they also show that Lucet-compiled Wasm is slow (∼80% slower than Vanilla). This, unfortunately, means that the transition cost savings alone are not enough to beat `IdealHeavy64`, even for a workload with many transitions. To compete with this ideal SFI scheme with heavyweight transitions, we would need to reduce the runtime overhead to ∼40%. [JPBG19] report the average runtime overhead of Mozilla SpiderMonkey JIT-compiled WebAssembly compared to native as ∼45% in a different set of benchmarks, while noting many correctable inefficiencies in the generated assembly code, suggesting that there is a lot of room for Lucet to be further optimised.

[8]This overhead is smaller than the 8× overhead reported by [NDG+20]; we attribute this difference to the different compilers — we use a more recent, and faster, version of Lucet.

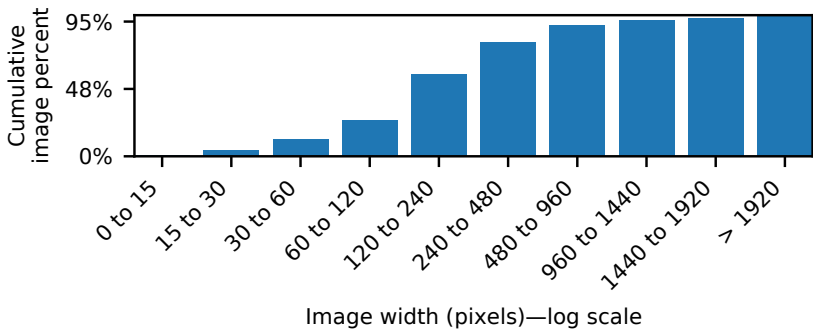


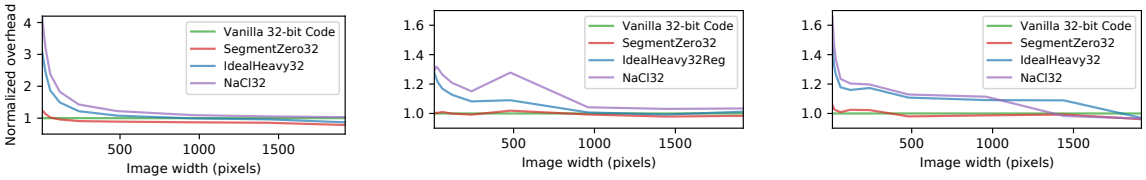
Figure 5.5: Cumulative distribution of image widths on the landing pages of the Alexa top 500 websites. Over 80% of the images have widths under 480 pixels. Narrower images have a higher transition rate, and thus higher relative overheads when using expensive transitions.

Image Rendering. Figure 5.2 report the overhead of Wasm-based sandboxing on image rendering, normalized to `WasmZero` to highlight the relative overheads of different transitions as compared to our zero-cost transitions. We report results of `WasmLucet` separately, in Figure 5.3 because the rendering times are up to 9.2× longer than the other builds. Here, we instead focus on evaluating the overheads of optimized heavy transitions.

As expected, `WasmZero` significantly outperforms other transitions when images are narrower and simpler. On `SimpleImage`, `WasmHeavy` and `WasmLucet` can take as much as 29.7% and 9.2× longer to render the image as with `WasmZero` transitions. However, this performance gap diminishes as image width increases (and the relative cost of context switching decreases). For `StockImage` and `RandomImage`, the `WasmHeavy` trends are similar, but the rendering time differences start at about 4.5%. Lucet's existing transitions (`WasmLucet`) are still significantly slower than zero-cost transitions (`WasmZero`) even on wide images.

Though the differences between the transitions are smaller as the image width increases, many images on the Web are narrow. Figure 5.5 shows the distribution of images on the landing pages of the Alexa top 500 websites. Of the 10.6K images, 8.6K (over 80%) have widths between 1 and 480 pixels, a range in which zero-cost transitions noticeably outperform the other kinds of transitions.

Like font rendering, we measure the target runtime overhead Lucet should achieve to beat



(a) `SimpleImage` (b) `StockImage` (c) `RandomImage`

Figure 5.6: Performance of image rendering with libjpeg sandboxed with `SegmentZero32` and `NaCl32` and `IdealHeavy32`. Times are relative to unsandboxed code. `NaCl32` and `IdealHeavy32` relative overheads are as high as 312% and 208% respectively, while `SegmentZero32` relative overheads do not exceed 24%.

`IdealHeavy64` end-to-end for rendering images. We report our results in Figure 5.4. For the small simple image, we observe this to be 94% — this is approximately the overhead of Lucet that we see already today. For the small stock image, we observe this to be 15% — this is much smaller than the overhead of Lucet today, but lower overheads have been demonstrated on some benchmarks by the prototype Wasm compiler of [GMP+20].

5.4.3 Performance Overhead of Purpose-Built Zero-Cost SFI Enforcement

In this section, we measure the performance overhead of `SegmentZero32` with zero-cost transitions. We compare `SegmentZero32` with NaCl (`NaCl32`) and `IdealHeavy32`— a hypothetical SFI scheme with no isolation enforcement overhead, both of which rely on heavyweight transitions. We measure the overhead of these systems on the standard SPEC CPU® 2006 benchmark suite, and the libgraphite and libjpeg font and image rendering benchmarks. Since both `SegmentZero32` and `NaCl32` use segmentation which is supported only in 32-bit mode, we implement these three isolation builds in 32-bit mode and compare it to native 32-bit unsandboxed code. We describe these benchmarks next.

SPEC. We report the impact of sandboxing on SPEC CPU® 2006 in Figure 5.2. Several benchmarks are not compatible with `NaCl32`; augmenting `NaCl32` runtime and libraries to fix such compatibility issues (e.g., as done in [YSD+09] for SPEC2000) is beyond the scope of this paper. The `gcc` benchmark, on the other hand, is not compatible with `SegmentZero32`—

Table 5.2: Overheads compared to native code on SPEC CPU® 2006 (nc), for NaCl32 and SegmentZero32.

System	400.perlbench	401.bzip2	403.gcc	429.mcf	445.gobmk	456.hmmer
NaCl32	—	—	1.10×	—	1.27×	0.97×
SegmentZero32	1.20×	1.08×	—	1.04×	1.25×	0.82×

System	458.sjeng	462.libquantum	464.h264ref	471.omnetpp	473.astar	483.xalancbmk
NaCl32	1.20×	1.06×	1.34×	1.06×	1.31×	—
SegmentZero32	1.16×	1.02×	1.01×	1.01×	1.10×	1.05×

gcc fails (at runtime) because the CFI used by SegmentZero32 — Clang's CFI — incorrectly computes a target CFI label. Clang's CFI implementation is more precise than necessary for our zero-cost conditions; as with NaCl32, we leave the implementation of a coarse-grain and more permissive CFI to future work. On the overlapping benchmarks, SegmentZero32's overhead is comparable to NaCl32's.

Font Rendering. The impact of these isolation schemes on font rendering is shown below:

	Vanilla (32-bit)	**IdealHeavy32**	**NaCl32**	**SegmentZero32**
Font render	1441ms	2399ms	2769ms	1765ms

We observe that NaCl32 and IdealHeavy32 impose an overhead of 92% and 66% respectively. In contrast, SegmentZero32 has a smaller overhead (22.5%) as it does not have to save and restore registers or switch stacks. We attribute the overhead of SegmentZero32 (over Vanilla) to three factors: (1) changing segments to enable/disable isolation during function calls, (2) using indirect function calls for cross-domain calls (a choice that simplifies engineering but is not fundamental), and (3) the structure imposed by our zero-cost condition enforcement.

Image Rendering. We report the impact of sandboxing on image rendering in Figure 5.6. For narrow images (10 pixel width), SegmentZero32 overheads relative to the native unsandboxed code are 24%, 1%, and 6.5% for SimpleImage, StockImage and RandomImage, respectively. These overheads are lower than the corresponding overheads for NaCl32 (312%, 29%, and 66%)

as well as `IdealHeavy32` (208%, 28% and 45%). As in the Wasm measurements, these overheads shrink as image width increases and the complexity of the image increases (e.g., the overheads for images wider than 480 pixels are negligible).

5.4.4 Effectiveness of the VeriZero Verifier

We evaluate VeriZero's effectiveness by using it to (1) verify 13 binaries — five third-party libraries shipping (or about to ship) with Firefox compiled across 3 binaries, and 10 binaries from the SPEC CPU® 2006 benchmarks — and (2) find nine manually introduced bugs, inspired by real calling convention bugs in previous SFI toolchains [Ryd20, Nac10, Nac12]. We measure VeriZero's performance verifying the aforementioned 13 binaries. Finally, we stress test VeriZero by running it on random binaries generated by Csmith [YCER11].

Experimental Setup. We run all VeriZero experiments on a 2.1GHz Intel® Xeon® Platinum 8160 machine with 96 cores and 1 TB of RAM running Arch Linux 5.11.12. All experiments run on a single core and use no more than 2GB of RAM. We compile the SPEC binaries used using the Lucet toolkit used in Section 5.4.2. We verify three of the Firefox libraries from Firefox Nightly; we compile the other two from the patches that are in the process of being integrated into Firefox.

Effectiveness and Performance Results. VeriZero successfully verifies the 13 Firefox and SPEC CPU® 2006 binaries. These binaries vary in size from 150 functions (the `lbm` benchmark from SPEC CPU® 2006) to 4094 functions (the binary consisting of the Firefox Nightly libraries libogg, libgraphite, and hunspell). It took VeriZero between 1.77 seconds and 19.28 seconds to verify these binaries, with an average of 9.2 seconds and median of 5.93 seconds. VeriZero's performance is on par with the original VeriWasm's performance: on the 10 SPEC CPU® 2006 benchmarks evaluated in the VeriWasm paper [JTA+21] VeriZero is slightly (15%) faster, despite checking zero-cost conditions in addition to all of VeriWasm's original checks. This is due to various engineering improvements that were made to VeriWasm in the course of developing

VeriZero.

VeriZero also successfully found bugs injected into nine binaries. These bugs tested all the zero-cost properties that VeriZero was designed to check, and when possible they were based on real bugs (like those in Cranelift [Ryd20]). VeriZero successfully detected all nine of these bugs, giving us confidence that VeriZero is capable of finding violations of the zero-cost conditions.

Fuzzing Results. We fuzzed VeriZero to both search for potential bugs in Lucet, as well as to ensure VeriZero does not incorrectly declare safe programs unsafe. The fuzzing pipeline works in four stages: first, we use Csmith [YCER11] to generate random C and C++ programs, next we use Clang to compile the generated C/C++ program to WebAssembly, followed by compiling the Wasm file to native code using Lucet, and finally we verify the generated binary with VeriZero. As these programs were compiled by Lucet, we expect them to adhere to the zero-cost conditions, and any binaries flagged by VeriZero are either bugs in Lucet or are spurious errors in VeriZero.

While we did not find bugs in Lucet, fuzzing did find cases where VeriZero triggered spurious errors. After fixing these errors, we verified 100,000 randomly generated programs with no false positives.

5.5 Limitations

Our Wasm SFI scheme is capable of sandboxing any C/C++-like language (with arbitrary intra-function control flow, arbitrary type casting, arbitrary pointer aliasing, arithmetic etc.) that can compile to Wasm, so long as it does not use features which Wasm must emulate through JavaScript[9] — most prominently C++-style exceptions, `setjmp/longjmp`, and multithreading. These limitations are not inherent to our zero-cost conditions, and Wasm is in the process of being extended with support for all of the above features [WRPP19, Web21].

Our `SegmentZero32` scheme is built as a proof-of-concept, using mostly existing LLVM

[9] See https://emscripten.org/

passes to sandbox C programs compiled to 32-bit x86, as an approach to understanding the overhead of zero-cost conditions on native code. As such, our `SegmentZero32` implementation does not support, for instance, `setjmp/longjmp` or multithreading (similar to Wasm). It also does not support user-defined variadic function arguments or position independent code. However, these limitations are not fundamental. For example, variadic function arguments could be supported by extending the SafeStack pass to move the variadic argument buffer into the unsafe stack, and position independent code could be supported through minor generalisations of existing compiler primitives.

Both Wasm and `SegmentZero32` rely on a type-directed forward-edge CFI which requires us to statically infer a limited amount of information about arguments to functions.[10] This information includes the number of arguments, their width, and the calling convention. In practice, this information is readily available as part of compilation and does not require any complex control flow inference (unlike more precise fine grain CFI schemes), so this is only a limitation when analyzing certain binary programs. Like most SFI schemes, both Wasm and our `SegmentZero32` do not currently support JIT code compilation within the isolated component; adding this would require engineering work, but can be done following the approaches of [AME+11, VOED+19]. Finally, side channels are out of scope for this paper.

5.6 Related work

A considerable amount of research has gone into efficient implementations of memory isolation and CFI techniques to provide SFI across many platforms [SMB+10, MM06, GCF15, Byt20a, LSW95, ATLLW96, SESS96, HBG+09, NT14, STM10, FC08, Tan17, PG11, BMHK08, LVOE+16, CRSL16, CCM+09]. However, these systems either implement or require the user to implement heavyweight springboards and trampolines to guarantee security.

[10] We do not need to infer any information about the heap or unsafe stack. Variadic functions, for example, can pass a dynamic number of arguments on the unsafe stack.

SFI Systems. [WLAG93] suggest two ways to optimize transitions: (1) partitioning the registers used by the application and the sandboxed component and (2) performing link time optimizations (LTO) that conservatively eliminates register saves that are never used in the entire sandboxed component (not just the callee). Register partitioning would cause slowdowns due to increased spilling. Native Client [YSD+09] optimized transitions by clearing and saving contexts using machine specific mechanisms to, e.g., clear floating point state and SIMD registers in bulk. However, we show (§5.4) that, even with those optimizations, the software model imposes significant transition overheads. While CPU makers continue to add optimized context switching instructions, such instructions do not yet eliminate all overhead.

[ZTM11] combine an SFI scheme with a rich CFI scheme enforcing structure on executing code. While a similar approach, their goal is to safely perform optimizations to elide SFI and CFI bounds checks, and they do not impose sufficient structure to enforce well-bracketing, a necessary property for zero-cost transitions. XFI [EAV+06] also combines an SFI scheme with a rich CFI scheme and adopts a safe stack model. While meeting many of the zero-cost conditions, it does not prevent reading uninitialized scratch registers and therefore cannot ensure confidentiality without heavyweight springboards that clear scratch registers. They also do not specify the CFG granularity, so it is not clear if is strong enough to satisfy the zero-cost type-safe CFI requirement.

WebAssembly Based Isolation. WasmBoxC [Zak20] sandboxes C code by compiling to Wasm followed by (de)compiling back to C, ensuring that the sandboxed code will inherit isolation properties from Wasm. The sandboxed library code can be safely linked with C applications, enabling a form of zero-cost transition. The zero-cost Wasm SFI system described by this paper was designed and released prior to and independently of WasmBoxC, as the creators of WasmBoxC acknowledge (citation elided for DBR). Moreover, we believe that the theory developed in this paper provides a foundation for analyzing and proving the security of WasmBoxC though such analysis would need to account for possible undefined behavior introduced in compiling to C.

Sledge [GMP+20] describes a Wasm runtime for edge computing, that relies on Wasm

properties to enable efficient isolation of serverless components. However, Sledge focuses on function scheduling including preempting running Wasm programs, so its needs for context saving differ from library sandboxing as contexts must be saved even in the middle of function calls.

SFI Verification. Previous work on SFI (e.g., [MM06, YSD+09, EAV+06, JTA+21]) uses a *verifier* or a theorem prover [ZLDSR11, KSA14] to validate the relevant SFI properties of compiled sandbox code. However, unlike VeriZero, none of these verifiers establish sufficient properties for zero-cost transitions.

Hardware Based Isolation. Hardware features such as memory protection keys [VOED+19, HGJ+19], extended page tables [QCD+17], virtualization instructions [QCD+17, BBM+12], or even dedicated hardware designs [SWS+20] can be used to speed up memory isolation. These works focus on the efficiency of memory isolation as well as switching between protected memory domains; however these approaches also use a single memory region that contain both the stack and heap making them incompatible with zero-cost conditions, i.e. they require heavyweight transitions. `IdealHeavy32` and `IdealHeavy64` in Section 5.4 studies an idealized version of such a scheme.

Capabilities. [Kar89] and [SDB19] look at protecting interacting components on systems that provide hardware-enforced capabilities. [Kar89] specifically looks at how register saving and restoration can be optimized based on different levels of trust between components, however their analysis does not offer formal security guarantees. [SDB19] investigate a calling-convention based on capabilities (à la CHERI [WWN+15]) that allow safe sharing of a stack between distrusting components. Their definition of well-bracketed control flow and local state encapsulation via an overlay inspired our work, and our logical relation is also based on their work. However, their technique does not yet ensure an equivalent notion to our confidentiality property, and further is tied to machine support for hardware capabilities.

Type Safety for Isolation. There has also been work on using strongly-typed languages to

provide similar security benefits. SingularityOS [AFH+06, HL07, FAH+06], explored using Sing# to build an OS with cheap transitions between mutually untrusting processes. Unlike the work on SFI techniques that zero-cost transitions extend, tools like SingularityOS require engineering effort to rewrite unsafe components in new safe languages.

At a lower level, Typed Assembly Language (TAL) [MWCG99, MCGW02, MCG+99] is a type-safe compilation target for high-level type-safe languages. Its type system enables proofs that assembly programs follow calling conventions, and enables an elegant definition of stack safety through polymorphism. Unfortunately, SFI is designed with unsafe code in mind, so cannot generally be compiled to meet TAL's static checks. To handle this our zero-cost and security conditions instead capture the *behavior* that TAL's type system is designed to ensure.

5.7 Chapter acknowledgments

Chapter 5, contains material reprinted from the Symposium on Principles of Programming Languages (POPL) 2021. Kolosick, Matthew; Narayan, Shravan; Johnson, Evan; Watt, Conrad; LeMay, Michael; Garg, Deepak; Jhala, Ranjit; Stefan, Deian. POPL, 2021. The dissertation author was one of the co-authors of this material.

The dissertation author made co-equal contributions to the design of zero-cost transitions and was responsible for implementation and evaluation of zero-cost transitions. Material from the POPL publication pertaining to the formalization of zero-cost transitions are not included in this dissertation chapter.

Conclusion

In this dissertation, we demonstrated that sandboxing allows developers to safely use code from third-parties in their applications. To do this, we systematically tackled the three major issues that have prevented the adoption of sandboxing in applications over the last three decades.

We first addressed the engineering challenges of retrofitting sandboxing onto large existing code bases. With RLBox, we showed that sandboxing can be adopted even in large existing code bases such as the Firefox browser to isolate third-party libraries. RLBox has been deployed in the Firefox browser for over two years at this time.

We next addressed the challenge of ensuring security in software-based sandboxing toolchains. With VeriWasm, we show how we can systematically catch compiler bugs that affect sandboxing. VeriWasm has been used in Fastly's FaaS platform to catch bugs in their sandboxing compilers for over a year at this time. With Swivel, we showed that sandboxing compilers can be made resistant to the Spectre microarchitectural attack. Swivel is the basis for an RFC we are actively developing to harden the sandboxing compilers used in real applications.

Finally, we addressed the performance overheads of sandboxing. With Zero-Cost, we showed how to eliminate overheads in the transitions between applications and sandboxed code. Zero-Cost is being used as part of RLBox's deployment in Firefox and is the key to efficiently sandboxing the font-rendering libraries in Firefox.

Our work demonstrates that sandboxing can indeed live up to the original promise made three decades ago, and will allow us to systematically remove untrusted components from the trusted code base of our applications.

Bibliography

[ABEL09] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[ABG+16] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the Servo Web browser engine using rust. In *ICS 2016: SEIP*. ACM, 2016.

[AFH+06] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory System Performance and Correctness, San Jose, California, USA, October 11, 2006*. ACM, 2006.

[afl13] google/afl. https://github.com/google/AFL, 2013.

[AMD18] AMD. Software techniques for managing speculation on AMD processors. http://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2018.

[AMD19] AMD. Speculation behavior in AMD micro-architectures. https://www.amd.com/system/files/documents/security-whitepaper.pdf, 2019.

[AME+11] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 2011.

[App18] Apple. About speculative execution vulnerabilities in ARM-based and Intel CPUs. https://support.apple.com/en-us/HT208394, 2018.

[ASU86] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 1986.

[ATLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*. ACM, 1996.

[aut16] autocannon: fast HTTP/1.1 benchmarking tool written in Node.js. https://github.com/mcollina/autocannon, 2016.

[AVBOP20] Fritz Alder, Jo Van Bulck, David Oswald, and Frank Piessens. Faulty Point Unit: ABI poisoning attacks on Intel SGX. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 2020.

[BA08] Suhabe Bugrara and Alex Aiken. Verifying the safety of user pointer dereferences. In *S&P*. IEEE, 2008.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. In *CACM*, 2010.

[BBD+19] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. Compiling sandboxes: Formally verified software fault isolation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*. Springer, 2019.

[BBM+12] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*. USENIX, 2012.

[BBZ+19] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *PACT*. IEEE, 2019.

[BCC+10] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA I@A*, 2010.

[BCN+17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow Integrity: Precision, Security, and Performance. *CSUR*, 2017.

[BD18] Alexandre Bartel and John Doe. Twenty years of escaping the Java sandbox. In *Phrack*, 2018.

[BDL+16] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*. Internet Society, 2016.

[BDMT19] Heiko Becker, Eva Darulova, Magnus O Myreen, and Zachary Tatlock. Icing: Supporting fast-math style optimizations in a verified compiler. In *CAV*, 2019.

[bin15] Binaryen WebAssembly toolchain. https://github.com/WebAssembly/binaryen/, 2015.

[BJL18] Frédéric Besson, Thomas Jensen, and Julien Lepiller. Modular software fault isolation as abstract interpretation. In *SAS*, 2018.

[BJRt08] Adam Barth, Collin Jackson, Charles Reis, and the Google Chrome Team. The security architecture of the Chromium browser. Technical report, 2008.

[BLMP13] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SQAS*, 2013.

[BLP16] Sandrine Blazy, Vincent Laporte, and David Pichardie. Verified abstract interpretation techniques for disassembling low-level self-modifying code. In *Journal of Automated Reasoning*, 2016.

[BLP20] Jay Bosamiya, Benjamin Lim, and Bryan Parno. WebAssembly as an intermediate language for provably-safe software sandboxing. PriSC, 2020.

[BMHK08] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*. USENIX, 2008.

[BRN+20] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. Towards a verified range analysis for JavaScript JITs. In *PLDI*, 2020.

[Bro18] Chris Brook. Firefox, Safari, Edge all fall at Pwn2Own 2018. https://digitalguardian.com/blog/firefox-safari-edge-all-fall-pwn2own-2018, 2018.

[BS04] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Security*. USENIX, 2004.

[BSI19] Christian Blichmann, Robert Swiecki, and ISE Sandboxing team. Open-sourcing sandboxed API. https://security.googleblog.com/2019/03/open-sourcing-sandboxed-api.html, 2019.

[Bug15] Bug 1192226 (CVE-2015-4506) vp9 init context buffers. https://bugzilla.mozilla.org/show_bug.cgi?id=1192226, 2015.

[Bug18] Bug 1446062 (CVE-2018-5146) ZDI-CAN-5822 - Mozilla Firefox Audio Driver Out of Bounds. https://bugzilla.mozilla.org/show_bug.cgi?id=1446062, 2018.

[Byt19a] Bytecode Alliance. Sightglass: a benchmark suite and tool to compare different implementations of the same primitives. https://github.com/bytecodealliance/sightglass, 2019.

[Byt19b] Bytecode Alliance. WebAssembly micro runtime. https://github.com/bytecodealliance/wasm-micro-runtime, 2019.

[Byt20a] Bytecode Alliance. Lucet, 2020.

[Byt20b] Bytecode Alliance. Webassembly micro runtime, 2020.

[BZP19] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *S&P*. IEEE, 2019.

[Cam83] Fiona Campbell. The portable UCSD p-system. In *MICROPRO*, 1983.

[Car18] Chandler Carruth. RFC: Speculative load hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP*, 2005.

[CCM+09] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[CDG+20] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *PLDI*. ACM, 2020.

[CGG+19] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*. ACM, 2019.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010.

[Chr19] Security: stack-buffer-overflow in break. https://bugs.chromium.org/p/chromium/issues/detail?id=850350, 2019.

[Chr20] Chromium Team. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety, 2020.

[CLH+15] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *S&P*. IEEE, 2015.

[Con58] Melvin E. Conway. Proposal for an UNCOL. In *CACM*, 1958.

[CRSL16] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[CSA+15] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[CVBS+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *SEC*. USENIX, 2019.

[CVS+15] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *CCS*. ACM, 2015.

[DMCS10] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. PNaCl: Portable native client executables. *Google White Paper*, 2010.

[DPF06] Stefano Di Paola and Giorgio Fedon. Subverting Ajax. Presented at 23C3, 2006.

[DZL15] Liang Deng, Qingkai Zeng, and Yao Liu. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP SEC*. Springer, 2015.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 75–88. USENIX Association, 2006.

[Ena20] Enarx. enarx/enarx Wiki. https://github.com/enarx/enarx/wiki/, 2020.

[ERAGP18] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*. ACM, 2018.

[ES00] Ulfar Erlingsson and Fred B Schneider. SASI enforcement of security policies: A retrospective. In *DISCEX*, 2000.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*. ACM, 2006.

[FC08] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of USENIX ATC 2008*. USENIX, 2008.

[FGBR18] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU. In *S&P*. IEEE, 2018.

[For05] Bryan Ford. VXA: A virtual architecture for durable compressed archives. In *FAST*, volume 5, 2005.

[Fro20] Nathan Froyd. Securing Firefox with WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, 2020.

[FSA97] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, 1997.

[GAS+17] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. Lazarus: Practical side-channel resilient kernel-space randomization. In *RAID*. Springer, 2017.

[GCF15] Nuwan Goonasekera, William Caelli, and Colin Fidge. LibVM: an architecture for shared library sandboxing. *Software: Practice and Experience*, 45(12), 2015.

[Geo] Georg Koppen. Use RLBox for sandboxing third-party libraries. https://trac.torproject.org/projects/tor/ticket/32379.

[GKM+20] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled detection of speculative information flows. In *S&P*. IEEE, 2020.

[GLM12] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. In *Queue*, 2012.

[GMP+20] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: a serverless-first, light-weight wasm runtime for the edge. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ACM, 2020.

[Goh18] Dan Gohman. Cranelift in SpiderMonkey. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/spidermonkey.md, 2018.

[Goo17] Google Chrome Team. (P)NaCl Deprecation Announcements. https://developer.chrome.com/native-client/migration#p-nacl-deprecation-announcements, 2017.

[Goo18] Google Chrome Team. Site isolation. https://www.chromium.org/Home/chromium-security/site-isolation, 2018.

[Goo20] Google. Safeside. https://github.com/google/safeside, 2020.

[Goo21a] Introducing the in-the-wild series:. https://googleprojectzero.blogspot.com/2021/01/introducing-in-wild-series.html, January 2021.

[Goo21b] October 2020 0-day discovery. https://googleprojectzero.blogspot.com/2021/03/in-wild-series-october-2020-0-day.html, March 2021.

[GSS+15] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Memory-safe execution of C on a Java VM. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2015.

[GTK11] Chris Grier, Shuo Tang, and Samuel T. King. Designing and implementing the OP and OP2 Web browsers. *ACM Transactions on the Web*, 5(2), 2011.

[GWA+15] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *CCS*. ACM, 2015.

[Han19a] L. Hansen. Mark the jump_table_entry instruction as loading. https://github.com/bytecodealliance/cranelift/pull/805, 2019.

[Han19b] Lars T Hansen. Cranelift: Performance parity with Baldr on x86-64. https://bugzilla.mozilla.org/show_bug.cgi?id=1539399, 2019.

[HBG+09] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Fault isolation for device drivers. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. IEEE Computer Society, 2009.

[HCLS15] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *ESORICS*, volume 9326 of *LNCS*. Springer, 2015.

[Hen06] John L Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

[HGJ+19] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019.

[HL07] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2), 2007.

[HNL+13] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *CGO*, 2013.

[HNTC+12] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd my gadgets go? In *S&P*. IEEE, 2012.

[Hol21] Bobby 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/, November 2021.

[Hor18] Jann Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[HR19] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *IoTDI*. ACM, 2019.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *PLDI*. ACM, 2017.

[Int17] Intel. CET Linux kernel implementation. https://github.com/hjl-tools/fedora, 2017.

[Int18a] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[Int18b] Intel. Speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[Int18c] Intel. Speculative store bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass, 2018.

[Int19] Intel. Deep dive: Intel analysis of microarchitectural data sampling. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-analysis-microarchitectural-data-sampling#SMT-mitigations, 2019.

[Int20a] Intel® 64 and IA-32 architectures software developer's manual, 2020.

[Int20b] Intel® C++ Compiler 19.1 Developer Guide and Reference, 2020.

[Int20c] Intel. Side channel mitigation by product CPU model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html, 2020.

[JAS+20] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J Peter Brady, Sergey Bratus, and Sean W Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *HotSos*, 2020.

[JCH+16] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. The "Web/local" boundary is fuzzy: A security study of Chrome's process-based sandboxing. In *CCS*. ACM, 2016.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL*, 2015.

[Joh] Evan Johnson. Integrate VeriWasm. https://github.com/bytecodealliance/lucet/pull/658.

[Joh21] Evan Johnson. Update veriwasm version, 2021.

[JPBG19] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *ATC*. USENIX, 2019.

[JTA+20] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Veriwasm verifier. https://veriwasm.programming.systems, 2020.

[JTA+21] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Доверяй, но проверяй: SFI safety for native-compiled Wasm. In *NDSS*. Internet Society, 2021.

[jvm19] Java platform, standard edition: Java virtual machine guide. Technical report, 2019.

[JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Security*. USENIX, 2004.

[Kar89] Paul A. Karger. Using registers to optimize cross-domain call performance. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, page 194–204, New York, NY, USA, 1989. Association for Computing Machinery.

[KD09] J Kroll and Drew Dean. BakerSFIeld: bringing software fault isolation to x64. *SRI International, Tech. Rep.*, 2009.

[ker19] kernel.org. TAA: TSX asynchronous abort. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/tsx_async_abort.html, 2019.

[ker20] kernel.org. Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html, 2020.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*. IEEE, 2019.

[Kil03] Douglas Kilpatrick. Privman: A library for partitioning applications. In *ATC*. USENIX, 2003.

[KKS+19] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *DAC*. IEEE, 2019.

[KKSAG18] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *WOOT*. USENIX, 2018.

[KLA+18] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO*. IEEE, 2018.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *POPL*, 2014.

[KS16] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *Security*. USENIX, 2016.

[KSA14] Joshua A Kroll, Gordon Stewart, and Andrew W Appel. Portable software fault isolation. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 18–32. IEEE, 2014.

[KSK+20] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SPECCFI: Mitigating Spectre attacks using CFI informed speculation. In *S&P*. IEEE, 2020.

[KSP+14] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *OSDI*. USENIX, 2014.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[Kwo17] Gary Kwong. JavaScript fuzzing in Mozilla, 2017. Presented at COSCUP 2017. https://nth10sd.github.io/js-fuzzing-in-mozilla/, 2017.

[Lar18] Michael Larabel. Benchmarking the performance impact of Speculative Store Bypass Disable for Spectre V4 on Intel Core i7. https://www.phoronix.com/scan.php?page=article&item=intel-spectre-ssbd&num=1, 2018.

[LBK+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A formally verified optimizing compiler. In *ERTS*, 2016.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, 2006.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. In *CACM*, 2009.

[LKP20] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In *SEC*. USENIX, 2020.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, 2003.

[LMG+18] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System v application binary interfaceamd64 architecture processor supplement(with lp64 and ilp32 programming models). Technical report, 2018.

[LMNR15] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, 2015.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[LSG+17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *SEC*. USENIX, 2017.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *SEC*. USENIX, 2018.

[LSL+15] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *CCS*. ACM, 2015.

[LSS15] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*, 2015.

[LSW95] Steve Lucco, Oliver Sharp, and Robert Wahbe. Omniware: A universal substrate for web programming. In *WWW*, 1995.

[LTJ17] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting general pointers in automatic program partitioning. In *CCS*. ACM, 2017.

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

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Addison-Wesley Professional, 2014.

[LZH+19] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against Spectre attacks. In *HPCA*. IEEE, 2019.

[Mac92] Stavros Macrakis. The structure of ANDF: Principles and examples. *Open Software Foundation*, 1992.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[MCGW02] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-Based Typed Assembly Language. *Journal of Functional Programming*, 12:43–88, January 2002. Publisher: Cambridge University Press.

[McM20] Tyler McMullen. Lucet: A compiler and runtime for high-concurrency low-latency sandboxing. In *PriSC*, 2020.

[MG01] Erik Meijer and John Gough. Technical overview of the common language runtime. http://research.microsoft.com/~emeijer/papers/CLR.pdf, 2001.

[Mic20a] Microsoft. More Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/more-spectre-mitigations-in-msvc/, 2020.

[Mic20b] Microsoft Flight Simulator Team. August 20th, 2020 development update. https://www.flightsimulator.com/august-20th-2020-development-update/, 2020.

[Mil19] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat, 2019.

[MLSS20] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *SEC*. USENIX, 2020.

[MM06] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Security*. USENIX, 2006.

[MMT10] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 125–140. IEEE Computer Society, 2010.

[mod11] Markdown filter module for apache httpd server. https://github.com/hamano/apache-mod-markdown, 2011.

[Moz18a] Performance sheriffing/Talos. https://wiki.mozilla.org/Performance_sheriffing/Talos, 2018.

[Moz18b] Security/sandbox. https://wiki.mozilla.org/Security/Sandbox, 2018.

[Moz19] Project Fission. https://wiki.mozilla.org/Project_Fission, 2019.

[Moz21] Mozilla. Firefox Public Data Report. https://data.firefox.com/dashboard/hardware, 2021.

[MR18] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*. ACM, 2018.

[MSL+08] M.S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf, June 2008.

[MSS20] Kathleen Metrick, Jared Semrau, and Shambavi Sadayappan. Think fast: Time between disclosure, patch release and vulnerability exploitation – intelligence for vulnerability management, part two. https://www.fireeye.com/blog/threat-research/2020/04/time-between-disclosure-patch-release-and-vulnerability-exploitation.html, August 2020.

[MST+19] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178*, 2019.

[MTT+12] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *PLDI*. ACM, 2012.

[MWC10] Adrian Mettler, David A Wagner, and Tyler Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21:527–568, May 1999.

[MZTG16] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *PLDI*, 2016.

[Nac10] Issue 775: Uninitialized sendmsg syscall arguments in sel_ldr. https://bugs.chromium.org/p/nativeclient/issues/detail?id=775, 2010.

[Nac11a] Issue 1607: Signal handling change allows inner sandbox escape on x86-32 linux in chrome. https://bugs.chromium.org/p/nativeclient/issues/detail?id=1607, 2011.

[Nac11b] Issue 1633: Inner sandbox escape on 64-bit windows via kiuserexceptiondispatcher. https://bugs.chromium.org/p/nativeclient/issues/detail?id=1633, 2011.

[Nac12] Issue 2919: Security: Naclswitch() leaks naclthreadcontext pointer to x86-32 untrusted code. https://bugs.chromium.org/p/nativeclient/issues/detail?id=2919, 2012.

[Nat09] Native Client team. Native Client security contest archive. https://developers.chrome.com/native-client/community/security-contest/index, 2009.

[NDG+20] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *SEC*. USENIX, 2020.

[Nec97] George C Necula. Proof-carrying code. In *POPL*, 1997.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[NGL+19] Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. Gobi: Webassembly as a practical path to library sandboxing, 2019.

[Nic19] Shaun Nichols. It's 2019, and a PNG file can pwn your Android smartphone or tablet: Patch me if you can. https://www.theregister.co.uk/2019/02/07/android_january_patches/, 2019.

[nod10] bcrypt for nodejs. https://github.com/kelektiv/node.bcrypt.js, 2010.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[NT14] Ben Niu and Gang Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014.

[NZMZ09] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[NZMZ10] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[OTR+18] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv:1805.08506*, 2018.

[Pat19] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019.

[PFH03] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Security*. USENIX, 2003.

[PG11] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2011. Association for Computing Machinery.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[QCD+17] Weizhong Qiang, Yong Cao, Weiqi Dai, Deqing Zou, Hai Jin, and Benxi Liu. Libsec: A hardware virtualization-based isolation for shared library. In *Proceedings of HPCC 2017*. IEEE, 2017.

[RCC+12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, 2012.

[RG09] Charles Reis and Steven D. Gribble. Isolating Web programs in modern browser architectures. In *EuroSys*. ACM, 2009.

[RL10] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *CC/ETAPS*, 2010.

[RMO19] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *SEC*. USENIX, 2019.

[roc20] Rocket. https://rocket.rs/, 2020.

[RRW05] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *TECS*, 2005.

[Ryd20] Henrik Rydgard. Windows (Fastcall) calling convention: Callee-saved XMM (FP) registers are not actually saved. https://github.com/bytecodealliance/wasmtime/issues/1177, 2020.

[Sam75] Hanan Samet. *Automatically proving the correctness of translations involving optimized code*. PhD thesis, Stanford University, 1975.

[Sau12] Georg Sauthoff. cgmemtime. https://github.com/gsauthof/cgmemtime, 2012.

[SBCA15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *POPL*, 2015.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *TOCS*, 1997.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[SDB19] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, January 2019.

[Sea13] Mark Seaborn. Sandboxing libraries in Chrome using SFI: zlib proof-of-concept. https://docs.google.com/presentation/d/1RD3bxsBfTZOIfrlq7HzGMsygPHgb61A1eTdelIYOurs/, 2013.

[SESS96] Margo I Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996*. ACM, 1996.

[SGDL19] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. In *POPL*. ACM, 2019.

[SGL+18] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *AsiaCCS*. ACM, 2018.

[SGS19] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *HASP*, 2019.

[SHF16] Karla Saur, Michael Hicks, and Jeffrey S Foster. C-strider: type-aware heap traversal for C. *Software: Practice and Experience*, 46(6), 2016.

[Shr] Shravan Narayan. Use Wasm sandboxed libraries in Firefox to reduce attack surface. https://bugzilla.mozilla.org/show_bug.cgi?id=1562797.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *WBIA*, 2009.

[Sin10] Alexey Sintsov. JIT-spray Attacks & Advanced Shellcode. In *HITBSecConf Amsterdam*, 2010.

[SKMT13] Ralf Sasse, Samuel T King, José Meseguer, and Shuo Tang. IBOS: A correct-by-construction modular browser. In *FACS*, volume 7684. Springer, 2013.

[SLC+20] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating Spectre. In *NDSS*. Internet Society, 2020.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[SLR+19] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *IEEE Security and Privacy 2019*, 2019.

[SLS16] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *OOPSLA*, 2016.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[SMB+10] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *Security*. USENIX, 2010.

[SMGM17] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *FC*. Springer, 2017.

[Sne18] Lachlan Sneff. Nebulet. https://github.com/nebulet/nebulet, 2018.

[SP20] Simon Shillaker and Peter Pietzuch. FAASM: Lightweight isolation for efficient stateful serverless computing. In *ATC*. USENIX, 2020.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS*. ACM, 2004.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[SQ19] Gururaj Saileshwar and Moinuddin K Qureshi. CleanupSpec: An "undo" approach to safe speculation. In *MICRO*. IEEE, 2019.

[SS98] Christopher Small and Margo Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE concurrency*, 6(3), 1998.

[ST14] Mengtao Sun and Gang Tan. NativeGuard: Protecting Android applications from third-party native libraries. In *Proceedings of WiSec 2014*. ACM, 2014.

[Ste17] Stefan Marsiske, Pierre Pronchery, Marcus Bointon. Penetration Test Report: Graphite font system. https://wiki.mozilla.org/images/9/98/Graphite-report.pdf, 2017.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In *CAV*, 2011.

[STM10] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the JVM. In *Proceedings of CCS 2010*. ACM, 2010.

[Str13] Bjarne Stroustrup. Exception handling (and RAII). In *The C++ Programming Language*, chapter 13. Addison-Wesley, 4th edition, 2013.

[SWG+16] Rui Shu, Peipei Wang, Sigmund A. Gorski, III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Computing Surveys*, 49(3), 2016.

[SWS+20] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[SZOC19] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with Venkman. *arXiv:1903.10651*, 2019.

[Tan17] Gang Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3), 2017.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[The18] The LLVM Foundation. Automatic variable initialization, 2018.

[the19] A brief Theora and Vorbis encoding guide. https://trac.ffmpeg.org/wiki/TheoraVorbisEncodingGuide, 2019.

[The21a] The LLVM Foundation. Control Flow Integrity, Clang 12 documentation, 2021.

[The21b] The LLVM Foundation. Safestack, Clang 12 documentation, 2021.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *POPL*, 2008.

[TL09] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *PLDI*, 2009.

[TL10] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.

[TLR20] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *CGO*, 2020.

[TRC+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium*, pages 941–955, 2014.

[Tsa21] Alex Tsariounov. Shielding linux resources—introduction, 2021.

[Tur18] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[TVT19] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*. ACM, 2019.

[Var17] Kenton Varda. Introducing Cloudflare Workers: Run JavaScript service workers at the edge. https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[Var18] K. Varda. WebAssembly on Cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, 2018.

[VBMS+20] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *S&P*. IEEE, 2020.

[VDG+21] Marco Vassena, Craig Disselkoen, Klaus V Gleissenthall, Sunjay Cauilgi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks with Blade. In *POPL*. ACM, 2021.

[VKM19] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *S&P*. IEEE, 2019.

[VOED+19] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Security*. USENIX, 2019.

[vp919] VP9 bitrate modes in detail. https://developers.google.com/media/vp9/bitrate-modes/, 2019.

[vSMÖ+19] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*. IEEE, 2019.

[vVZN+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: a verified compiler for relaxed-memory concurrency. In *ACM*, 2013.

[VYY10] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[WCG+19] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against Spectre attacks via binary analysis. *TSE*, 2019.

[Web21] WebAssembly Community Group. Exception handling, 2021.

[WGM+09] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle web browser. In *Security*. USENIX, 2009.

[WKL+17] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Security*. USENIX, 2017.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP*. ACM, 1993.

[WNL+19] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *MICRO*. IEEE, 2019.

[WRPP19] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Weakening webassembly. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[WSYL12] Yongzheng Wu, Sai Sathyanarayan, Roland H.C. Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *ESORICS*, volume 7459 of *LNCS*. Springer, 2012.

[WWN+15] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[WZH+11] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *ECML PKDD*, 2011.

[XQL+18] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *S&P*. IEEE, 2018.

[yax20] Yaxpeax-x86 disassembler. https://github.com/iximeow/yaxpeax-x86, 2020.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[YCS+18] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *MICRO*. IEEE, 2018.

[YJO+17] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In *USENIX Sec*, 2017.

[YLX+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Sec*, 2018.

[YSD+09] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*. IEEE, 2009.

[Yve16] Yves Younan. Vulnerability Spotlight: Libgraphite Font Processing Vulnerabilities. https://blog.talosintelligence.com/2016/02/vulnerability-spotlight-libgraphite.html, 2016.

[YYK+19] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO*. IEEE, 2019.

[Zak20] Alon Zakai. Wasmboxc: Simple, easy, and fast vm-less sandboxing. https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html, 2020.

[zer19] Optimized transitions for Lucet compiler. https://github.com/bytecodealliance/cranelift/issues/1083, 2019.

[ZKE20] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*. ACM, 2020.

[ZLDSR11] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. ARMor: fully verified software fault isolation. In *EMSOFT*, 2011.

[ZS13] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Sec*, 2013.

[ZTE13] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Sec*, 2013.

[ZTM11] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[ZWCW14] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014.