Thesis Proposal

Jay Bosamiya

1 Introduction

There has been a constant tug of war between security and software development. Market considerations require development of new features at a break-neck pace, in addition to needing software to be fast (improved responsiveness, reduced server costs, improved debug time, and more). Software *engineering* thus attempts, like any other engineering field, to carefully balance various tradeoffs to optimize for various objectives. Unfortunately, an objective that seems to have been in constant conflict with the others is security. While many developers (and more so, users) would like their software to be secure, often security comes with an apology for at least one of the other objectives (often multiple at once), and thus becomes under-prioritized. To be adopted, advances in security thus need to not only improve the state-of-the-art in security, but also focus on other practical considerations that have historically inhibited widespread deployment, and indeed prevented building secure software from being the natural default choice.

In this thesis, we focus on questions that arise from the interaction of security and the development of practical software systems. In particular, we explore if one can achieve security *without* compromising on the aforementioned other axes of development velocity, maintainability, runtime performance, and more. The goal of this thesis is to make a large step towards being able to answer with a strong affirmative "yes". Specifically, we propose that:

> Security objectives are achievable without apology, through the use of principled approaches and formalism.

As evidence towards this thesis statement, we show that successfully applying principled approaches and formalism removes the need for apology, across a collection of different kinds of software systems, described below.

We begin, in Section 2, by focusing on high-performance code that is at the heart of securing the world's internet traffic—cryptographic primitives. In particular, AES-GCM secures 98.9% of the world's internet traffic [38]. This has been driven by increased adoption of HTTPS, a major success story for security. However, with encrypted connections come increased costs. An obvious cost is increased server CPU utilization, potentially requiring deploying more servers than needed before. In fact, for some websites, encryption and decryption can be the biggest bottleneck for user-perceived latency. This naturally leads to a focus on making the implementation of this crucial primitive as fast as possible. Using hardware-accelerated instructions (AES-NI), cryptography providers such as OpenSSL have written some extremely optimized code. But with this comes a cost—complexity. The

most optimized AES-GCM implementation in OpenSSL consists of 724 SLOC lines of hand-written x86-64 assembly, implemented by using over 950 SLOC of Perl as a macro expander, via string manipulation. While incredibly fast encryption is great, familiarity with security should also set our radars tingling. Complexity is the enemy. Beware complexity, for here lie dragons! Thankfully, we have a powerful sword to tame this beast—formal verification. Indeed, by proving the semantic equivalence of this mass of ugly assembly to a simpler specification, we can reduce the complexity without sacrificing performance [48]. This too comes at a cost—inelegance; the machine-checked proofs to prove this semantic equivalence are extremely subtle, detailed, and ugly, needing to worry about extremely low-level details, thereby requiring tremendous developer effort to write. Worse, they may be fragile, even under small changes to the code, which makes it harder for people to further optimize the code. Considering all these costs, we introduce a technique that allows one to write elegant "beautiful" proofs about this "ugly" assembly code, reducing proof burden both for writing the proofs, as well as maintaining them. Additionally, we use this technique to push beyond unverified approaches, using the strong safety net of formal verification, to optimize code upto 27% beyond OpenSSL's fastest implementation. In short, we show that for certain ultra-high-performance systems, formal techniques can be used to provide high-assurance safety, without apologizing for development velocity, or runtime performance.

Next, in Section 3, we turn to focus on securing the execution of untrusted code. Practical considerations have us regularly execute untrusted code—software plugins, 3$^{\text{rd}}$-party libraries, even client-side code run when browsing the Web. Furthermore, new contexts such as dynamic content delivery networks (CDNs), edge computing, and smart contracts have created additional motivation to run untrusted code that could potentially harm its execution environment. This should set us on alert. Beware untrusted code, for here lie dragons! A naïve solution is to not have any untrusted code, but this comes with additional development costs (and do we *really* need yet another poorly implemented XML or JPEG parser?). A classic solution is to use software isolation, or sandboxing. However, despite decades of prior work on software fault isolation (SFI), there has been little deployment, due to technical and practical hurdles. Instead, we identify WebAssembly (Wasm) [22], originally intended for the Web, as an attractive narrow waist for software sandboxing, since it promises safety *and* performance. Existing (untrusted) compilers can compile *to* Wasm, while an execution engine for Wasm can provide sandboxing. However, Wasm's safety guarantees are only as strong as the implementation that enforces them. Looking at the design space for implementing Wasm, we noticed a lack of high-security, high-performance, and portable execution engines. In particular, interpreters often apologize for performance, and compilers often apologize for either security, or portability, or both. Thus, we implemented two provably-safe compilers from WebAssembly. One, vWasm, is the first formally-verified sandboxing compiler; the other, rWasm, is the first provably-safe sandboxing compiler that meets or exceeds the performance of other compilers, including ones focused primarily on performance and not necessarily safety/security. Additionally, rWasm elides the need for the tedium of formal proofs. In short, we show that safe execution of arbitrary code can be achieved, through principled and formal approaches, without compromising on either development velocity, runtime performance, or portability.

Expanding further in Section 4, practically, one might additionally want to be pre-emptively defensive against one's own (i.e., 1$^{\text{st}}$ party) code, in addition to 3$^{\text{rd}}$ party code. In such scenarios, a naïve approach is to treat all code in the project as potentially adversarial, but this often requires an apology in terms of performance (not to mention the psychological burden of treating *yourself* as the enemy). Instead, we recognize that a higher degree of assurance can be provided by utilizing an agile

stance on security enforcement. In particular, the strength of the enforcement (and relevant potential performance tradeoff) can be modulated depending on contextual or environmental circumstances. For example, libraries with low testing-coverage, or particularly security-sensitive end-users can opt in to stronger enforcement while most users can benefit from greater performance. To support agile enforcement without increasing development burden, we propose MSWasm—an intra-module memory-safety-aware extension to WebAssembly. Our compiler from C to MSWasm requires no changes to the C code, thereby alleviating developer burden. Next, we extend rWasm to support MSWasm as input, and provide multiple backends that each provide different degrees of assurance for the code, without needing to change the code, or even the MSWasm module itself. Based upon real-world context, for example if a vulnerability has been discovered but no patch has been applied or written yet, an engineer can switch the enforcement level for that single module to better protect against possible attacks. Additionally, as new alternative software and hardware enforcement mechanisms are released, they can be enabled at very low development cost—just a small change to the rWasm backend, allowing for easy and quick deployment. In short, with the principled design of agile safety enforcement, we show how we can adapt to the ever-changing threat landscape without apologizing for development velocity and (for the most part) performance.

Switching gears to untrusted data in Section 5, we focus on safe parsing and serialization. High-level well-typed data is great to work with. Yet eventually it must interact with the real world, either other programs, or even other machines. Thus, we must deal with on-wire/flat representations that can be parsed to or serialized from high-level data. Unfortunately, parsers are a notoriously common source of bugs. Beware untrusted data, for here lie dragons! Additionally parsers are generally a burden to write, and require additional maintenance to keep in sync with a serializer that matches the parser. Formally verifying parsers and serializers has been done in the past [49, 53], but has often come with high complexity and limited extensibility. Recognizing that there are two kinds of data formats in play, we reduce complexity by splitting the task in two: handling *intrinsic*, and *extrinsic* data formats. The former focuses on high-level data, where the actual on-wire format is flexible; the latter focuses on formats where the on-wire format is pre-specified and inflexible (such as communicating with other servers/clients). We are in the process of implementing formally-verified parsers and serializers for these two kinds of data formats, such that each can provide a good user experience for their particular domain, while guaranteeing important safety and correctness properties. While this is ongoing work, we already have promising results showing how these different serialization libraries integrate well with other verified projects. In short, through a principled approach for automatic generation of parsers and serializers, we can safely deal with arbitrary untrusted data without apologizing for developer time, approachability, expressivity, or runtime performance.

Finally, we focus on systems with unusable or unavailable source code in Section 6. When securing software systems, one must often also deal with software where we must rely only on the executable/binary code. A common situation for this is legacy software. Understanding what such source-unavailable software does is an important first step towards securing it (which can either be through a rewrite, or more often, binary patching). Unfortunately, despite decades of advancements in the science and art of decompilation (i.e., reverse compilation), the quality of decompiled output still leaves much to be desired. In particular, many published prior works show increasingly sophisticated and complicated techniques to recover source-level type information from binary executables, and yet, the state-of-the-art tools used by practitioners in the field barely recover any types automatically, and require a large amount of human annotation. We have analyzed this

gap to understand why it exists, and what can be done to solve it. We observe that there is a mismatch of goals between the practitioners and the academic work; the practitioner wants to see what the software is really *doing* while the academic work tries to *recover* type information, in an attempt to match "ground truth". Worse, we also identified that there can be no (traditional) ground truth for the problem at hand; thus prior techniques were optimizing towards a goal under a fundamentally flawed assumption. With the need to identify a better evaluation strategy, we have focused on building an evaluation that can better capture the practitioner's requirements from the decompilers. Additionally, to improve upon the state of the art, we have been building a prototype type-reconstruction system to achieve high quality source-level types. For this, we identified that structural types capture the essence of the computation better than the nominal types used by prior works, and thus have been building our tool around this idea. While this is ongoing work, we are already seeing promising results showing better output than the state-of-the-art. In short, through a principled approach towards understanding decompilation, we have produced a better formalism that need not compromise on either elegance or output quality, in order to advance software comprehension. We discuss this further in Section 6.

To summarize the proposed thesis, principled approaches and formalism help alleviate situations where security was previously believed to be in conflict with other goals, including development velocity, software performance, portability, or elegance. Sections 2 to 4 describe prior work towards this thesis, and Sections 5 and 6 describe ongoing work. We conclude with a proposed timeline in Section 7.

2 Beautiful Proofs for Ugly Code

Status: Completed [6]

Hand-optimized assembly language code is often difficult to formally verify. In this section, we describe our work to combine Hoare logic with verified code transformations to make it easier to verify such code. This approach greatly simplifies existing proofs of highly optimized OpenSSL-based AES-GCM cryptographic code. Furthermore, applying various verified transformations to the AES-GCM code enables additional platform-specific performance improvements.

Some of the most important code in the world is also some of the ugliest. The most commonly used implementations of cryptographic algorithms are heavily optimized, typically employing hand-crafted assembly language for maximum performance. For example, OpenSSL's implementation of AES-GCM, the cryptographic algorithm used for 98.9% of secure web traffic [38], contains thousands of lines of hand-optimized x86-64 assembly language code, implemented by using Perl as a macro expander, via string manipulation. The optimizations unroll loops, prefetch data from memory, carefully hand-schedule instructions, and interleave otherwise unrelated instructions in an effort to expose parallelism and keep the processor's functional units busy (Figure 1a shows a representative snippet, using different colors for conceptually different operations). The resulting code is extremely fast, but difficult to understand, maintain, and verify.

Recent work on EverCrypt [48] used Hoare logic to verify a variant of OpenSSL's AES-GCM x86-64 code. Hoare logic is a natural way to express the verification of well-structured programs. Unfortunately, the optimizations in OpenSSL's AES-GCM code obscure the natural structure of the underlying AES-GCM algorithm, making Hoare logic awkward to use directly on the optimized code.

```
vpclmulqdq      \$0x01,$Hkey,$Ii,$T2
lea             ($in0,%r12),$in0
vaesenc         $rndkey,$inout0,$inout0
vpxor           16+8(%rsp),$Xi,$Xi
vpclmulqdq      \$0x11,$Hkey,$Ii,$Hkey
vmovdqu         0x40+8(%rsp),$Ii
vaesenc         $rndkey,$inout1,$inout1
movbe           0x58($in0),%r13
vaesenc         $rndkey,$inout2,$inout2
movbe           0x50($in0),%r12
vaesenc         $rndkey,$inout3,$inout3
mov             %r13,0x20+8(%rsp)
vaesenc         $rndkey,$inout4,$inout4
mov             %r12,0x28+8(%rsp)
vmovdqu         0x30—0x20($Xip),$Z1
vaesenc         $rndkey,$inout5,$inout5
```

(a) **Representative Snippet of OpenSSL**

```
lea             ($in0,%r12),$in0
vpclmulqdq      \$0x01,$Hkey,$Ii,$T2
vpclmulqdq      \$0x11,$Hkey,$Ii,$Hkey
vpxor           16+8(%rsp),$Xi,$Xi
vmovdqu         0x40+8(%rsp),$Ii
vmovdqu         0x30—0x20($Xip),$Z1
movbe           0x58($in0),%r13
movbe           0x50($in0),%r12
mov             %r13,0x20+8(%rsp)
mov             %r12,0x28+8(%rsp)
vaesenc         $rndkey,$inout0,$inout0
vaesenc         $rndkey,$inout1,$inout1
vaesenc         $rndkey,$inout2,$inout2
vaesenc         $rndkey,$inout3,$inout3
vaesenc         $rndkey,$inout4,$inout4
vaesenc         $rndkey,$inout5,$inout5
```

(b) **A Modular Version of the Code**

Figure 1: *Representative snippets of AES-GCM code. We write proofs about the cleaner, more modular version of the AES-GCM code (b) and then use verified code transformers to connect these proofs to the original OpenSSL code (a). AES operations are highlighted in blue, GCM in green, prefetching and processing of input data in red, and loop control checks in yellow.*

In particular, to automate the proofs, it helps to keep code units relatively small, since that keeps the proof "debug" cycle tolerable for developers. However, the interleaving of unrelated instructions makes it difficult to modularly decompose the code into smaller units with natural preconditions and postconditions. As a result, the preconditions and postconditions describe situations where natural invariants do not yet hold or have already been broken. Worse, each repeated section of code generated from loop unrolling has to be verified separately, because the instruction scheduling and interleaving cause each section to contain slightly different code with slightly different preconditions and postconditions. The ugly code leads to ugly proofs and duplicated effort.

This creates a stark trade-off. On one hand, the performance gains from carefully optimized code are enormous and valuable: the verified code based on OpenSSL's optimized code runs 6× faster than earlier verified code written in a simpler, easier-to-verify style [18]. On the other hand, the effort involved in verifying optimized code may dissuade authors of cryptographic code from attempting any formal verification.

We argue that the trade-off is not as stark as it may seem at first glance:

- First, we demonstrate how to use verified transformers to recover the elegance of Hoare logic. In this approach, the programmer uses Hoare logic to verify a clean, modular version of the code (Figure 1b). In addition, the programmer writes (but does not directly verify) the optimized, non-modular version of the code (Figure 1a). Our verified transformation tool then attempts to automatically discover the relationship between the clean and optimized

versions and prove their equivalence. This proves that the properties established via Hoare logic for the clean code apply to the optimized code.

- Second, we manually create a clean, modular version of EverCrypt's AES-GCM code and measure its performance. To our surprise, on some CPUs, the clean code actually runs slightly *faster* on average than the original EverCrypt code. In other words, not all of OpenSSL's optimizations are equally necessary to achieving its fast performance, and the optimization that causes the most trouble for EverCrypt's verification does not appear to pay off consistently.
- Third, inspired by the observed performance difference between the clean code and EverCrypt code, we investigate the performance of alternate interleavings of the assembly language instructions for various x86-64 processor models. We develop a tool that automatically finds interleavings that are faster than both the EverCrypt code and the clean code, and we use our verified transformation tool to verify the correctness of these new interleavings. Hence, verified transformers support automated development of hyper-targeted optimized implementations while still allowing the developer to write beautiful, Hoare-style proofs.

In summary, we show that for even for ultra-high-performance software that require ugly assembly, a principled approach to discover equivalence for code can recover elegance in formal techniques. This provides high-assurance safety, without apologizing for developement velocity, maintainability, or runtime performance. Indeed, it even safely unlocks greater performance than previously achieved by unverified approaches.

3 Provably-Safe Software Sandboxing

Status: Completed [7]

Lightweight, safe execution of untrusted code is valuable in many contexts, including software plugins, third-party libraries, or even client-side code run when browsing the Web. New contexts such as dynamic content delivery networks (CDNs), edge computing, and smart contracts have created additional motivation to run untrusted code that could potentially harm its execution environment. Software sandboxing (via Software Fault Isolation, aka SFI [55]) is a well-studied technique, with a long and rich history [15, 34, 60, 51, 37, 25, 41, 23], to provide this crucial primitive. It limits the effects of bugs to the buggy code itself, confining any bug's impact within a user-defined boundary, typically within a module or library. Multiple such boundaries can be introduced within the same OS-level process, as shown in Figure 2. Nevertheless, previous efforts to deploy it in production have failed, due to technical and marketplace hurdles.

On the Web, after failed attempts with Java, ActiveX, Flash, NaCl [60], and Asm.js, a new contender for fast code execution was born—WebAssembly [22]. With lightweight, safe, portable, and fast code execution as its goals, WebAssembly (or Wasm) has rapidly become a popular compilation target for client-side code execution on the Web. Designed with sandboxing in mind, it has clean, succinct, and well-defined semantics, which, along with its portability and speed, has made it appealing for use in non-Web contexts too [57, 46, 41, 19, 16]. With the standardization of the WebAssembly Systems Interface (WASI) [57], it even exposes a POSIX-like API for programs to interact with their environment in a controlled manner. This makes it an attractive compilation target for both Web and non-Web contexts, and compilers for most popular languages, such as C,

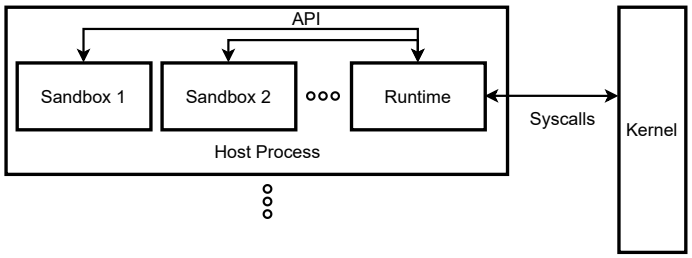

Figure 2: *SFI-based intra-process sandboxing. In practice, the Host Process might be a video player, and each sandbox might contain a different codec or extension.*

C++, Rust, Java, Go, C#, PHP, Python, TypeScript, Zig, and Kotlin, now support it as a target.

As a result, an implementation of Wasm can provide strong guarantees about the safe execution of a large variety of languages on a large number of platforms, making it an attractive narrow waist for sandboxing.

However, WebAssembly's sandboxing guarantees hold only at the specification level; real Wasm implementations can, and do, have bugs. These bugs can completely compromise all the guarantees provided by the specification. A plausible explanation for such disastrous sandbox-compromising bugs, even in code designed with sandboxing as an explicit focus, is that the correct, let alone secure, implementation of high-performance compilers is difficult and remains an active area of research, despite decades of work.

In reviewing the design space for executing Wasm code, we identified a lack of high-security, high-performance, and portable execution engines, despite some engines attempting to achieve these goals. In particular, interpreters often apologize for performance, and compilers often apologize for either security, or portability, or both. Hence, we propose and explore two techniques, with varying performance and development complexity, which guarantee safe sandboxing using provably safe compilers. Along the way, we demonstrate that one can have safety without apologizing for execution performance.

We implement the first of our techniques as a compiler called vWasm, which utilizes formal methods to mathematically prove that the compiled Wasm code can only interact with its host environment via an explicitly provided API, hence ruling out problems where the compiled code, say, reads/writes to prohibited host-memory locations, or jumps to prohibited host code. We accomplish this by writing a machine-checked formal proof about vWasm's implementation. In particular, the executable code produced by the implementation is formally guaranteed to stay within the confines of the sandbox provided to it. Note that this differs from traditional compiler correctness (in the vein of CompCert [30]), which guarantees that the output program matches the input. Indeed, the two properties are orthogonal, and thus we focus on provable sandboxing. To our knowledge, vWasm is the first formally verified implementation of a multi-lingual sandboxing compiler, since it can sandbox any of the many languages with an existing Wasm backend. This complements earlier work [25] that verifiably sandboxed Cminor, one of CompCert's intermediate languages, for CompCert's three backends.

Our second technique, implemented as a compiler called rWasm, takes a different approach

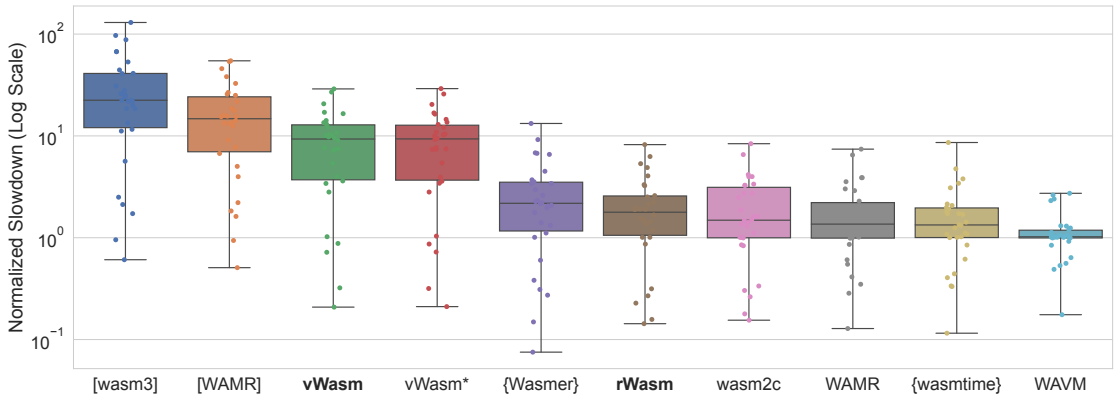

Figure 3: *Mean execution time of PolyBench-C benchmarks across the Wasm runtimes, normalized to pure native execution. Interpreters have square brackets; JIT compilers have braces; the rest are AOT compilers. vWasm* disables sandboxing. Note the log scale on the y-axis.*

that targets the special nature of software sandboxing. By careful optimized lifting of Wasm code to Rust, followed by compilation down to native code, it provides high-performance execution of Wasm code while guaranteeing safe sandboxing. By leveraging Rust's safety guarantees, rWasm provides safety *without requiring any explicit proofs* from the developer. Our benchmarks show that rWasm is competitive with, or on some benchmarks even beats, other Wasm runtimes, including ones optimized for performance, rather than safety. By leveraging Rust, rWasm provides the first multi-lingual, multi-platform sandboxing compiler with provably safety guarantees and competitive performance.

vWasm, implemented in F* [52], currently compiles Wasm programs into x86-64 assembly code, although the code and proofs are designed for portability. To prove its high-level theorem, we model a subset of x86-64 semantics and prove that the produced code satisfies the sandboxing statement given these semantics. rWasm, on the other hand, is implemented in Rust, and can compile code to any architecture that is supported as a target by Rust (which covers all the widely-used architectures). It also supports the ability to conveniently customize the output program, e.g., to add inline reference monitors [15].

Both vWasm and rWasm are competitive in performance. Figure 3 summarizes our results, showing the normalized execution time of a standard Wasm benchmark, PolyBench-C [47, 22] on popular Wasm runtimes. As a compiler, unsurprisingly, vWasm outperforms the interpreters. Additionally, rWasm is competitive even with the compilers which are optimized for speed, and not necessarily safety, only slower by 3% to 26% on average than the first three of the four faster runtimes on the list (wasm2c, WAMR in AOT compilation mode, and wasmtime respectively). The fastest, WAVM, is almost twice as fast as rWasm on average, but on some of the longer running PolyBench-C benchmarks, rWasm is more than twice as fast than WAVM, and thus we see that relative performance can vary drastically based upon workload.

We additionally compare both implementations on other quantitative and qualitative aspects, including development/maintenance effort and extensibility, in the full paper [7].

An alternative to provably emitting sandboxed code is to *validate* that code has been properly sandboxed (cf. NaCl [60], RockSalt [37], and VeriWasm [23]). However, these approaches require a custom validator for each targeted platform. NaCl and RockSalt also rely on a custom compiler toolchain for x86/x86-64 to make the emitted code easier to validate. Verifying code that was not so customized, e.g., with VeriWasm, is tricky to do without rejecting legitimate programs or suffering soundness issues. For example, VeriWasm missed CVE-2021-32629, a sandbox-compromising bug in Wasmtime [58] and Lucet [2], due to improper modeling of signedness in their specification [45].

Of course, a specification failure is problematic both for verified compilation and for validation. However, verified compilation allows greater control over the produced code; e.g., vWasm only needs to model a small, simple fragment of x64. In contrast, validation typically handles complex assembly produced by an independent compiler. The two approaches are complementary though, and ideally both would be used.

In summary, we explore two distinct techniques to achieve provably safe sandboxing of arbitrary code. Principled design allows us to achieve this without compromising on development velocity, runtime performance, or portability.

4 Agile Enforcement of Pre-Emptively Defensive Code

Status: Completed [35]

In addition to defending against 3$^{\text{rd}}$ party code, one might pragmatically also wish to defend against issues in one's own (i.e., 1$^{\text{st}}$ party) code. Naïve approaches for this might need to apologize on development costs, performance, debuggability, etc. However, we recognize that an agile stance on security enforcement help towards mitigating issues and vulnerabilities, whether it be in 1$^{\text{st}}$ or 3$^{\text{rd}}$ party software. We approach this through a principled extension to WebAssembly, that allows modulating the strength of enforcement (and relevant potential performance tradeoffs) based on real-world contexts, without needing changes to the code.

As described in Section 3, Wasm is a great narrow-waist for providing software sandboxing. This allows for isolating the impact of a vulnerability in one Wasm module from directly impacting another in the same process. Within the sandbox, however, Wasm offers little protection. Memory-unsafety in Wasm modules (compiled, say, from unsafe C code), can still cause havoc within their *own* memory space (which could then potentially be used to mount confused deputy attacks). Lehmann et. al. [28], for example, show how attackers can turn a buffer overflow vulnerability in the `libpng` image processing library (executing in a Wasm sandbox) into a cross-site scripting (XSS) attack.

To prevent such attacks, C/C++ compilers would have to insert memory-safety checks *before* compiling to Wasm—e.g., to ensure that pointers are valid, within bounds, and point to memory that has not been freed [39, 40, 42]. Industrial compilers like Emscripten and Clang do not. Also, they *should not*. Retrofitting programs to enforce memory safety gives up on *robustness*, i.e., preserving memory safety when linking a (retrofitted) memory-safe module with a potentially memory-unsafe module. It gives up on *performance*: efficient memory-safety enforcement techniques rely on operating system abstractions (e.g., virtual memory [11]), abuse platform-specific details (e.g., encoding bounds information in the (unused) upper bits of an address [1]), and take advantage of hardware extensions (e.g., Arm's pointer authentication and memory tagging extensions [3, 31]).

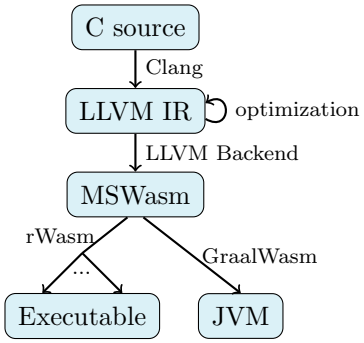

Figure 4: *End-to-end compilation pipeline. We first compile C to MSWasm (via LLVM), and then compile MSWasm to machine code using either our modified rWasm AOT compiler (which supports different notions of safety) or our modified GraalWasm JIT compiler.*

Finally, it also makes it harder to prove that memory safety is preserved end-to-end.

With *Memory-Safe WebAssembly*, Disselkoen et al. [14] propose to bridge this gap by extending Wasm with language-level memory-safety abstractions. In particular, MSWasm extends Wasm with *segments*, i.e., linear regions of memory that can only be accessed using *handles*. Handles, like CHERI capabilities [59], are unforgeable, well-typed pointers—they encapsulate information that make it possible for MSWasm compilers to ensure that each memory access is valid and within the segment bounds. Alas, the MSWasm position paper only outlines this design—they do not give a precise semantics for MSWasm, nor implement or evaluate MSWasm as a memory-safe intermediate representation.

Our work builds on this work to realize the vision of MSWasm.

We describe precise and formal semantics of MSWasm in our paper [35], giving precise meaning to the previous informal design [14], in addition to being able to prove useful properties such as *robust memory safety*, and sound compilation from C to MSWasm. Using our formal results as a guide, we implement both just-in-time (JIT) and ahead-of-time (AOT) compilers of MSWasm to native code, and a C-to-MSWasm compiler (by extending Clang). Figure 4 shows our end-to-end compilation pipeline. We describe the AOT compilers from MSWasm-to-machine code here.

Extending rWasm (Section 3) with 1900 lines of code, we introduce support for the MSWasm extension, and multiple backends to support modulating differing levels of memory safety (spatial and temporal safety, and handle integrity). This allows for a single MSWasm module, *without* changes, to have different enforcement mechanisms applied depending on external circumstances (e.g., maintain highest performance by default, but switch a specific module to stricter enforcement the moment a vulnerability is found, while waiting on a patch).

We benchmark MSWasm on PolyBench-C, the de-facto Wasm benchmarking suite [47]. We find that, on (geomean) average, MSWasm when enforced in software using our AOT compiler imposes an overhead of 197.5%, which is comparable with prior work on enforcing memory safety for C [40]. MSWasm, however, makes it easy to change the underlying enforcement mechanism (e.g., to boost performance), without changing the application. To this end, we find that enforcing just spatial and temporal safety imposes a 52.2% overhead, and enforcing spatial safety alone using a technique

similar to Baggy Bounds [1], is even cheaper—21.4%.

While these overheads are relatively large on today's hardware, upcoming hardware features explicitly designed for memory-safety enforcement can reduce these overheads (e.g., Arm's PAC can be used to reduce pointer integrity enforcement to under 20% [31], while Arm's CHERI [21] or Intel's CCC [29] can also reduce the cost of enforcing temporal and spatial safety). MSWasm will be able to take advantage of these features as soon they become available, for almost free, as illustrated by the ease of swapping memory-safety enforcement techniques within our AOT compiler.

In summary, we demonstrate that MSWasm provides a principled design for agile safety enforcement, that allows pre-emptively defending code based upon emerging threat contexts. We do this without apologizing for development velocity, and (for the most part) performance. Additionally, it unlocks easy upgrades to new enforcement mechanisms, both in software and hardware, promoting long term improvements at reduced development cost.

5 Parsing Untrusted Data

Status: Proposed/ongoing work

Serialized representations of data are common in all kinds of software, since software necessarily must interact with the real world, either with humans, other software (possibly on a different machine), or even itself across multiple invocations. Thus, serializers and parsers for data formats have been well studied. Unfortunately, they still remain a common source of security issues. Across the past 5 years, MITRE considers Improper Input Validation (CWE-20) and Deserialization of Untrusted Data (CWE-502) to be, "despite ongoing visibility to the community", some of 15 "most challenging weaknesses that exist today" [36].

Especially in high-assurance software, such as with formally verified network protocol implementations [5, 13], one might wish to use formally verified parsing and serialization. Prior works [49, 53, 56, 24, 12] introduce verified parsers, and serializers. However, these have involved various apologies, e.g., high complexity (for developers to use them), limited expressivity (only supporting features needed for their particular use case), or lack of broad applicability (often limiting usage only to users of the verification language).

We recognize that the cost of complexity and expressivity comes from a fundamental split in usage scenarios for serialized representations of data. In particular, the choice of the data format plays a non-trivial role in the decisions that a parser/serializer framework must make. We thus classify data formats into two broad categories, which we name *intrinsic* and *extrinsic*, described below. With this split, we design two separate parser/serializer frameworks, reducing burden for users. Additionally, for broad applicability, so that even unverified contexts can take advantage of these parsers, we use Verus [27, 26], a systems verification language built on top of Rust. This allows unverified Rust programs to also receive many of the benefits of a verified parser/serializer, without needing to rely on foreign function interface (FFI) or awkward API.

Our data format classification is driven by where the format might be used. If the format is intended to be used only with the same program (such as communicating over the network to itself, or saving state for future executions), then the choice of the data format is not as crucial as the high-level data itself. Said differently, the high-level data structures of the program must dictate

```
derive_marshallable_for_enum! {
  enum CSingleMessage {
    #[tag = 0] Message {
      #[o=o0] seqno: u64,
      #[o=o1] dst: EndPoint,
      #[o=o2] m: CMessage,
    },
    #[tag = 1] Ack {
      #[o=o0] ack_seqno: u64,
    },
    #[tag = 2] InvalidMessage,
  }
}
```

Figure 5: *Automatically implementing the* `Marshallable` *trait. Our macros automatically produce a marshaller, a parser, and proofs of all relevant lemmas for arbitrary* `enum`*s or* `struct`*s. The* `#[..]` *annotations guide, and provide fresh identifiers for the automatically generated proofs. These are a compromise due to the current* `macro_rules!`*-based definition of* `derive_marshallable_for_enum`*.*

the data format, and thus the data format is *intrinsic*. In contrast, software might need to talk to other software, or its communications must satisfy some pre-specified standard. In this case, the data format is *extrinsic*, and cannot be determined purely based on the whims of the program's high-level data structures.

Recognizing this domain split allows us to design frameworks that are targeted at each, allowing for a nicer user experience. Additionally, by not needing to optimize against both directions at once, both the design and implementation processes are simplified. Below, we describe our ongoing work on our two frameworks for parsing and serializing data formats.

Our framework for intrinsic data formats, called VMarshal, is a macro- and trait-based library in Verus that automatically generates parsers and serializers for arbitrary Rust `struct`s and `enum`s. Figure 5 shows an example of how a user of VMarshal can derive all the executable code and proofs for an arbitrary high-level data type. Custom hand-written implementations handle the same for `Vec`s and primitives such as `u64`. Figure 6 shows the definition of the `Marshallable` trait, showing the interface that users of this framework see on any type that has it implemented. In short, we show that the parser and serializer are inverses, that no serialization of a value can be the prefix of the serialization of a different value, etc.

Our framework for extrinsic data formats, called Veil, takes a different approach, but still in Verus. For Veil, we design a domain-specific language (DSL), inspired by Nail [4] (an unverified parser generator library), that helps define an extended Parsing Expression Grammar (PEG) [17]. From this, an unverified and untrusted compiler compiles this to Verus code and proofs, that can then be used in both verified and unverified contexts. These proofs demonstrate useful properties such as the ones listed above.

We have already started to see promising results on integrating both the VMarshal and Veil

```
pub trait Marshallable : Sized {
  spec fn marshallable(&self) -> bool;
  exec fn is_marshallable(&self) -> (res: bool)
    ensures res == self.marshallable();
  spec fn ghost_serialize(&self) -> Seq<u8>
    recommends self.marshallable();
  exec fn serialized_size(&self) -> (res: usize)
    requires self.marshallable(),
    ensures res as int == self.ghost_serialize().len();
  exec fn serialize(&self, data: &mut Vec<u8>)
    requires self.marshallable()
    ensures
      data@.len() >= old(data).len(),
      data@.subrange(0, old(data)@.len() as int) == old(data)@,
      data@.subrange(old(data)@.len() as int, data@.len() as int) == self.ghost_serialize();
  exec fn deserialize(data: &Vec<u8>, start: usize) -> (res: Option<(Self, usize)>)
    ensures match res {
      Some((x, end)) => {
        &&& x.marshallable()
        &&& start <= end <= data.len()
        &&& data@.subrange(start as int, end as int) == x.ghost_serialize()
      }
      None => true,
    };

  spec fn view_equal(&self, other: &Self) -> bool;
  proof fn lemma_view_equal_symmetric(&self, other: &Self)
    ensures self.view_equal(other) == other.view_equal(self);

  proof fn lemma_serialization_is_not_a_prefix_of(&self, other: &Self)
    requires
      !self.view_equal(other),
      self.ghost_serialize().len() <= other.ghost_serialize().len(),
    ensures
      self.ghost_serialize() != other.ghost_serialize().subrange(0, self.ghost_serialize().len() as int);

  proof fn lemma_same_views_serialize_the_same(&self, other: &Self)
    requires
      self.view_equal(other),
    ensures
      self.marshallable() == other.marshallable(),
      self.ghost_serialize() == other.ghost_serialize();

  proof fn lemma_serialize_injective(&self, other: &Self)
    requires
      self.marshallable(),
      other.marshallable(),
      self.ghost_serialize() == other.ghost_serialize(),
    ensures
      self.view_equal(other);
}

```

Figure 6: *The* `Marshallable` *trait.*

frameworks in other verified projects.

In summary, through a principled categorization of data formats, we enable formally verified parser and serializer frameworks that do not need to apologize for developer time, approachability, or expressivity.

6 Reconstructing Source-Level Types from Binary Executables

Status: Proposed/ongoing work

Legacy software is a challenge to secure, often through lack of (or unusable) source. While the actual securing of the software happens through methods like binary patching, software rewrites, etc., the first step begins with understanding what the software itself was engineered to do. Put differently, one must reverse engineer the software from the machine code "binaries". Many tools have been built to help reverse engineers perform their task, including disassemblers, debuggers, and decompilers. Decompilers, such as Ghidra [44], Binary Ninja [54], and HexRays [50], are popular amongst reverse engineers, since they provide a high-level source-code-like view over low-level machine code.

Unfortunately, despite decades of advancements in the science and art of decompilation, the quality of decompiled output leaves much to be desired. A persistent problem that has still not been solved is high quality source-level type information for the decompiled code. High quality source-level types aid not only in human understanding of program behavior, but can also aid the decompiler itself in producing higher quality decompiled output. Current decompilers used in practice are weak at automated type inference. Instead, they rely heavily on human experts to interactively provide better type information to improve decompiled output.

This problem of higher quality automated initial type inference has been studied extensively. Broadly, approaches fall into two camps: (i) *deductive*, and (ii) *inductive* type inference. The former derives types through a series of deductions, often by understanding the semantics of the program in question. Although deductive type inference approaches may have differences in sensitivity of analysis, constraint solving techniques, and even choice of static- or dynamic-analysis, a common theme is a focus on a balance of accuracy and conservativeness of the types produced. While rarely (if ever) explicitly concretized as such, the results of these approaches could be traced through a series of deduction steps, and thus (at least in theory) produce *reasonable* types. In contrast, inductive type inference focuses primarily on directly recovering source-level types, often through the use of machine learning. Despite having produced increasingly impressive results over time [9, 32, 10], it can be hard to depend on the correctness of their output, especially in scenarios that might differ drastically from their training corpus. Thus, here, we focus on deductive type inference. Caballero and Lin [8] provide a detailed survey of the sixteen years of work on type inference on binaries up to 2015. Furthermore, recent works [43, 33] on deductive type inference have continued to show increasing sophistication, both in techniques and the expressivity of types produced.

However, despite decades of published works on automated type inference on binaries, the state-of-the-art tools used by practitioners in the field barely recover any types automatically, and require a large amount of human annotation. Non-trivial factors in this could be that the tools

described in the aforementioned works are not available for use, comparisons between them were largely based on inconsistent and non-reproducible benchmarks, and that the techniques involve complexities and subtleties that are not captured in their papers.[1]

To truly understand this gap between decompilation in practice, and decompilation in academic works, we decided to develop a new tool to perform better automated initial type inference. Additionally, as a step towards reversing the unfortunate tendency of closed-source tools and non-reproducible benchmarks, we intend to both open-source our tool, and release reproducible scripts for the benchmarks.

During our preliminary study, we studied the requirements for automated initial type inference. Here, we observe that there is a mismatch in viewpoint adopted by prior techniques, and what reverse engineers expect from their tools—while automated initial type inference tools attempt to *recover* source-level types, this is often impossible; instead one would prefer the *reconstruction* of types that capture the actual *behavior* of the variable in the code. In particular, since source-level (often C, the de-facto output language for decompilers) types are *nominal* types, where names of types essentially define the type (by controlling its usage), these are somewhat ill-suited to the task of automated type inference from binaries. In contrast, structural types, where the available behaviors/features of the type define the type, better capture what reverse engineers expect to see from their tools.

In accordance with our aforementioned new world-view of type reconstruction, we implement a new tool, named Kairos, for automated type inference of source-level types for binary code. Kairos makes only a small number of assumptions on its input and output, allowing it to be agnostic to the choice of both the binary's architecture, as well as choice of output source-level types. In particular, it produces types that are far more expressive than C. Kairos produces machine-readable types, aimed at (potential) further downstream analyses, and human-readable C-like types projected from the more precise internal machine-readable types. Phase separation is not limited to just this distinction between machine-readable and human-readable types, but extends further into both in the internal design of our approach, and also how it integrates with over-arching binary analysis frameworks. While correct disassembly or correct control flow recovery is a known hard problem (indeed undecidable due to the halting problem), by separating concerns explicitly, different parts of the binary analysis pipeline can be improved upon independently. Indeed, in a world of hard problems, it seems prudent to carve out explicit boundaries with conditional conservativeness guarantees attached. This is why we have designed Kairos to be useful with any binary analysis framework, without tight coupling to any of them. For our own reverse engineering projects, and also to improve the broader open-source binary-analysis ecosystem, we build in support for Ghidra, but also note that only approximately 1000 lines of code are Ghidra-specific.

In the process of constructing Kairos, we have encountered various challenges and discovered various insights about type reconstruction, that to the best of our knowledge, are either novel, or are tacit knowledge in the community and remain unpublished. These insights help drastically reduce the complexity of the design and implementation of the tool, while still providing high expressivity and quality of output.

One of our core insights is that there can be no perfect objective evaluation of type reconstruction,

[1]Indeed for a recent paper [43], other employees at the same company attempting to reproduce the work state "It is a powerful system but difficult to understand" and "presentation is very dense and subtle" [20]

since there can be no correct ground truth. Specifically, we construct a collection of C source code with drastically different types, that compile to the exact same assembly code, showing that none of them can be *the* valid ground truth. Instead, we need a new understanding of what ground truth itself is, and that picking between the various possible outputs would necessarily be a subjective choice.

Another insight is that a common design decision that *must* be made is to choose some balance of conservativeness, and getting valuable output from a tool. While naïvely, one might wish that a deductive type inference system never makes any non-conservative leaps, a hypothetical fully-conservative analysis would often quickly stonewall into being unable to provide anything useful, or instead inundating its user with largely irrelevant information. For example, if a register is only used to perform an 32-bit arithmetic integer add operation, then it is unknown whether its type is a signed or an unsigned int; picking either of them would be a non-conservative decision, but at the same time, outputting their union (or intersection) adds additional burden to the human analyst. Since, in practice, many C developers simply use `int` (which is a signed integer) when they need an integer without worrying about signedness, it might seem reasonable that the tool should output `int` in this scenario as well. Nonetheless, at this point, the tool has made an opportunistically non-conservative choice; and non-conservative choices, if not controlled and accounted for, can add up and snowball into providing inaccurate information. Thus, management of non-conservativeness is crucial to producing useful output. Kairos manages the conservativeness loss using a collection of distinct phases, along with toggles that a reverse engineer can flip if they require more conservative analysis than the default (say, if analysing software that has been deliberately obfuscated to thwart automated analyses).

Additionally, in exploring the difficulties of various phases for type reconstruction, we have discovered an algorithmic hardness result for a phase we call *type rounding*, which we can show to be NP-hard.

To evaluate Kairos, especially with our insight regarding the lack of a valid ground truth, we propose to evaluate it qualitatively using hand-written examples to better understand its behavior, in comparison against other tools available to reverse engineering practitioners, as well as quantitatively by proposing a new metric. Our new metric attempts to capture the expectations of reverse engineers, while working around the inherent difficulty in objectively evaluating type reconstruction. Preliminary results are already promising, where Kairos provides types that improve over the state of the practice.

In summary, we propose a new principled lens to look at automated type analysis for decompilation, namely type reconstruction. This lens accounts for the inherrent impossibility of recovering lost source level types, and instead focuses on improving the experience of reverse engineers in practice. To achieve this, we have a new formalism that need not compromise on either elegance or output quality, to advance software comprehension. This formalism helps recognize crucial new insights in the field. By making Kairos open-source (along with reproducible benchmarks), and by re-aligning the task with what reverse engineers need, we remove apologies for measurability, output expressiveness, and quality.

7 Proposed Timeline

This section proposes a timeline for completing my thesis:

- Jan 10: Thesis Proposal
- Jan 10—Feb 8: Writing for Type Reconstruction paper (Section 6) for submission to USENIX Security, and for a paper that includes the Veil work (Section 5) to the same deadline.
- Feb 8—Mar 15: Thesis writing
- Mar 15—mid April: Even more writing, and prep for defense
- mid April: Defense. XKCD 1403?
- Post Defense: Celebrate?

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, 2009.

[2] Announcing Lucet: Fastly's native WebAssembly compiler and runtime. `https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime`, March 2019.

[3] Arm. Armv8.5-a memory tagging extension. *White Paper*, 2019.

[4] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 615–628, Broomfield, CO, October 2014. USENIX Association.

[5] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[6] Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. Verified transformations and Hoare logic: Beautiful proofs for ugly assembly language. In *Proceedings of the Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, July 2020.

[7] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-safe multilingual software sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*, August 2022.

[8] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys*, 48(4):65:1–65:35, May 2016.

[9] Ligeng Chen, Zhongling He, and Bing Mao. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98, 2020.

[10] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types.

In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, Boston, MA, August 2022. USENIX Association.

[11] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security*, 2017.

[12] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[13] Antoine Delignat-Lavaud, Cedric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2021.

[14] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2019.

[15] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, page 87–95, New York, NY, USA, 1999. Association for Computing Machinery.

[16] Ethereum WebAssembly (ewasm). `https://ewasm.readthedocs.io/en/mkdocs/`, 2021.

[17] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 111–122, New York, NY, USA, 2004. Association for Computing Machinery.

[18] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2019.

[19] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight Wasm runtime for the Edge. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery.

[20] GrammaTech. Type inference (in the style of retypd). `https://github.com/GrammaTech/retypd/blob/f8dd231478c3e1722d0d160c3cf99c628a257022/reference/type-recovery.rst`. Archived: `https://archive.is/GbsUB`, July 2021.

[21] Richard Grisenthwaite. Supporting the UK in becoming a leading global player in cybersecurity. `https://community.arm.com/blog/company/b/blog/posts/supporting-the-uk-in-becoming-a-leading-global-player-in-cybersecurity`, 2019.

[22] 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. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[23] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Trust but verify: SFI safety for native-compiled Wasm. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, February 2021.

[24] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating lr(1) parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, page 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. Portable software fault isolation. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 18–32, 2014.

[26] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jay Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In Submission.

[27] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 2023.

[28] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[29] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, et al. Cryptographic capability computing. In *IEEE/ACM International Symposium on Microarchitecture*. ACM, 2021.

[30] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*. SEE, 2016.

[31] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[32] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. Typeminer: Recovering types in binary programs using machine learning. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 288–308, Cham, 2019. Springer International Publishing.

[33] Matthew Maurer. *Holmes: Binary Analysis Integration Through Datalog*. PhD thesis, Carnegie Mellon University, Oct 2018.

[34] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium (USENIX Security 06)*, Vancouver, B.C. Canada, July 2006. USENIX Association.

[35] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. MSWasm: Soundly enforcing memory-safe execution of unsafe code. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2023.

[36] MITRE. Stubborn weaknesses in the cwe top 25. `https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html`, 2023.

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

[38] Mozilla. Measurement dashboard. `https://archive.is/ku005`, January 2024.

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44(6):245–258, June 2009.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31–40, June 2010.

[41] 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 *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.

[42] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[43] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 27–41. Association for Computing Machinery, Jun 2016.

[44] National Security Agency (NSA). Ghidra. `https://www.nsa.gov/ghidra`.

[45] Authors of VeriWasm. Private Correspondence.

[46] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. eWASM: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3492–3505, 2020.

[47] PolyBench-C: the Polyhedral Benchmark suite. `https://web.cs.ucla.edu/~pouchet/software/polybench/`. Accessed: January 2021.

[48] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy,

Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.

[49] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure Zero-Copy parsers for authenticated message formats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, Santa Clara, CA, August 2019. USENIX Association.

[50] Hex-Rays SA. Hex-Rays Decompiler. `https://hex-rays.com/decompiler/`.

[51] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, Washington, DC, August 2010. USENIX Association.

[52] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[53] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. Hardening Attack Surfaces with Formally Proven Binary Format Parsers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2022.

[54] Vector 35. Binary Ninja. `https://binary.ninja/`.

[55] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[56] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. Comparse: Provably secure formats for cryptographic protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 564–578, New York, NY, USA, 2023. Association for Computing Machinery.

[57] WASI – The WebAssembly System Interface. `https://github.com/WebAssembly/WASI`, 2021.

[58] Wasmtime: A small and efficient runtime for WebAssembly & WASI. `https://wasmtime.dev/`, 2021.

[59] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[60] 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 *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.