A Principled Approach towards Unapologetic Security

Jay Bosamiya

CMU-CS-24-127

May 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Bryan Parno (Chair)
Jonathan Aldrich
Phillip Gibbons
Chris Hawblitzel (Microsoft Research)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

The work presented in this thesis was supported in part by the Department of the Navy, Office of Naval Research under Grant No. N00014-18-1-2892, an internship at Microsoft Research, a Cylab Presidential Fellowship, a grant from the Alfred P. Sloan Foundation, a grant from the Intel Corporation, the USENIX 2022 Internet Defense Prize, the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab), and the CONIX Research Center (one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA). Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of these sponsors.

Chapter 5 includes work from co-authors partially supported: by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019), by two grants from the Alfred P. Sloan Foundation, by gifts from Intel and Google, by the National Science Foundation under grants CAREER CNS-2048262 and CNS-2155235, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by a generous gift to support research on applied cryptography and society in the Center for Research on Computation and Society (Harvard University) and Computing Innovating Fellowship (2021). Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of these sponsors.

Keywords: Formal Techniques, Unapologetic Security, Software Verification, Cryptographic Implementations, Sandboxing, WebAssembly, Parsing, Decompilation

To my family and my friends

Abstract

Software is incredibly difficult to write correctly, let alone safely. Prior work (quite successfully) has, amongst other things, relied on formal verification—a powerful hammer to achieve provable guarantees. However, improvements in the state-of-the-art of software security often come with significant apologies for other important objectives, such as development velocity, software performance, or loss of functionality. While many developers (and more so, users) would like their software to be secure, these apologies cause security to become 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 argue that security objectives are achievable *without* apology, through the use of principled approaches and formalism. To validate this thesis, we look at a collection of case studies that span across a wide collection of different kinds of software systems: (i) high-performance cryptographic primitives, (ii) safe execution of arbitrary untrusted code, (iii) agile safety enforcement for code, (iv) low-level serializers and parsers for untrusted data, and (v) source-unavailable executable comprehension. In each, we demonstrate that principled approaches and formalism help remove the need for the apologies required by prior work.

Our hope is that providing security without apology, even in the face of practical complexities, makes a big step towards the shared goal of security researchers—making security the natural default choice.

Acknowledgments

This work would not have been possible without the help and support of my mentors, collaborators, friends, and family.

First and foremost, I would like to thank my advisor, Bryan Parno. It has been an absolute privilege to work alongside him for the past seven years. He has been a source of great inspiration in both work, and life. Bryan's systematic approach, and attention to detail, while never losing sight of the big picture, is genuinely amazing. Especially as someone with varied interests, I am also sincerely grateful for the enthusiastic support for exploring my own directions. Also crucially, I am glad that, despite our deep divide in choice of text editors, I can call him a friend.

I would also like to thank my committee members, Jonathan Aldrich, Phil Gibbons, and Chris Hawblitzel, for their feedback, guidance, and support that has helped shape this dissertation into one of which I am proud.

Additionally, sincere thanks to Maverick Woo, who has played a significant role in how I look at seemingly-solved problems. His approach to meticulously breaking a problem down to fundamental underlying assumptions has been eye-opening. I also appreciate the long discussions on the many research-adjacent (and non-research) aspects of the graduate experience.

I also extend my thanks to Justine Sherry's "Readings on Research" reading group, which was incredibly helpful as a first-year graduate student trying to understand the world of academia and research.

Thanks also to all my co-authors on the papers we worked on together, whether they made it into this dissertation or not. I have learnt so much from each of you, and I am grateful for the opportunity to work with you all.

My fellow research group members were crucial, not only for wonderful discussions, but also for keeping me sane across these many years. Abhiram Kothapalli, Aymeric Fromherz, Ben Lim, Chanhee Cho, Josh Gancher, Lisa Masserova, Mike McLoughlin, Pratap Singh, Steve Matsumoto, Sydney Gibson, Travis Hance, Val Choung, Yi Zhou, and Zhengyao Lin—you all have taught me so much in the many hours spent discussing life, research, and everything in between. I am going to miss our times being part of this vibrant group. Special shout-out to Yi and Chanhee for the countless late-evening discussions, and to Lisa for the many chats over board-games and tea.

Also a shout-out to the late-evening discussions with fellow CyLab, and CSD members. Too many to list, but I appreciate all the conversations and debates.

I am also extremely thankful to long-time friend, Jenish Rakholiya, who has probably heard more ideas, gripes, and rants than probably anyone else on this (or any other) planet. His enthusiasm and support have been invaluable.

Sincere thanks to my fellow members in the Plaid Parliament of Pwning, who are some of the most phenomenal hackers I've had the privilege to meet. Special shout outs to Matt Savage, Ethan Oh, Mindy Hsu, Kevin Stephens, Andrew Haberlandt, Carolina Zarate, Corwin de Zahr, Zach Wade, Jenish Rakholiya, Palash Oswal, and Sam Kim, who I've spent so many of my weekends playing CTFs (and board games!) with, and learnt so much from.

Many thanks to my parents, Asmita and Hitesh, for their continuous support and encouragement. And to my brother, Ish, who is always enthusiastic and excited about whatever I do—I sincerely appreciate it.

And last but definitely not least, I would like to thank everyone I somehow forgot to thank above. I owe you at least a coffee for this mistake, and likely much more for all your help!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hologram of the Doctor: *This is security protocol 712. This time capsule has detected the presence of an authorized control disc, valid one journey. Please insert the disc and prepare for departure.*

Doctor Who
S03E10 "Blink"

1.1 My Thesis

Security objectives are achievable without apologies previously inhibiting secure software development.

Software development and security have been in a constant tug of war for decades. Development velocity, execution performance, maintainability, and more are important criteria that have seemingly come in repeated apparent conflict with security. Too often, due to market considerations, security ends up taking a back-seat, as evidenced by the widespread proliferation of vulnerabilities, breaches, and exploits.

Developers (and perhaps more so, users) of software would like their code to be secure. Indeed, for folks at high risk of attacks, and for security researchers, it is rational to aim for security above all else. However, security too often comes with an apology for at least one of the other important criteria for good software (often multiple at once), and thus becomes under-prioritized. In practice, a large fraction of software users would not like to make this tradeoff of accepting software that takes longer to develop/maintain, has fewer features, and ends up slower. Thus, we have the current state of the world—widespread insecure software.

Insecure software, and the study of securing it, is by no means recent. In fact, security as a field has advanced by leaps and bounds across the decades. While a significant fraction of software would've crashed on a dark and stormy night [92], and had its stack smashed for fun and profit [110] at the end of the 20th century, today we security researchers know many techniques to make software more secure. The field has gained great depth in static and dynamic analysis, fuzzing, effective exploit mitigations, and even formal methods to provide mathematical guarantees. Truly, there has never been a better time for us to write safe software.

Yet, to be adopted, advances in security 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.

My thesis thus focuses on the important practical question that arises from the interaction of security and the development of practical software systems—whether we can both have our cake and secure it too. With this dissertation I show that we emphatically *can* achieve security *without* compromising on the aforementioned other axes of development velocity, maintainability, runtime performance, and more, which have plagued prior approaches to developing secure software.

My hope is that this dissertation kicks off greater interest and excitement into formal and principled techniques that advance our shared goal of making the software world a better place, where we share all the joys of features, maintainability and performance, without compromising on security.

1.2 Structure of the Dissertation

Software is not homogeneous, and thus we demonstrate achieving unapologetic security with a breadth of case studies. For each, we look at the core security requirements and existing apologies preventing the adoption of security, and demonstrate how we can manage and eliminate these apologies through judicious use of formalism and principled approaches.

The rest of this dissertation is structured as follows:

- Chapter 2 gives some shared background across the latter chapters, although each chapter will introduce necessary background too.

- Chapters 3 to 7 each focus on a case-study in achieving unapologetic security. The case studies in each chapter consist of collaborative work—in this dissertation, we focus primarily on parts that are mine, along with some work by my papers' co-authors when necessary to aid understanding; the start of each chapter specifically clarifies my contributions for the relevant paper(s). We briefly explore each of these case-studies in the rest of this chapter.
- Finally, we conclude with Chapter 8 that discusses broader lessons learned, future directions, and more.

In summary, with this dissertation, we show that principled approaches and formalism help alleviate situations where security was previously believed to be in conflict with other goals. Specifically, we show, through a collection of case-studies across a wide breadth of software, that security is achievable without needing to apologize for important goals such as development velocity, software performance, portability, and even elegance.

1.2.1 Securing High-Performance Code

In Chapter 3, we focus on high-performance code that is at the heart of securing the world's Web traffic—cryptographic primitives.

AES-GCM secures 98.9% of the world's internet traffic [97]. 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 yet another 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 [116]. Yet 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 uses verified code transformations to allow 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 up to 27% beyond OpenSSL's fastest implementation.

In short, we show that for important ultra-high-performance software, formal techniques can be used to provide high-assurance safety, without apologizing for development velocity, or runtime performance.

1.2.2 Surviving Arbitrary Code

Next, in Chapter 4, 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) [45], 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.

1.2.3 Pre-Emptively Defending Code

Expanding further in Chapter 5, practically, one might additionally want to be pre-emptively defensive against one's own (i.e., 1st party) code, in addition to 3rd 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 using MSWasm—an intra-module memory-safety-aware extension [28, 91] to WebAssembly. The compiler from C to MSWasm requires no changes to the C code, thereby alleviating developer burden.

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.

1.2.4 Handling Untrusted Data

Switching gears to untrusted data in Chapter 6, 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 [119, 137], 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 implement 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. The qualitatively better user experience for intrinsic formats derives from having to only work with the high-level data, while for extrinsic formats, a convenient expressive domain-specific language (inspired by the unverified Nail tool [9]) to succinctly capture both low-level format *and* high-level data model. We additionally integrate these different serialization libraries 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.

1.2.5 Dealing with Legacy Code

Finally, we focus on systems with unusable or unavailable source code in Chapter 7.

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 advancement in the art and science of decompilation (i.e., reverse compilation), a lack of high-quality source-level types plagues decompiled code. Accurate type information is crucial in understanding program behavior, and existing decompilers rely heavily on manual input from human experts to improve decompiled output.

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.

We thus propose an alternate perspective—type *reconstruction*, which accounts for the inherent impossibility of recovering lost source types. In contrast with type *recovery* that attempts the impossible, type reconstruction focuses on obtaining types that best capture the practically observable behaviors of the executable code.

Adopting this new perspective, we implement TRex, a tool that performs automated deductive type reconstruction. Compared with Ghidra [106], a state-of-the-art decompiler used by practitioners, TRex shows a noticeable improvement in the quality of output types on 115 of 125 binaries.

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, aiding in eventually securing legacy code.

Chapter 2

Background

The story so far: In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move.

The Restaurant at the End of the Universe
Douglas Adams

To build unapologetically secure systems, we must first understand some preliminary material. While each of the upcoming chapters contains relevant background close to its applicable use, we collect some shared pre-requisites here that will aid in understanding latter chapters better. Readers already familiar with these concepts can safely skip this chapter.

2.1 Defining Security

When we think of securing some software, we might colloquially think of software as either being secure or insecure. However, to properly define the security of a software system, it is important to define the relevant attacker context. Without modeling potential threat, we can neither call software secure nor insecure.

Threat modeling is an identification of what an adversary/attacker is capable of, and what they wish to achieve. A system is secure if and only if an adversary is unable to use its capabilities to achieve their goals; i.e., they are prevented from escalating from their initial capabilities into achieving their objective. Put differently, the security of the software system derives from the enforcement of *limitations* placed on an attacker.

Perhaps counter-intuitively, this means that even the first non-trivial C program written by someone (with buffer overflows, and format string vulnerabilities), may not necessarily be insecure. In particular, if there is no sensitive data, or differing privileges, then a vulnerability itself does not cause harm. However, such code is a ticking time-bomb, since the moment such code comes into contact with some privilege boundary, it can allow an attacker to escalate their privilege. This perhaps explains why colloquially, we would consider such code insecure, even if it is *not yet* insecure.

Thus, to understand the security of a system, it is important for us to understand what we are protecting, and what the attacker is capable of. With that, the security of a system derives from the *how*. Formal reasoning lets us talk and reason about these topics in a mathematically rigorous fashion.

2.2 Formal Verification

Interpreted broadly, formal methods in software engineering simply refers to the use of mathematical rigor in developing software. This spans a wide range of topics, including, but not limited to, formal specification, model checking, abstract interpretation, synthesis, property-based testing, formal verification, and more.

Our focus here is on formal verification—a particular aspect of formal methods that deals with proving (or sometimes disproving) some formal property of a computer system. Formal verification is one of the best tools we have to provide strong guarantees about software systems. In particular, it allows for machine-checked proofs (in the mathematical sense) that the software satisfies a formal specification. In contrast to (say) testing, or fuzzing, which can only tell us about the execution of the software on a small (finite) number of inputs, formal verification allows us to reason about the execution of code on *all* possible inputs, before even executing it once.

Formal verification of software derives from decades of work on reasoning about software, starting with seminal work by Floyd [33] and Hoare [48] in the late 1960s. Their work, known as Floyd-Hoare logic (aka Hoare logic), forms the crucial foundation of much of what we do today. At its core, Hoare logic is a collection of axioms and rules that revolve around a judgment called *the Hoare triple*, which describes how some code changes the state of the system. More precisely, $\{P\}\, C\, \{Q\}$ denotes the judgment that: starting from a state where P is true, upon finishing execution of the code C, we reach a state where Q is true.

Modern formal verification has grown well beyond the simplistic situations that could be dealt with in the early years of verification, but still utilizes the same deductive nature and general structure of the pre- and post-conditions to reason about code. At a high-level, to prove that some software is secure under a specification, one first writes a precise formal specification, and then uses a prover to show that the (mathematical model of) the software matches the specification, through a series of machine-checked deductive steps.

Unfortunately, due to Rice's theorem [124], all non-trivial semantic properties of programs are undecidable; thus, formal verification cannot be fully automated. For strong security, the soundness of the analysis cannot be compromised; i.e., no insecure program should ever be considered secure. Due to the undecidability, this often means a compromise on completeness (i.e., there will be safe programs that the machine might not call safe).[1]

In practice, this means that formal verification often falls into some point in a spectrum of possibilities. On one end, *interactive theorem proving* (such as with Coq [138] or Lean [22]) is the most "manual", requiring detailed human-provided proofs to show that the property holds. On the other end, *push-button verification* [133] is the most automated, requiring no manual annotations or proofs about the implementation code, but restricts the space of programs it works with and properties that it can prove. Somewhere in the middle, auto-active verification, is what the rest of this section (as well as most of this dissertation) focuses on.

Auto-active verification, as exemplified by F* [136], Dafny [74], or Verus [68, 69], uses powerful solver engines (called SMT solvers), to automatically discharge many of the proofs, by translating the code and specifications into *verification conditions* that the solver can understand. These solvers cannot get around Rice's theorem, however, and thus verification tools require some manual auxiliary annotations, on top of the specifications and code, to successfully verify the proofs. These annotations can take the form of pre-/post-conditions, assertions, calls to lemmas, and invariants for loops. Some tools (such as F*) also use dependent types, and a tactic engine, reminiscent of those seen in interactive theorem provers.

Prominent examples of formal verification include CompCert [76], the first formally-verified C compiler, and seL4 [61], the first formally-verified general-purpose microkernel. More recently, formally verified code has even started to be deployed in popular publicly used software, such as Mozilla Firefox, the Linux Kernel, and the Wireguard VPN [116].

[1]Technically, a prover might compromise on termination instead, but since we would like to eventually run the code, we often place some reasonable timeout, beyond which a program is not considered safe, even if it might well be. Hey, we've got places to be, and sights to see.

2.3 Trusted Computing Base

To the best of our knowledge, we need to trust the correctness of some components of our system in order to achieve the security policy. As long as these components behave correctly, the misbehavior of the rest of the system should not compromise security. These components collectively form the Trusted Computing Base (TCB) of the system [66, 107].

Analogous to the axioms used to prove theorems in Mathematics, the TCB forms the core foundation from which the security of the system is derived. More precisely, formal verification reduces the attack surface of the system to the TCB. If any of the components of the TCB are broken, then all bets are off. On the flip side, once a software system is proven secure under the assumptions of its TCB, the scope of potential attacks is drastically reduced from vulnerabilities that could exist anywhere in the system, to only possible issues in the (usually smaller) TCB.

For practical security, it is crucial to have a trustworthy TCB, since the TCB is the part of the system being verified that is taken *on faith* to work as intended. Note that we distinguish *trustworthy* from *trusted*—the former has a positive connotation (something worthy of trust), while the latter demonstrates the "weak underbelly" of the system (something trusted, whether deserving of it or not). In an ideal scenario, all trusted components would be trustworthy—they should have demonstrated good evidence that they would not break their guarantees.

To further illustrate the importance of understanding the TCB, let's consider a particularly "obvious" implicit assumption made by most secure systems—the impossibility of teleportation. Despite our best hopes, dreams, and stories in science-fiction, current Physics does not support teleportation.[2] Trust in this fact is what allows us assurance in the security of an air-gapped computer system—a system that is physically isolated from other outside networks [132], thereby necessitating physical access to control. If we suddenly woke up one day in a world where teleportation was real, then no air-gapped system would be secure. However, all known evidence leads us to believe that this trusted assumption of teleportation's impossibility, is indeed trustworthy. Thus, while taken on faith, it is a good trusted assumption to make.

While it is fascinating to think about the implications of a different physical universe, for the most part, when discussing the security of computer systems, we often only refer to the computationally-relevant aspects of the trusted base—the "C" in TCB. This computing

[2]Here, we ignore small-scale quantum mechanical phenomena, such as tunneling. Teleportation, as seen in (say) Marvel or DC comics, does not appear to be an actual feature of our universe. Sadly.

base includes both the hardware and software components that the system is built upon. Non-exhaustively, this often includes the CPU, I/O peripherals, memory, operating system, and more.

Often, large swathes of the TCB are left implicit when describing the security of systems, due to many components being obvious from context. In the upcoming chapters, we too will not do a fully exhaustive accounting of the TCB. However, we endeavor to be precise with the specific components that might lie on the boundary, since these are the most subtle, and the most crucial to understand correctly. Nonetheless, generally speaking, for auto-active verification projects, the TCB consists of the verification tool (including the verification condition generator, and the extraction pipeline that produces executable code), the SMT solver, the specification of the property (and relevant definitions, as well as axioms), and the actual underlying system that the code executes on (hardware, OS, peripherals, etc.).

Chapter 3

Securing High-Performance Code

Whiterose: *Every hacker has her fixation. You hack people. I hack time.*

Mr. Robot
S01E07 “esp1.7_wh1ter0se.m4V”

Our first case study in achieving unapologetic security focuses on the high-performance code that is at the heart of securing the world’s internet traffic—cryptographic primitives. These, for important business reasons, need to be as fast as possible; and for important technical reasons, this means that the code is ugly hand-written assembly. As crucial infrastructure code that is fundamental to the security of a large fraction of internet traffic (AES-GCM secures 98.9% of the world’s Web traffic [97]), we believe it is important to formally verify its correctness and safety. However, hand-optimized assembly code is often difficult to formally verify. In this chapter, 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, the strong safety net of verified transformations enables us to perform additional platform-specific performance improvements, allowing us to optimize code up to 27% beyond OpenSSL’s fastest implementation.

Contributions. The work presented in this chapter first appeared at VSTTE’20 [12]. As lead author, I designed, implemented, and verified all the transformers, and the automated optimizer. The updated Vale operational semantics, and the migration to transformer-based proofs for AES-GCM are primarily by my co-authors.

3.1 Introduction

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 [97],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. The resulting code is extremely fast, but difficult to understand, maintain, and verify.

Recent work on EverCrypt [116] 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. 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 [37]. 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 code transformations to recover the elegance of Hoare logic. In this approach, the programmer uses Hoare logic to verify a clean, modular version of the code. In addition, the programmer writes (but does not directly verify) the optimized, non-modular version of the code. 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.

The rest of this chapter is organized follows. Section 3.2 presents background on the Vale language and tool [11, 37], which provides the operational semantics and Hoare Logic reasoning for our assembly language code. Section 3.3 presents our verified transformation tool and describes how it deals with subtle equivalence issues, such as assembly language status flags. Section 3.4 applies the tool to an important real-world case study: OpenSSL's optimized AES-GCM. Section 3.5 shows that our tool can verify alternate interleavings of OpenSSL's code that are faster than the original code. Section 3.6 compares to related work, including related verification of cryptographic code such as Fiat-Crypto [30] and Jasmin [2, 3]. Section 3.7 concludes with recommendations for verifying optimized code.

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 development velocity, maintainability, or runtime performance. Indeed, it even safely unlocks greater performance than previously achieved by unverified approaches.

All of our code and proofs are available online, under an open source license.[1]

[1] `https://github.com/project-everest/hacl-star/tree/_vale_unstructured/vale`

3.2 Background: Vale and Assembly Language

In order to verify x64 code for AES-GCM, previous work [37, 116] defined syntax and operational semantics for x64 instructions as F* [136] datatypes and functions. In Figure 3.1, we provide a representative sampling of these definitions.

```
type reg = Rax | Rbx | Rcx | Rdx | ...
type operand =
  | OConst: n:int -> operand
  | OReg: r:reg -> operand
  | OMem: m:mem_addr -> operand
type ins =
  | Mov64: dst:operand -> src:operand -> ins
  | Add64: dst:operand -> src:operand -> ins
  ...
type code =
  | Ins: ins:ins -> code
  | Block: block:list code -> code
  ...
type state = {
  ok:bool;
  regs:reg -> nat64;
  flags:nat64;
  mem:map int nat8;
}
let eval_ins (ins:ins) =
  ...
  match ins with
  | Mov64 dst src -> ...
  | Add64 dst src -> ...
  ...
let rec eval_code (c:code) (s:state) ... =
  match c with
  | Ins ins -> Some (run (eval_ins ins) s)
  ...
```

Figure 3.1: Representative sample of instruction syntax and operational semantics

Here, the big-step operational semantics defined by `eval_ins` and `eval_code` evaluate the effects of assembly language instructions on some state, producing a new state as a result. The state tracks the values in registers (`regs`), the CPU status flags (`flags`), and the memory (`mem`). An additional state field `ok` indicates whether execution has succeeded or crashed. (This description is simplified for clarity; the full F* implementation of `state` also includes multimedia (xmm) registers, a stack, and a more complex memory model.)

The previous work then used the Vale tool [11, 37] to build a Hoare logic on top of the F* syntax and semantics, building the Hoare logic as verified rules on top of the operational semantics, so that the operational semantics in F* were part of the trusted computing base but the Hoare logic and the Vale tool did not need to be trusted. The Vale language uses procedures with modifies, requires, and ensures clauses to express the Hoare logic semantics of instructions like `Add64` and to build more complex procedures on top of instructions, as shown in Figure 3.2.

```
procedure Add64(inout dst:dst_opr64, in src:opr64)
    modifies efl;
    requires src + dst < pow2_64;
    ensures dst == old(dst + src);

procedure Test()
    modifies efl; rax;
    requires rax < 100;
    ensures rax == old(rax + 2);
{
    Add64(rax, 1);
    Add64(rax, 1);
}
```

Figure 3.2: Example Vale procedures

In this work, we leverage the distinction between operational semantics in F* and Hoare logic in Vale to define verified transformers (Section 3.3) that translate between idealized, structured code and optimized, unstructured code. Specifically, we first use Vale's Hoare logic to verify the idealized code. Since the Hoare logic rules are already built on top of the operational semantics, this gives us a proof about the idealized code in terms of the operational semantics. We then define verified transformers in terms of the x64 syntax and operational semantics, without having to modify the Hoare logic. These verified transformers prove that the operational semantics for the idealized code is equivalent to the operational semantics for the optimized code.

The transformers work by comparing the idealized code to the optimized code, where both versions of the code are expressed as an F* datatype, as in the `ins` datatype shown above. Since the code is a datatype, the transformers can inspect the code by pattern matching on the datatype. For better modularity, however, we used a slight variant of this approach: we refactored the `ins` type to be a more general dependent type that contains an arbitrary number of input and output operands and a function that computes the values for the output operands from the values from the input operands. With this, the transformer only has to match on a small number of general instructions rather than matching separately on `Mov64`, `Add64`, etc. (For the most part, only the input and output operands matter; whether an instruction adds numbers, subtracts numbers, or just moves numbers is usually irrelevant to the transformations.)

3.3 Verified Code Transformers

It is easier to write beautiful proofs about modular code. Our goal is to enable such proofs even for high-performance ugly code. We achieve this by designing a collection of verified code transformers, which allow the developer to write proofs about the modular code and then apply one or more transformers to automatically produce the high-performance ugly code. By proving that each transformer preserves the semantics of the original code, we ensure that the results of the elegant proofs carry over to the ugly code.

Below, we describe the core workflow for a developer using these transformers (Section 3.3.1), details about their design, implementation, and verification (Section 3.3.2), as well as several transformers (Section 3.3.3) which have significantly improved the modularity of the proofs for AES-GCM, a case study we describe in Section 3.4.

3.3.1 Developer Workflow

To make use of verified code transformers, a developer first writes a clean, modular version of their code, and writes proofs about it. Next, they write, but prove nothing about, a performance-optimized version of their code. This can be based on existing code (e.g., OpenSSL's), their own intuition as to what will maximize performance for a particular architecture, or even an automated empirical search (see Section 3.5). Following this, the developer adds simple, high-level annotations to indicate which transformations (e.g., register re-allocation or instruction shuffling) they believe will convert their modular code into the high-performance version. At this point, the transformers take over. An untrusted tool first deduces a collection of hints necessary to apply a given transformation (e.g., which permutation should be applied to reorder the instructions). Next, a verified transformer uses the hints to validate the proposed transformation, and if successful, performs it.

If unsuccessful, then the transformer indicates to the developer why it was unable to automatically perform a safe transformation.

As an example of using the workflow to verify the high-performance but ugly code `foo_ugly`, a developer first annotates `foo_ugly` with the attribute `{:codeOnly}`, which indicates that no proofs have been written (yet) about it. Next, they mark the cleaner code `foo` (which they have proven against its Hoare logic spec) with the attribute `{:transform T, foo_ugly}`. This indicates that they wish to apply the `T` transformation to map `foo` to `foo_ugly`. These top-level annotations are the only ones the developer needs to supply, and the transformers automatically recursively apply themselves to internally called Vale procedures. Vale then replaces the code and proofs of `foo` with the result of applying the transformer (i.e., the code of `foo_ugly`), as well as automatically generated proofs derived from the original proof for `foo` and the generic proof for the transformer `T`, that show that the transformed code satisfies the same pre- and post-conditions as `foo`. Thus, any callers of `foo` obtain the useful Hoare conditions for `foo`, but they also transparently obtain the higher performance of `foo_ugly`.

3.3.2 Proving a Code Transformer Correct

To support the workflow described above, we design our code transformers with an eye towards automation and one-time verification effort. In particular, we ensure that our transformers are provably guaranteed to preserve the semantics of the original code, and we structure each transformer as a combination of an untrusted front-end that *finds* the necessary transformation steps and describes them via a series of hints, and a verified back-end that checks the proposed transformation and then performs it.

We define two blocks of code to be semantically equivalent in the standard way; i.e., if and only if starting from valid equivalent initial states, both execute to equivalent final states, i.e., roughly:

```
let semantically_equivalent (c1 c2:code) =
  (forall (s1 s2:state).
     equiv_states s1 s2 ==>
       equiv_states (eval_code c1 s1)
       (eval_code c2 s2))
```

Two states are defined to be equivalent if and only if they are pairwise equal on all of their observable projections. That is, their registers are equal, values in memory are equal, flags are equal, etc. We can then define a verified code transformer as a total function that takes code (and possibly some auxiliary data, called "hints") as input and produces code that is semantically equivalent to the original code. By allowing for untrusted hints, we follow a de Bruijn structure that allows us to use arbitrarily complicated algorithms for finding transformations, without needing to prove anything about those algorithms. Additionally, by choosing the correct representation for the hints (which can be different for each transformer), we can allow for highly expressive control over the transformer.

We describe more transformers below, but as a simple example, it is easy to see why a no-op transformer (i.e., a transformer that simply returns its input) is a verified code transformer, albeit a trivial one. It may still be practically useful, however, when integrated with a more complex transformer that might fail, since the no-op transformer can be invoked on failure and hence produce an overall total transformer. Keeping the transformers total makes it easier to stay within the pre-existing Vale framework, without needing special handling for transformed code. We show error messages that may arise during a failure via an additional field added to the internal representation of procedures, which is checked upon extraction.

The code for the transformers is completely untrusted, since their results are proven against the pre-existing Vale semantics. In addition, since the transformers prove the semantic equivalence of their results, Vale's existing correctness lemmas follow simply and immediately.

Finally, as an important security precaution, we ensure that we perform the transformations *before* Vale's verified taint analyzer (which runs a dataflow analysis on the instructions to ensure that the code is free of basic digital side channels [11]) runs. As a result, the taint analyzer runs directly on the final, ugly code. Hence, the transformers do not impact the results of the side-channel analysis.

3.3.3 Example Transformers

Our framework is extensible and many transformers can be written. Here we describe three transformers we developed to support the modularization of proofs for AES-GCM: (i) the *generic peephole transformer* (particularly its instantiation for *movbe-elimination*) searches for a small pattern of instructions and replaces them with equivalent instructions; (ii) *control-flow lowering* transforms high-level if/else/while statements into low-level control-flow, and (iii) *instruction reordering* reorders instructions to improve run-time performance.

A Generic Peephole Transformer A peephole transformation searches for a small pattern of instructions and replaces them with equivalent instructions. For example, a simple peephole transformation might replace all occurrences of `mov {reg},0` with `xor {reg},{reg}`. Peephole optimizations are well studied in the compiler literature [87] and have been verified for CompCert [98]. Here, we implement a generic peephole transformer, which can safely perform such search-and-replace operations in a single pass over the code when provided with an arbitrary replacement pattern that (provably) preserves semantics. As a further convenience, the transformation recursively applies itself to all callees of that procedure too. Since such a replacement is locally semantics-preserving, and the rest of the code remains untouched, we prove that it is also globally semantics preserving.

As a concrete example, we use the peephole transformer to safely refactor pre-existing code, which relies on the `movbe` instruction, to work on older architectures. The `movbe` instruction is a recent addition to the x86-64 architecture (introduced on Atom, supported since Haswell on mainstream Intel processors). It performs an endianness change (i.e., a byte swap) while performing a `mov`. On earlier processor generations, this step is typically implemented via a `mov` and then a `bswap` instruction, which performs an in-place endianness change. OpenSSL's version of AES-GCM (and hence the verified EverCrypt version) regularly uses `movbe`, which prevents it from running on older processors that otherwise support the necessary AES extensions.

Hence, we instantiate our generic peephole transformer with a pattern to replace `movbe {dst},{src}` with `mov {dst},{src}; bswap {dst}`. This transformation takes no auxiliary data, and it can be used to automatically update code to work on processor generations without `movbe` support, simply by adding a `{:transform movbe_eliminate}` attribute to the top-level procedure.

Similarly, we instantiate the peephole transformer to allow the insertion of `prefetch` instructions, which act as processor hints to prefetch lines of data into the cache. Our automatic optimization technique (Section 3.5) uses this transformer to improve performance.

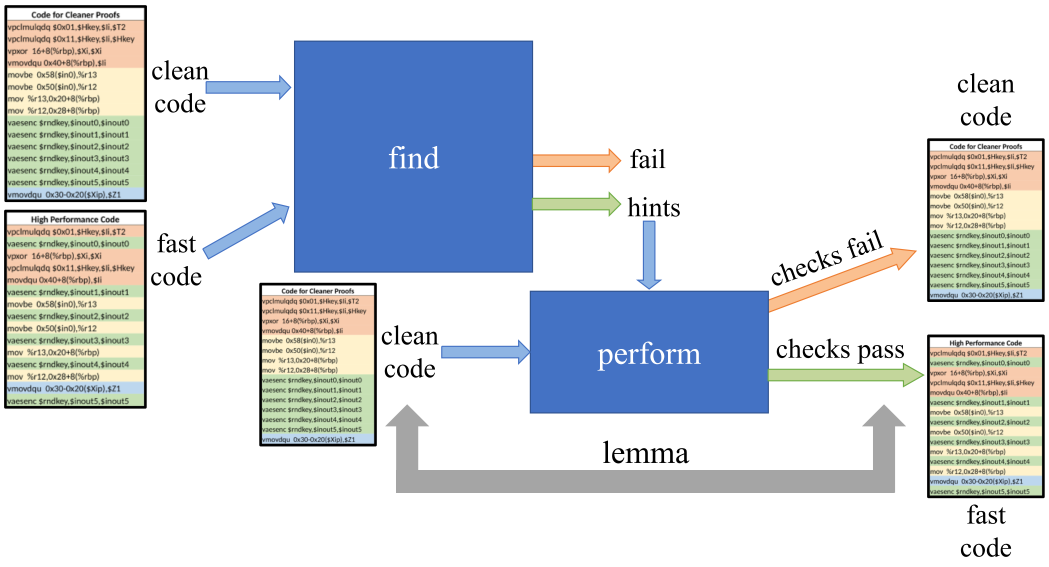
Control-Flow Lowering Since Vale is focused on cryptographic implementations, the Vale language supports only structured control flow statements (if/else and while) rather than unstructured control flow, as one might expect for assembly. Previously, this structured control flow was built directly into the operational semantics, and Vale's assembly language printer had to be trusted to correctly translate these statements into the right labels and conditional branches. To add flexibility and reduce the trusted computing base, we extend the operational semantics to support unstructured control flow. A verified transformer then translates if/else and while statements into labels and branches, in the style of certified compilation [76].

This transformer is slightly different from other transformers, in that it is applied to all Vale code, rather than only code that is explicitly user-annotated with a `{:transform ...}` declaration. Additionally, beyond the standard guarantee of semantics-preservation provided by other transformers, this transformer also guarantees preservation of digital traces, and thus strongly ensures maintenance of digital side-channel freedom, independent of whether Vale's verified taint-analyzer is run before or after the transformer.

Instruction Reordering Our most powerful workhorse transformer is the instruction reordering transform. As Figure 3.3 illustrates, this transformer takes as input two code objects (as Vale procedures), and tries to transform one into the other, as long as it is able to do so in a safe way. These code objects correspond to the verification-friendly clean modular code, and either hand-optimized code (for example, from OpenSSL), or automatically optimized candidate code (as described in Section 3.5). To do this transformation safely, the transformer feeds both code objects into an unverified `find` function, which produces hints that a verified `perform` then validates. If validation succeeds, it applies the transformation to the first code object in order to produce the second, along with a proof of semantic equivalence. If validation fails, it returns the first code object, and provides an informative error message to the developer.

The hints that are sent from `find` to `perform` are of the form "move the 12^{th} instruction to the start", "move the 5^{th} group of instructions next", etc., which taken as a whole specify a permutation of the original instructions. This supports permutation past Vale procedure boundaries, which allows the clean code to remain modular, despite the lack of clean modularity in the EverCrypt code. These hints are then validated by `perform`, which checks that these moves preserve the semantics of the code. It does so by decomposing moves into a series of swaps, and checking that each swap is allowed, performing it if so. If at any point a swap is disallowed, the transformer immediately stops any further processing, and sends a description of the failure to the user.

In building the transformer, we prove that a swap of two groups of instructions, say A and B, is semantics-preserving when the locations written to by A's instructions is strictly disjoint from the locations that are read by or written to by B, and vice-versa. That is,



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

```
High Performance Code
vpclmulqdq $0x01,$Hkey,$Ii,$T2
vaesenc $rndkey,$inout0,$inout0
vpxor  16+8(%rbp),$Xi,$Xi
vpclmulqdq $0x11,$Hkey,$Ii,$Hkey
movdqu 0x40+8(%rbp),$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(%rbp)
vaesenc $rndkey,$inout4,$inout4
mov  %r12,0x28+8(%rbp)
vmovdqu  0x30-0x20($Xip),$Z1
vaesenc $rndkey,$inout5,$inout5
```

Figure 3.3: Instruction-Reordering Transformation

there are no read-write or write-write conflicts:

$$\begin{aligned}
&(\forall (X, Y) \in \{(A, B), (B, A)\}. \\
&\quad (\forall l \in \text{writes}(X). \\
&\quad\quad (l \notin \text{reads}(Y) \land \\
&\quad\quad l \notin \text{writes}(Y))))
\end{aligned}$$

Given this proof, `perform` checks for both types of conflicts, and if the checks pass, performs the proposed swap. By proceeding through a series of such swaps, we prove that if `perform` succeeds, then it produces semantically equivalent code. Hence, in combination with `find`, we have a verified transformer that can safely reorder code from a clean, modular form into a user-chosen, performance-optimized ordering.

Unfortunately, allowing swaps only when there are no read-write or write-write conflicts disallows many reorderings that are actually safe. In particular, many instructions on x86-64 affect the flags. Changing the flags is technically a write to a location in the state, and hence any two instructions that modify the flags would be prevented from moving past one another even if the value of the flags they set was never used after their execution. This is a frequent use case in Vale, since the Vale semantics conservatively model most instructions as "havocing" the flags, i.e., setting them to an unrestricted and underspecified value. To overcome this issue, we observe that two underspecified writes are safe to exchange with each other, since their result, while different, is equivalent with respect to any value that can be observed. Thus, if we consider the value of each flag to be a ternary value (e.g., true, false, or unspecified), then we can safely and provably refine our condition for safe swaps as follows: a swap that proposes to move instruction group B in front of instruction group A is semantics-preserving if all of the locations written to by A both do not belong to the locations read by B and either do not belong to the locations that are written to by B or are a constant-write for both A and B and the same constant is written; and vice-versa. That is, write-write conflicts are allowed as long as they write the same constant value:

$$\begin{aligned}
&(\forall (X, Y) \in \{(A, B), (B, A)\}.\\
&\quad (\forall l \in \text{writes}(X).\ (l \notin \text{reads}(Y) \land\\
&\qquad (l \notin \text{writes}(Y) \lor\\
&\qquad\quad (l \in \text{constwrites}(X)\ \land\ l \in \text{constwrites}(Y) \land\\
&\qquad\quad\ \text{constwrites}(X)[l] = \text{constwrites}(Y)[l])))))\\
&\implies \text{safeswap}(A, B)
\end{aligned}$$

Note that we refer to "group of instructions" above, when performing the moves and swaps. This is because it is convenient to move certain instructions as a coherent block of code to avoid having a proposed swap be rejected by the conservative checks in `perform`. As a toy example, consider two groups of instructions `add rax, 1; adc rbx, 1` and `add rcx, 1; adc rdx, 1`. In both cases, the `add` instruction sets the carry-flag (based on its arguments), and then the add-with-carry instruction (`adc`) reads that flag as part of its addition calculation. It is easy to see that both orderings of the two groups are semantically equivalent (assuming the flags can be ignored after these instructions). However, if one were to naïvely try to change one into the other using a series of *instruction-only* swaps, then one runs into trouble. The `add`s set up the carry flag specifically for their immediately succeeding `adc`, and simple instruction-only swapping would lead to an ordering that consisted of two `add`s followed by two `adc`s, along the way to actually reordering them. However, if we consider them as groups, then they satisfy the constraints for the swap. A large portion of `find` is thus dedicated to automatically finding groups of instructions to move together, rather than individually. Note that this is where the separation between an unverified `find` and the verified `perform` shines: we can have arbitrarily complicated heuristics for finding good groups of instructions, without needing to write any proofs about the heuristics, since the transformation and hints they produce are validated.

3.4 Verifying AES-GCM

As a case study on the utility of verified code transformations, we apply them to a version of OpenSSL's implementation that was verified in prior work [116].

```
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 3.4: 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.*

3.4.1 Background on AES-GCM

AES-GCM is a cryptographic scheme for Authenticated Encryption with Authenticated Data (AEAD). In other words, it protects the secrecy and integrity of a message (e.g., the payload of a network packet) and (optionally) protects the integrity of some additional public information (e.g., a network packet's header). AES-GCM is one of the world's main tools for protecting bulk data, particularly on the Internet, where it is used for 98.9% of secure traffic [97]. In many of these settings, AES-GCM is on the critical path, since it dictates how quickly data can be read/written.

Given this ubiquity, the world has devoted considerable effort to optimizing the performance of AES-GCM, both in hardware and software. On the hardware side, in 2008, Intel introduced AES-NI [44], a collection of new instructions devoted to accelerating portions of the AES-GCM computation. Even with these hardware instructions, optimal software implementations are non-trivial. For example, OpenSSL's most optimized implementation on x64 involves over 950 SLOC of Perl scripts [155], which generate 724 SLOC of assembly. Using Perl allows the code to, for example, generate assembly for unrolled loops and customize the registers used in each unrolling. The complexity of these optimizations can lead to concrete security vulnerabilities: In 2013, a performance improvement was added to OpenSSL's codebase, passed all tests, and was on its way into the mainline code when two cryptographers noticed that the optimization would allow an attacker to forge arbitrary messages [43].

Conceptually, computing AES-GCM involves splitting the input into 128-bit blocks, computing AES counter-mode encryption on each block to produce a ciphertext, and finally computing the GCM message authentication algorithm on the resulting ciphertext and any additional authenticated data. A naïve implementation written along these conceptual lines is relatively straightforward to verify, but results in performance that is $6\times$ slower than OpenSSL's [116].

Indeed, in its drive for better performance, OpenSSL's implementation merges these conceptual operations so that it need only perform a single pass over the data. It also, when able, processes the input six blocks at a time, so as to make maximal use of available registers. Even at the block level, the individual instructions for performing AES steps are intermixed with those for performing the GCM steps as well as with memory manipulation steps (e.g., loading input data, transforming it into a suitable form to feed to AES, and storing results back to memory), presumably in an effort to keep the processor's functional units fully saturated. The GCM instructions for carryless multiplies and polynomial reductions are carefully ordered to improve parallelism, and various modular reduction steps are delayed to amortize their cost. Individual instructions themselves rely heavily on SIMD operations over 128-bit XMM registers.

As shown in Figure 3.4a, the result is Perl code that mixes, instruction-by-instruction, conceptually different operations (shown by the various colors).

Unsurprisingly, such code is far more challenging to verify. Prior work [116] relies on SMT solvers and hence is limited in how many instructions can reasonably fit into a single procedure. As a result, they divide OpenSSL's code into smaller units demarcated with Hoare-logic pre- and postconditions. Unfortunately, the intermingled nature of the code makes it difficult to decompose the code in a clean modular fashion, since at any given instruction boundary, the invariants for one conceptual step do not yet hold or have already been broken. The result is large ($\sim$3500 LoC), inelegant proofs, despite the automation provided by the SMT solver (and a custom VC generator [37]).

3.4.2 Verifying AES-GCM via Code Transformations

With the power of verified code transformers at hand, we re-write the previously verified, OpenSSL-based AES-GCM code [116], in a clean, modular fashion. Figure 3.4b shows a representative snippet, where the instructions for each conceptual operation are now grouped, and even within a group, logically similar operations (e.g., `vpclmulqdq`) are themselves grouped together. This reordering already makes the conceptual structure of the code much simpler to see and reason about (e.g., the AES instructions are now more obviously computing six 128-bit blocks in parallel, using the same round key for each block).

Furthermore, with the ability to reorder instructions, we are able to extract the three major functional steps (AES, GCM, and input manipulation) into generic procedures that utilize Vale's *operand parameters* and *inline arguments* to customize each procedure at compile time. Hence, each procedure can be customized at its invocation point to, for example, use a particular register assignment in a given round of the AES computation, as shown in Figure 3.5. This eliminates the need for custom per-round procedures and dramatically reduces the total amount of code and proofs needed.

For our case study, we applied our transformers to the inner loop of EverCrypt's AES-GCM implementation, where the proof-to-code ratio is 5.0:1 (compared to 2.6:1 in the remaining 3250 lines of proof and code). Overall, the instruction-reordering transformer enabled us to write the inner loop of AES-GCM in 450 lines of code and proof, compared with EverCrypt's version, which required 1250 lines, a nearly 3× reduction. Both versions produce the same approximately 250 lines of assembly code.

Given our clean, modular version and the original ugly EverCrypt version, the instruction-reordering transformer automatically discovered the necessary instruction permutations; the only annotation needed was to specify which transformation we desired.

We subsequently employed our peephole transformer to create custom variants of the code that can run on older platforms that do not support the `movbe` instruction. Also, using our peephole transformer, we show that the `prefetch` optimizations are indeed safe.

Finally, since we apply the control-flow lowering transform to all Vale code, we automatically obtain provably-correct unstructured code.

```
procedure Loop6x_plain(
  inline alg:algorithm, // inline argument
  inline rnd:nat, // inline argument
  ...
  out rndkey:xmm) // rndkey is an operand parameter
...
{
  Load128_buffer(rndkey, rcx, // rcx is base address
      16 * (rnd + 1) - 0x80, // offset from rcx
      ...);
  VAESNI_enc(inout0, inout0, rndkey);
  VAESNI_enc(inout1, inout1, rndkey);
  VAESNI_enc(inout2, inout2, rndkey);
  VAESNI_enc(inout3, inout3, rndkey);
  VAESNI_enc(inout4, inout4, rndkey);
  VAESNI_enc(inout5, inout5, rndkey);
...
}
...
Loop6x_plain(alg, 0, ..., xmm2);
Loop6x_plain(alg, 1, ..., xmm15);
...
Loop6x_plain(alg, 8, ..., xmm15);
```

Figure 3.5: Compile-time customization of procedures in our modular AES-GCM code

3.5 Optimizing Code for Each Processor Generation

When writing high-performance software, micro-architectural details of a given processor can influence which style of code performs best. In particular, code that is optimized for one processor generation may not perform optimally on another generation of the same processor. Nevertheless, maintaining a different version of the code for each generation of each processor is a daunting task, and hence, even OpenSSL (which supports a wide variety of CPUs with and without various extensions like SSE2, AVX, AES-NI) typically does not go to these lengths to squeeze out additional performance. With the use of verified code transformers, however, we show that we can now safely and automatically produce code that is optimized on a per-generation basis. Hence, verification enables us to reap the rewards of higher performance, in a provably safe way, with marginal extra effort.

The key observation is that having done the work to produce a verified transformer (in particular, the instruction-reordering transformer from Section 3.3.3), we can supply it with our clean modular code and *any* unverified code that produces a performance improvement. As long as the transformer accepts that code, we can safely employ it.

Hence, we develop a genetic algorithm to search for faster instruction orderings on a given processor. The algorithm takes as input an initial code object and applies a series of random mutations (shuffling of instructions) to create the first "generation" of candidates. Candidates also are allowed to mutate with a small chance to have random `prefetch` instructions added. Each candidate is sanity-checked for correctness on a single input-output test pair. Candidates that pass this fast sanity-check are then automatically benchmarked on the processor. The fastest candidates then "breed" by combining portions of their mutations (along with a small chance of new random mutations appearing), to produce a new generation of candidates. This process continues looping up to a time or generation bound provided by the developer, at which time the overall fastest candidate across all the generations is returned.

To evaluate the effectiveness of this approach, we run this algorithm on five x86-64 processors of varying generations, namely Intel's i5-2500, i7-3770, i7-7600U, and i9-9900K, and AMD's Ryzen 7 3700X. For each processor, we experiment with starting the genetic algorithm both from the original EverCrypt version of OpenSSL's hand-optimized assembly, and from our clean, modular version. We take the resulting optimized algorithm from each processor and automatically verify them using the transformers from our cleaned up modular version to confirm semantic equivalence. For the i5-2500 and i7-3770 processors, which do not support the `movbe` instruction that is used in EverCrypt, we apply the movbe-elimination transformer (Section 3.3.3) before starting optimization. We also apply it to implementations optimized for newer platforms before running them on the older non-`movbe` platforms. As additional context, we also include the (unverified) OpenSSL code that was the basis for the EverCrypt implementation; running this code on the i5-2500 and i7-3770 processors entailed manual modifications to replace `movbe` instructions. We then run all of these implementations on all of our processors.

The results of these benchmarks are shown in Table 3.1. Each row of the table corresponds to a different instruction-ordering of the code, while each column corresponds to a different processor. Each cell in the table shows the minimum number of cycles it took to encrypt 4096 bytes of data, with zero bytes of additional data, across 20 million iterations. The smallest value in each column is marked in **bold**, and represents the fastest code for that processor. As the table illustrates, each processor has an optimal ordering, and this optimal ordering can give speedups on top of state-of-the-art OpenSSL or EverCrypt code

Code \ Tested on	i5-2500	i7-3770	i7-7600U	i9-9900K	3700X
Optimized for i5-2500	**ˆ12957**	ˆ12560	ˆ2454	**ˆ2378**	ˆ3492
Optimized for i7-3770	ˆ12960	**ˆ12557**	ˆ2454	ˆ2382	ˆ3456
Optimized for i7-7600U	ˆ14340	ˆ13917	**2450**	2476	3528
Optimized for i9-9900K	ˆ14382	ˆ13941	2453	**2378**	3528
Optimized for 3700X	ˆ14127	ˆ13696	2452	2486	**3168**
Clean	ˆ13632	ˆ13222	2463	2486	3420
EverCrypt	ˆ14619	ˆ14198	2452	2474	3492
OpenSSL	*14943	*14428	2986	2980	4032

Table 3.1: Cycle counts for various reorderings on different processors. *Code with ˆ used the movbe-elimination transformer to run/optimize on older processors. Code with * denotes manual elimination of `movbe` from OpenSSL's interleaved variant.*

by up to 27% or 13% respectively.

Overall, our results show that highly targeted code implementations can give non-trivial performance improvements. Further optimizations are, of course, still possible, either via an improved automated search algorithm, or via targeted changes from performance optimization experts. Either way, the verified code transformers conveniently and automatically connect the clean, proven code with the optimized versions, ensuring that such optimizations will never violate correctness or security (unlike some previous optimizations attempts [43]).

3.6 Related Work

Although few projects have explored verified translations at the level of assembly language, verified transformations are well understood at higher levels. In particular, verified transformations are the basis for certified compilers such as CompCert [76], which applies repeated translations and optimizations to compiler intermediate languages. The VST project [5] built a Hoare logic on top of CompCert, so that Hoare logic proofs about high-level code imply properties about compiler-generated low-level code too. An example application of VST was a cryptographic primitive (SHA) [6], although compiler-generated cryptographic code often runs slower than hand-optimized assembly language code [11].

To support hand-optimized code, our reordering transformation must examine two existing versions of code and discover the relationship between them. This contrasts with typical verified compilers and optimizers, which are given just one version of the code and can then decide which code to generate.

Translation validation is a pragmatic alternative to compiler verification [114]. In contrast to the latter, which aims to verify that a compiler *always* produces the correct code, this approach verifies that a *particular* compiled code correctly implements its source program. For example, Sewell et. al. [131] use this approach to extend the verification of the source code of seL4, an operating system microkernel [60], written in C, to that of the compiled binary. The validation is based on a refinement proof between the two programs. TInA [121] takes a different approach, lifting inline assembly into C, to improve the precision of existing C analyzers, by applying translation validation to confirm that the lifted-and-recompiled code is equivalent to the original. While translation validation can leverage the general-purpose reasoning of an SMT solver, it can also suffer from the unpredictability of SMT solvers. Targeted verified translations are less general, but produce more predictable results.

Fiat-Crypto [30] and Jasmin [2, 3] both support verified translation from higher-level code to lower-level code. Jasmin uses Hoare logic at a high level; its lowest is slightly higher than assembly language, although low enough to make compilation straightforward (for example, the Jasmin compiler's register allocator never spills variables to the stack). Fiat-Crypto includes high-level domain-specific optimizations for elliptic curve cryptography, relieving the programmer of having to generate low-level optimized C code. Nevertheless, for widely used algorithms like AES-GCM, cryptography developers still consider it worthwhile to develop hand-optimized assembly code, and we believe that it is valuable to verify this hand-optimized code.

Superoptimizers [82] search for fast assembly language sequences and try to automatically establish equivalence with the original source code. However, typical superoptimizers can only generate short code sequences. Our genetic algorithm and verified transformer works on much longer instruction sequences (100s of instructions), albeit only for specific types of transformations.

3.7 Concluding Remarks

Code designed to be verified is not necessarily the same as code designed to run fast. However, some simple transformations can connect the two versions of the code. We have demonstrated such transformations both to increase confidence in existing code (in particular, OpenSSL's highly interleaved AES-GCM implementation) and to point the way towards alternate implementations that can have even higher performance on some platforms. Although the performance impact of interleaving instructions is small on the most recent Intel processors, we did find significant differences in performance both on older Intel processors and on a recent AMD processor; based on this, we speculate that such differences may be even more significant on less recent and/or non-Intel processors, such as less-powerful embedded processors.

For people focused on verification, it is heartening (and surprising) that the verification-friendly version of the AES-GCM code that we developed often outperformed the original, more interleaved code on several platforms. This suggests that developers of high-performance verified code should focus on major optimizations like domain-specific algorithm optimizations, loop unrolling and careful register allocation, and worry less about the exact sequence of instructions; this sequence may be better determined by automated search algorithms, supported by verified transformation tools.

One limitation of our current verified transformations is that writes to the heap cannot be reordered relative to other heap reads and writes. We believe that it is possible to relax this restriction. For example, we might annotate heap loads and stores with identifiers that represent disjoint regions of memory; with these annotations, a transformer can safely reorder memory operations annotated with distinct identifiers.

In summary, in this chapter we have shown that, for important ultra-high-performance software, unapologetic security is achievable—formal techniques can provide high-assurance safety, without apologizing for development velocity, and can even improve runtime performance beyond unverified approaches.

Chapter 4

Surviving Arbitrary Code

Apollo: *Nobody's expecting a miracle.*
Tyrol: *Maybe that's the problem.*

Battlestar Galactica
S02E09 “Flight of the Phoenix”

Having looked at how we can unapologetically secure a highly specific security-critical kind of software in the previous chapter, we cast the net wider in this chapter, with a study of provably-safe sandboxing. Many applications, from the Web, to third-party libraries[1] and smart contracts, need to execute untrusted code. We observe that WebAssembly (Wasm) is ideally positioned to support such applications, since it promises safety *and* performance, while serving as a compiler target for many high-level languages. 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. Hence, in this chapter, we explore two distinct approaches to provable sandboxing. Our implementation and evaluation of these two techniques indicate that safe execution of arbitrary code can be achieved, through principled and formal approaches, without compromising on either development velocity, runtime performance, or portability.

[1]A review by Google's Threat Analysis Group and Mandiant [80] of zero-day in-the-wild exploits states “Zero-day vulnerabilities in third party components and libraries were a prime attack surface in 2023, since exploiting this type of vulnerability can scale to affect more than one product”.

Contributions. The work presented here first appeared at USENIX Security'22 [13]. As lead author, the majority of this work was mine. The implementation of the vWasm simplification and register allocation phases is primarily by Wen Shih Lim.

4.1 Introduction

Lightweight, safe execution of untrusted code is valuable in many contexts where one might wish to run untrusted code that could harm its execution environment, such as third-party libraries, dynamic content delivery networks (CDNs), and more. Software sandboxing (via Software Fault Isolation, aka SFI [143]) is a well-studied technique, with a long and rich history [31, 54, 63, 86, 96, 102, 127, 159] (Section 4.2.1), to provide this crucial primitive of lightweight safe code execution. 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. Nevertheless, previous efforts to deploy it in production have failed, due to technical and marketplace hurdles (Section 4.2.2).

On the Web, after failed attempts with Java, ActiveX, Flash, NaCl [159], and Asm.js, a new contender for fast code execution was born—WebAssembly [45]. 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 [32, 38, 102, 112, 145]. With the standardization of the WebAssembly Systems Interface (WASI) [145], 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, 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 (Section 4.2.3).

However, WebAssembly's sandboxing guarantees hold only at the specification level; real Wasm implementations can, and do, have bugs (Section 4.2.3). 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 (Section 4.3) for executing Wasm code, we identify a crucial 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 (Section 4.4) 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 [76]), 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 (Section 4.2.3). This complements earlier work [63] that verifiably sandboxed Cminor, one of CompCert's intermediate languages, for CompCert's three backends.

Our second technique (Section 4.5), implemented as a compiler called rWasm, takes a different approach 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 (Section 4.6.1) 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* [136], 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 [31].

Both vWasm and rWasm are competitive in performance, with the latter providing similar performance as other performance-optimized Wasm implementations on various benchmarks. The former is faster than interpreters, but slower than unsafe compilers. We compare both implementations on qualitative aspects in Section 4.6.2, including development/maintenance effort and extensibility.

As with most sandboxing tools, we focus only on protecting the host environment from the sandboxed code. Hence, in this chapter, do not make any claims about the impact of buggy code on itself. We also assume that the environment is not corrupt or overly permissive in the APIs exposed to sandboxed code. Finally, protections from denial of service, speculative execution, and side-channel attacks are orthogonal and out of scope.

As we discuss in Section 4.7, an alternative to provably emitting sandboxed code is to *validate* that code has been properly sandboxed (cf. NaCl [159], RockSalt [96], and VeriWasm [54]). 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 [149] and Lucet [4], due to improper modeling of signedness in their specification [108].

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.

Overall, this chapter makes the following contributions.

1. An exploration of two distinct techniques to achieve provably safe, performant, multi-lingual sandboxing. We implement these as open-source tools, and evaluate them on

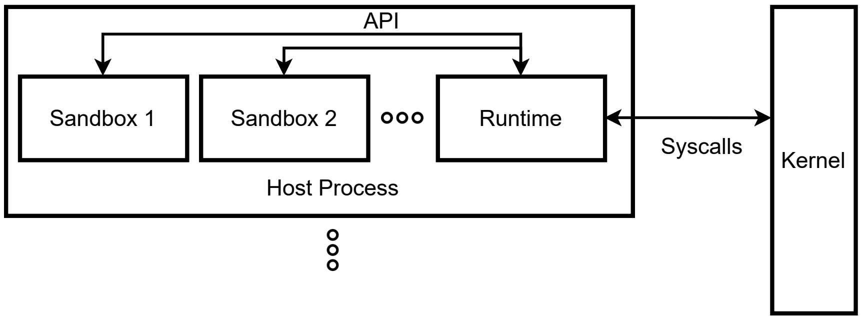


Figure 4.1: SFI-based intra-process sandboxing. *In practice, the Host Process might be a video player, and each sandbox might contain a different codec or extension.*

a collection of quantitative and qualitative metrics.

2. vWasm, the first verified sandboxing compiler for Wasm, achieved via traditional formal methods, using machine-checked proofs to provide a strong formal guarantee about the compiler's output.
3. rWasm, the first provably safe sandboxing compiler with competitive run-time performance. We achieve this using non-traditional repurposing of existing tools to provide provable guarantees without writing formal proofs.

In short, in this chapter we demonstrate that principled design allow us to achieve provably safe sandboxing of arbitrary code, without apologizing for development velocity, runtime performance, or portability.

All code in this chapter (including vWasm and rWasm) is available as open source.[2]

4.2 Wasm as a Narrow Waist for Software Sandboxing

We review software-fault isolation (Section 4.2.1) and Wasm (Section 4.2.2), before discussing Wasm's unique suitability for multi-lingual, cross-platform sandboxing (Section 4.2.3).

4.2.1 Background: Software Sandboxing

As discussed in Section 4.1, safe, lightweight execution of untrusted code is necessary in many contexts. A popular approach is Software Fault Isolation (SFI) [143], which limits the effects of bugs to the buggy code itself. Without requiring any special hardware, it

[2]`https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22`

confines 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 4.1. Inline checks before any potentially unsafe memory access enforce this boundary. The cost of these checks is offset by the performance savings from cheap transitions between the sandboxed code and the host process. Common techniques to implement these checks include restricting offsets through bit masks, explicitly checking bounds, or using hardware quirks like x64's zero-extend on 32-bit arithmetic. Ensuring these checks always run requires some form of Control Flow Integrity.

4.2.2 Background: WebAssembly

On the early Web, sites consisted of mostly static pages, with code execution on the server. Over time developers wanted better dynamic content and soon browsers regularly ran downloaded code. Multiple technologies and frameworks allowed such client-side code execution, including Java Applets, JavaScript, ActiveX, Silverlight, Flash, and NaCl [159]. However, for various reasons, e.g., insecurity (ActiveX, Flash), proprietary standards (ActiveX, Flash, Silverlight), performance issues (Flash, Java), and single-vendor support (ActiveX, Flash, NaCl, Silverlight), each became a historical artifact, except for JavaScript. Quite possibly largely as a historical accident, it became the de-facto standard for client-side code on the Internet. However, not everyone wants to run client-side code in JavaScript. One might already have an optimized library written in, say, C. Manually porting the code to JavaScript would be painful, and thus it would be preferable to compile the library to JavaScript. Alternatively, one might wish to write new code with good predictable performance, which JavaScript makes challenging.

Indeed, JavaScript makes a terrible compilation target. Especially for performance- or security-critical code, it can even be disastrous. This is because JavaScript is a high-level language, uses garbage collection, and has surprisingly "weird" semantics (such as its lack of first-class integers, thereby requiring emulation of something as ubiquitous as 64-bit integers). Since performance and security properties depend upon low-level details, reasoning about them in the compiled code also involves reasoning about the various actual implementations (and versions!) of the interpreters and/or compilers running in the user's browser.

To address these issues, Wasm introduces a new virtual architecture built from the ground up with speed, safety, and portability in mind. Its virtual architecture provides a platform-agnostic solution to compilation and code execution on the Web. It is a stack-based architecture with well-defined semantics and a basic type system, providing safer semantics than untyped assembly. However, it does not impose the full burden of more complex Typed Assembly Languages [95] to the compiler. Its semantics are entirely deterministic, except for floating-point NaNs. It does not have a garbage collector, and hence gives the developer (or compiler) more control over run-time performance. Despite being a virtual architecture, it is designed to be close to modern hardware, making reasoning about its execution much simpler. These properties make it a great compilation target.

In more detail, WebAssembly programs are composed of separate modules, each of which consists of collections of code, data, and associated connections to the environment (or other modules). Code lives in simply-typed functions that can access the module's memory and global variables. Memory is a (potentially growable) sequence of bytes, called linear memory, whose length is a multiple of 64 KiB. This memory is disjoint from all other parts of the module. Globals consist of named scalar values (i.e., no arrays). Functions can be called either directly, or indirectly by picking an offset in an indirect call table. Control flow within a function consists of conditionals, blocks, loops, direct jumps (conditional and unconditional), and indirect jumps. Wasm guarantees that all control flow is structured by only allowing jumps to labels of blocks (or loops) that enclose the jump. Indirect jumps are performed, similar to indirect calls, by picking an offset in an indirect branch table. All imports to, and exports from, the module are made explicit, avoiding implicit access to the environment.

4.2.3 Motivation: A Narrow Waist

WebAssembly's careful design enables sandboxed execution of high-performance code on the Web. However, this same design can also benefit non-Web applications, since the Wasm standard explicitly separates the core Wasm language from the specific API provided to each Wasm module by the runtime or other modules. For example, instead of offering a Web-oriented API, (say) for manipulating the DOM, many runtimes [4, 146, 147, 148, 149, 152, 153] offer the WebAssembly System Interface (WASI) [145] API to run Wasm beyond the Web.[3] Our compilers vWasm and rWasm are agnostic to the particular runtime API picked by the host.

Given its popularity, and the large number of compilers that support compilation *to*

[3]In a sense, WebAssembly could now be considered a misnomer, now that it has truly transcended beyond its original roots in the Web. Perhaps stylizing it differently might catch on: ~~Web~~Assembly.

Wasm,[4] from languages including C, C++, Rust, Java, Go, C#, PHP, Python, TypeScript, Zig, Kotlin, and more, a single compiler *from* Wasm is sufficient to immediately support sandboxed code execution for all those languages. This makes Wasm an attractive narrow waist to provide high-performance lightweight sandboxing.

Such a compiler from Wasm to (say) x86-64 is simpler in design than x64-to-x64 SFI rewriting or sandboxing. Wasm's stack-based architecture, type system, and well defined semantics all make Wasm easier to reason about than x64. It also has a drastically smaller architecture, with under 200 instructions, compared to the $\sim 1500 - 6000$ instructions in x64 [49, 51]. Additionally, it has no unexpected, or hardware/platform-specific behavior. While it *does* have minor non-determinism, in the form of computations involving floating-point NaNs, this is the only source of non-determinism, and thus its behavior is much easier to reason about.

As a narrow waist, then, Wasm seems to immediately provide high-performance lightweight sandboxing across all platforms, but note that the actual implementation of the compiler *from* Wasm is a critical part of the TCB for that guarantee. In particular, any bug in the compiler could threaten the sandboxing protections, and indeed such bugs have been found in existing runtimes, and would lead to arbitrary code execution by an adversary. For example, using carefully crafted Wasm modules, an attacker could achieve a memory-out-of-bounds read in Safari/WebKit using a logic bug (CVE-2018-4222), memory corruption in Chrome/V8 using an integer overflow bug (CVE-2018-6092), arbitrary memory read in Chrome/V8 using a parsing bug (CVE-2017-5088), arbitrary code execution in Safari/WebKit using an integer overflow bug (CVE-2021-30734); a sandbox escape in both Lucet and Wasmtime using an optimization bug (CVE-2021-32629); a memory-out-of-bounds read/write in Wasmtime (CVE-2023-26489), and many others. Recall that writing a high-performance compiler is already hard, and compilers from Wasm need to protect even against adversarial inputs, which makes it even harder. Indeed, there appears to be a tension between functionality (especially performance) and safety. We explore this in the next section.

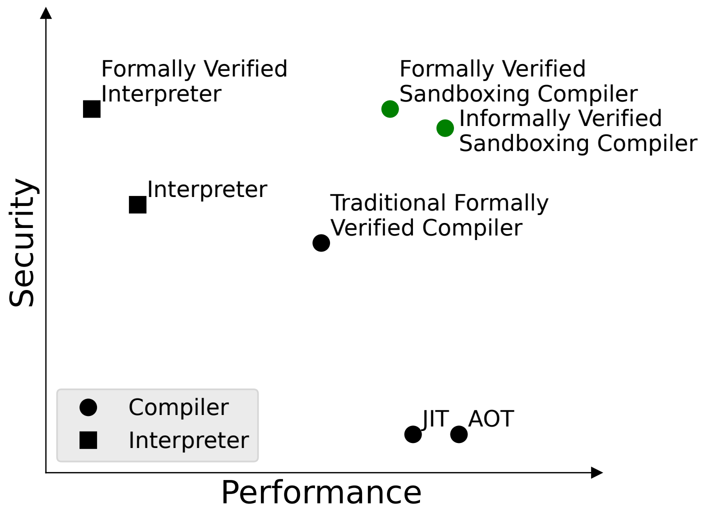


Figure 4.2: The Design Space for a Wasm-based Sandbox

4.3 The Design Space for Implementing a Wasm-based Sandbox

We consider the design space for a Wasm-based sandbox along two major axes– Security and Performance. Figure 4.2 shows an informal representation of the space, and the location of types of Wasm runtimes within it.

Security and Threat Model To describe the security of various Wasm-based sandboxes, we must first understand the threat model. For the purposes of software sandboxing, this is largely standard, but we describe it in particular for sandboxing Wasm modules. At a high level, the attacker has some control over execution of the sandboxed module (through buggy, or even malicious, attacker-provided module code) and wishes to "break out" into the environment outside the sandbox to obtain more control. Under our threat model, the attacker may start executing code from any of the explicitly exposed "starting points" within the Wasm module, can manipulate Wasm's linear memory at arbitrary points between Wasm executions, and can pass arbitrary arguments to Wasm functions when calling them. The attacker, however, does not arbitrarily control the environment, since if they already have that level of control, then they need not "break out" of the sandbox. Additionally, side-channel attacks, speculative execution attacks, hardware attacks (such as

[4]Any library or POSIX-compliant program (except for multi-threading and `longjmp`, which Wasm currently lacks) can be compiled to Wasm; our compilers support all current Wasm constructs.

Row Hammer [59] or power glitching [58]), denial of service, or excessive resource usage are out of scope. Confused deputy attacks [46], where the more privileged host process is tricked into misusing its authority due to a badly designed API, are also out of scope but addressed in other orthogonal work [102].

Performance Obviously, for critical systems, security is vital. However, for deployment in production, high performance is also important. Different workloads may have different performance requirements. Short workloads that run only once have different requirements from long running ones, or ones that require multiple runs. To simplify the discussion, within Figure 4.2, we focus on workloads that are either long running or require many runs. Our rough categorization here is supported by quantitative measurements in Section 4.6.

Design Space The top-left corner of Figure 4.2 consists of high-safety but low-performance implementations. Interpreters occupy this space, since (barring a language bug in the implementation of the interpreter), an attacker cannot escape the interpreter. However, this safety comes at the cost of run-time performance. Verified interpreters, such as Watt's verified Wasm interpreter [151], may provide better/additional safety properties relative to unverified interpreters.

The bottom-right corner consists of high-performance but low-safety implementations. Occupied by compilers, this space is used in many production scenarios, such as browsers. Since compilers are complex software, they are prone to bugs and thus can compromise safety (see Section 4.2.3). Amongst the two types of compilers, ahead-of-time (AOT) and just-in-time (JIT), the AOT compilers typically produce faster execution at run time since they can afford to spend more time on optimization. In contrast, JIT compilers tend to optimize the total execution time, including compilation time, and thus spend less time optimizing.

In the middle, we have traditional formally verified compilers [64, 76]. For most languages, traditional compiler correctness is orthogonal to sandboxing safety. The former reasons about the semantic equivalence of safe input code and its output and makes no guarantees about its compilation results when given an unsafe input program, e.g., a C program with a buffer overflow. Sandboxing safety reasons about the output code independent of the input program (or its safety).

However, Wasm is special since its semantics are (almost) deterministic. This means that *if* one formally proves this determinism, and composes that with traditional compiler correctness, then sandboxing safety could be proven from it. Thus, a traditional formally verified compiler, while safer than unverified compilers, is still imperfectly safe.

Finally, we have the upper-right quadrant of Figure 4.2, which shows our goal, where neither safety nor performance are compromised. We achieve this goal using the traditional means of formal verification for vWasm, while using a non-traditional repurposing of existing tools for rWasm. We defer the discussion of our techniques to Sections 4.4 to 4.6.

In Section 4.6, we consider additional axes, including compilation time, development complexity, portability, and extensibility.

4.4 vWasm: A Formally Verified Sandboxing Compiler

Inspired by previous successes in constructing formally verified compilers for C [76] and ML [64], we construct vWasm to compile Wasm to provably sandboxed code. Previous work primarily focused on proving that correct input code is faithfully compiled to correct output code, whereas vWasm focuses on proving that all input code (regardless of correctness or even malice) is compiled to safely sandboxed code.

Concretely, we implement vWasm in the F* proof assistant (Section 4.4.1), following a relatively standard compilation pipeline (Section 4.4.2). We formally state and prove vWasm's guarantee that all output code will be properly sandboxed (Section 4.4.3), relative to a mechanized semantics of a subset of the x64 ISA. Finally, we summarize some lessons learned from vWasm's development (Section 4.4.4).

4.4.1 Background: Formal Verification and F*

Formal verification of software consists of writing a formal statement of the property we wish to prove about the software, and then writing a formal proof that shows this statement is true. The proof is machine checked and thus provides the highest degree of assurance in its correctness. In contrast to techniques such as software testing, fuzzing, and manual reviews, formal verification is able to reason about *all* execution paths, provided *any* input. This means that behaviors like buffer overflows, use-after-frees, etc. are completely ruled out. We describe vWasm's top-level property, as well as our proof strategy, in Section 4.4.3.

Our verification tool, F* [136], is a general-purpose functional programming language with effects, built for formal verification. Syntactically, it is closest to languages from the

ML family (such as OCaml, F#, or SML). It has the full expressive power of dependent types, and has proof automation backed by Z3 [21], an SMT solver. Code written in F* can be extracted to multiple languages, and for vWasm, we use F*'s OCaml extraction. Proofs are written within vWasm as a combination of pre-/post-conditions, extrinsic lemmas, and intrinsic dependently-typed values. Also, to aid in proof management, we regularly use F*'s layered effects [120].

4.4.2 Compilation Strategy

vWasm is implemented as a compiler from Wasm to x86-64 (abbreviated as x64 henceforth), but it is designed to keep most of its code and proofs generic with respect to the target architecture. Here, we describe the process of compiling to x64, but the techniques generalize in a straightforward way to other architectures such as ARM. In compiling from Wasm to x64, there are three important conceptual stages: (i) a frontend that compiles Wasm to an architecture-parametric IR, (ii) a sandboxing pass that acts upon the architecture-parametric IR, and (iii) a printer that outputs the x64 assembly code.

The frontend for the compiler is both untrusted and unverified. This means that one neither needs to trust its correctness for the overall theorem statement to be true, nor does one need to write proofs about it. Note that this is in stark contrast with traditional compiler verification, where any stage of the compilation must either be trusted or verified. This means that we are free to use any compiler technology for the compiler's frontend, including arbitrarily complicated optimizations, as long as it outputs code within our architecture-parametric IR. This drastically reduces the development cost of such a compiler, compared to a traditional verified compiler, and it can also allow for fast code by allowing full usage of compiler optimization research. Since compiler optimization is orthogonal to our primary goal, for vWasm's frontend, we implemented only a simple register allocator and a basic peephole optimizer. We leave such optimizations for future work.

On the other end of the compilation pipeline is the x64 assembly printer, which is trusted to be correct. We discuss vWasm's overall TCB in Section 4.4.3, but we note that the printer is largely a straightforward one-to-one translation of our IR to strings, making it fairly simple to audit.

Finally, the sandboxing pass, which lies between the above two, is untrusted but verified to be correct. We define this formally in the next subsection, but informally, this means that the sandboxing code has been proven (and the proof mechanically checked) to produce safely sandboxed code, given any input. Within the sandboxing pass, all accesses (reads or writes) into the Wasm module's linear memory, indirect function call table, imports, globals, etc. are proven (sometimes after suitable transformations) to be safe.

In particular, to perform safe sandboxing, we bound accesses primarily via a bitwise-AND operation whenever possible, falling back to a check-and-branch-based bound otherwise. Admittedly, this runs counter to the WebAssembly specification, which dictates that *any* access outside linear memory must immediately trap. However, we allow for this small difference in semantics to support simplified sandboxing. For valid program executions, this does not impact their execution trace. However, upon an invalid out-of-bounds access, instead of trapping and exiting, the module may continue execution while corrupting its own memory space.[5] This transforms a security bug into a correctness (but sandbox-safe) bug. In some scenarios, this may be undesirable, so vWasm could be modified to instantly terminate.

To prove sandbox safety, we additionally prove that the sandboxing pass also guarantees (a restricted form of) Control-Flow Integrity (CFI) that ensures that any checks performed for sandboxing cannot be bypassed, and thus must be obeyed. Due to the convenient *explicit* split between different types of accesses in Wasm (e.g., linear memory is disjoint from the indirect call table and globals), sandbox checks can be safely elided in most cases except for direct linear memory accesses. This elision too is proven safe against the x64 machine model.

The sandbox used in vWasm has a fixed compile-time size, but since many Wasm programs need the ability to grow memory, we emulate the size accessible to the program, while using the sandbox size as a constant upper bound.[6] The location of the sandbox in memory itself is not fixed at compile time, but instead is chosen at run time, allowing for extra mitigations, such as Address Space Layout Randomization.

4.4.3 Provably Safe Sandboxing

Reasoning about sandboxing code involves first defining a machine model, and then defining what sandbox safety is within this model. Our machine model covers the subset

[5]Despite allowing such corruption, vWasm does *not* depend on external assumptions like write-xor-execute (W^X) to prevent code-modification attacks. Explicit checks force writes to stay within explicitly-provided data-only regions. Our proofs demonstrate these checks suffice and cannot be bypassed.

[6]Wasm uses 32-bit addressing; thus Wasm programs cannot refer to memory beyond 4GiB. vWasm supports setting the limit to this maximum, so any limitations due to memory-size bounds are due to Wasm itself. Wasm's real-world usage suggests this is not very limiting in practice.

```
type operandi =
  | OConst : n:int → operandi
  | OReg : r:regi → size:rsize → operandi
  | OMemRel : offset:maddr → operandi
  | ...

type ocmp =
  | OEq32 : o1:operandi → o2:operandi → ocmp
  | ...

type ins_t =
  | Add32 : dst:operandi → o2:operandi → ins_t
  | ...

type precode =
  | Ins : i:ins_t → next:loc → precode
  | Cmp : cond:ocmp → t:loc → f:loc → precode
  | Call : tgt:target_t → onreturn:loc → precode
  | ...

type code = list precode
type ok_t = | AllOk | MemFailure | AstFailure | ...

type state = {
  ok: ok_t; ip:loc; reg_i: int ↪ nat64;
  mem: heap; stack: stack; ...
}

let eval_step (c:code) (s:state): state = ...
```

Figure 4.3: Sample of our machine model for x64 in F*

```
val sandbox_compile
  (a:aux) (c:code) (s:erased state): Err code
    (requires (
        (s.ok = AllOk) ∧
        (reasonable_size a.sb_size s.mem) ∧
        (s.ip `in_code` c) ∧ ...))
    (ensures (fun c' →
        forall n. (eval_steps n c' s).ok = AllOk))
```

Figure 4.4: Simplified theorem statement in F* for provably safe sandboxing in vWasm

of x64 targeted by the compiler. A simplified version of this model, written in F*, is in Figure 4.3. The complete model can be found in our open-sourced code. Note that these semantics for x64 are defined as small-step semantics, allowing for reasoning about even potentially infinitely running code. Within the definition of the x64 `state`, the `ok` field is crucial for defining safe sandboxing. This field is set to the value `AllOk` if and only if, until that point in execution, nothing invalid has occurred. Crucially, this also means that no accesses outside the memory allocated to the module have occurred.

Sandboxing is safe if and only if, informally, starting from any initial `AllOk` state, executing the sandboxed code for any number of steps leads to an `AllOk` state. Figure 4.4 shows the overall statement of this theorem for the sandboxing pass, more formally written in F*. This statement, written as pre- and post-conditions for the sandboxing pass `sandbox_compile`, shows that any code output by the sandboxer is formally guaranteed via the machine-checked proof to be safe. The pass takes two arguments `a` (auxiliary data) and `c` (the input program), and a computationally-irrelevant argument `s` (the initial state of the program, which is used for reasoning in our proofs, but that is erased when running the compiler), and returns output code `c'` under the custom effect `Err` (which allows the compiler to quit early upon error, for example if it finds a call to a non-existent function). The statement guarantees that as long as the pre-conditions in the `requires` clause are satisfied, the post-condition in the `ensures` clause provably holds on the produced output code. The conditions say that the initial state must be safe, have a reasonable sandbox size, and start from a valid location in the code; if these conditions are met, the output code `c'` will be safe when executed for any number of steps `n`.

The safety of the code means that it cannot access memory outside the allowed boundary. In particular, this means that any invalid (or even malicious) code provided to the sandboxer is made safe, and thus issues such as buffer overflows can only corrupt the module's internal state and nothing outside it. This property is distinct from traditional compiler correctness, which is predicated upon the input code being safe.

To better understand this guarantee, we must understand the Trusted Computing Base (TCB) under which it holds, which then explains potential limitations. Since this proof is written in F*, our TCB includes all that comes with it, specifically, F*, Z3 (its SMT backend), and OCaml (which is what we extract the F* code to, in order to execute). Additionally, our TCB includes the x64 machine model, the full theorem statement from Figure 4.4, and the trusted x64 assembly printer, as well as the assembler that converts the printed assembly into machine code. Note that all of these portions of the TCB would be required even for the implementation of a traditional verified compiler, but crucially, we do not have the semantics of the input language in our TCB, while traditional compiler correctness would necessarily need it in order to state its semantic equivalence (or simulation) theorem.

Our proof strategy consists of multiple parts. First, we use small-step semantics, which allow for more fine-grained reasoning about code execution. Next, we combine some instructions into groups of instructions. A group of instructions can be any positive number of instructions executed in a straight-line fashion. We then show a CFI property that this grouping cannot be escaped from; i.e., a group can only be entered at its start, and only exited at its end, thereby behaving similar to a (subset of a) basic block. For normal control flow (such as straight-line code, or for conditional/unconditional direct jumps) this is straightforward, but more effort is required for indirect control flow (indirect function calls, indirect jumps, lookup/jump tables, and function returns). Next, we show that each group maintains the `AllOk` invariant; i.e., no invalid memory accesses or any other transition to an invalid state has occurred yet. By ensuring that all potentially unsafe accesses are correctly checked within a group, this invariant follows as a result. Finally, we prove that this invariant suffices to show that executing any number of steps in the program maintains `AllOk`.

4.4.4 Useful Insights

In the process of implementing and verifying vWasm, we uncovered some useful insights, which are likely to generalize to any formally verified sandboxing compiler. The first of these is that proving sandboxing is far more convenient with small-step semantics than it is to do it with big-step semantics along with a full execution trace. We initially started with the latter, but found that the proofs were convoluted and were simplified significantly by switching to the former. This is because small-step semantics allowed us to more naturally state and work with the invariant along groups of instructions. At its core, each group of instructions behaved like a single "medium step" formed from a finite collection of small steps, and this property is more naturally expressed as combining small steps on an unchanged program, rather than as a big step on the restricted program consisting of only the group.

Next, we found, to our surprise, that an unstructured IR was more convenient to work with to prove sandboxing. Wasm is carefully designed as a structured virtual architecture, not allowing for arbitrary `goto`s, and such structure generally simplifies proofs. However, Wasm introduces multiple distinct control-flow constructs consisting of loops, blocks, conditionals, branches, conditional branches, indirect branches, and direct/indirect calls. Each of these contribute non-trivial complexity to the proof, and hence we found that making the IR unstructured quite early in the compiler simplifies the overall implementation and associated proofs. In particular, vWasm quickly moves to an IR based on an unstructured Control Flow Graph (CFG) and maintains this right down to emitting x64 assembly at the printer. This means that all the control flow constructs are unified and do not require special handling.

Another useful property we noticed is that most of the proofs for the sandboxing are architecture independent. While we have implemented our compiler to only emit x64, very few proofs depend on x64-specific behavior. Instead, most of them rely more abstractly upon whether an instruction may access a certain region of memory or not. Thus, we believe that the compiler and its proofs are *almost* architecture independent. Specifically, with some more proof engineering, location modeling could be made more abstract, similar to that done by Bosamiya et al. [12], at which point the proofs would be practically architecture independent. We leave this for future work.

Finally, to implement a convenient-to-reason-about CFG-based semantics, a list of instructions with jumps indexing into it is useful. However, this makes for a very slow compiler, since updates to a functional list are expensive in languages like OCaml that use immutable linked lists. In order to balance proof complexity and implementation performance, we implemented a new functional data structure, which we call an Append Optimized List (AOL). An AOL contains all the operations that one might require from a functional list, and each of these operations are proven to correspond to the equivalent operation on the simulated functional list. However, these operations are optimized for performance by implementing AOL as a tree that allows for delaying operations. For example, appending two functional lists is a linear time operation (in the length of the first list), but appending two AOLs is a constant-time operation, since it only requires the creation of a single node that points to both. Other operations that are faster on AOLs than on functional lists include: `length`, `zip`, `unzip`, `repeat`, `split`, `get_at_index`, and `update_at_index`. Since these operations are proven to match the functional specification, proofs written with lists in mind work directly with no changes required; however, the overall performance of the compiler is improved by an order of magnitude (Section 4.6.1.2).

4.5 rWasm: High-Performance Informally-Proven-Safe Sandboxing

rWasm leverages the insight that Rust (Section 4.5.1) can provide *both* safety (Section 4.5.3) and good performance, if we employ a suitable compilation strategy (Section 4.5.2) that plays to the strengths of Rust's optimization strategies (Section 4.5.4). rWasm's approach also makes it surprisingly simple to instrument the produced code with reference monitors (Section 4.5.5).

4.5.1 Background: Rust

Rust [83, 139] is a systems programming language with a strong focus on performance, reliability, and safety. Developed originally for use in Firefox, it has since gained use in the industry for security- and safety-critical software components. Amongst other goals, Rust aims to eliminate memory safety errors entirely, and it does so through a memory-ownership discipline. This provides safety without garbage collection, which practically all popular memory-safe languages previously required. Rust allows a developer to write high-level code, with low-level control when needed. However, as a systems programming language, there might be certain scenarios (e.g., writing an OS) where one might need more control than directly allowed by the language, like directly accessing an arbitrary location in memory. Rust provides an explicit escape hatch for this via the keyword `unsafe`.

However, Rust guarantees that correctly typed code without `unsafe` (ensured by the declaration `#![forbid(unsafe)]`) will always be memory safe. The Rust community takes this guarantee *extremely* seriously, and considers any unsoundness in its type system to be immediately security critical (even if it may not actually be exploitable), assigns a CVE to it, and then works to fix the unsoundness [157]. Given the prevalence of Rust in industry, and how seriously the Rust team takes unsoundness bugs, safe Rust is thus battle-tested to be memory safe, even if not (yet) proven to be so. Early efforts towards formalization of Rust and its security guarantees have already begun, such as with the RustBelt [56] project and Oxide [154].

4.5.2 Compilation Strategy

rWasm compiles WebAssembly code to safe Rust. It consists of (i) a frontend that parses the Wasm binary into an internal intermediate representation, (ii) a stack and dead code analyzer, and (iii) a backend printer that prints the intermediate representation into Rust code. This Rust code is then fed into the Rust compiler (`rustc`) to produce machine code.

We implement all stages of rWasm in safe Rust, and no stage of the compiler needs to be verified or trusted. This means we do not need to depend upon the safety or correctness of any part of rWasm for the safety of the produced executable machine code. Instead, the safety of the produced code simply comes from the lack of any `unsafe` in the generated Rust code. We discuss this further in Section 4.5.3. The compiler itself is thus free to generate code however it likes, but not all approaches produce efficient code. We discuss techniques to produce efficient code, amongst other details in Section 4.5.4.

4.5.3 Provably Safe Sandboxing

Our key insight for rWasm is that emulation of low-level code can be done by lifting it to a high-level language, which provides the guarantees of the high-level language to the low-level code under emulation. The high-level property that we want for provably safe sandboxing is essentially memory safety, meaning that the sandboxed code cannot access any memory that is not explicitly allocated to it by the runtime in the host process. Thus, compiling Wasm to *any* type-safe language (e.g., OCaml or Haskell) would provide provably-safe sandboxing through guaranteed object integrity, lack of type confusion, lack of out-of-bound memory accesses, etc. without trusting the generated type-safe code. Contrast this with `wasm2c` [146], which requires trusting the compiler, or its generated C code, since C does not guarantee memory safety.

The usual side effect of such lifting is that it leads to either unpredictable or bad performance. We recognize, however, that Rust presents us with a new opportunity to provide high-level guarantees without necessarily suffering this side effect. We have to be careful to actually eliminate it, but due to Rust's focus on performance, and lack of a garbage collector, it is feasible to obtain good and predictable performance by lifting code to Rust.

Since safe Rust's type-safety guarantees memory safety, we can thus informally prove memory safety by ensuring that there is no usage of `unsafe` in the produced code. We can achieve this simply through the use of a `#![forbid(unsafe)]` declaration. The TCB in this scenario then is only the Rust compiler. More explicitly, neither rWasm nor its generated code needs to be trusted for memory safety. While the Rust compiler itself is large, the Rust team takes type- and memory-safety extremely seriously, and thus this provides (informally) provably safe sandboxing.

Astute readers will note that sandbox safety in any type-safe language also depends on the language's runtime libraries. Fortunately, rWasm imports nothing, uses only allocation-related features (for `Vec`), and even eliminates dependency on the Rust standard library via the `#![no_std]` directive. As with any sandbox, care is required when exposing an API to sandboxed code [102] (e.g., to avoid APIs enabling sandbox bypasses directly or via confused-deputies), but such concerns are orthogonal to sandbox construction.

4.5.4 Useful Insights

Here, we describe an optimization-friendly collection of techniques to preserve WebAssembly semantics in Rust. Figure 4.5 illustrates the net effect of these techniques when compiling an example Wasm function that takes a 32-bit integer argument and computes the sum of positive integers up to that number. Note that the simple Wasm code becomes seemingly complex Rust code. However, the Rust code is written to be optimization friendly, and thus the Rust compiler is able to optimize it nearly all away, even recognizing that this convoluted looking code can be optimized via the mathematical closed form $\sum_{i=0}^{a} i = \frac{a(a+1)}{2}$. We briefly summarize key design decisions below.

The first challenge for any Wasm compiler is mapping Wasm's stack-based virtual machine to finite-register hardware. Rather than write our own register allocator for rWasm, our stack analysis pass emulates the stack-based machine via an infinite-register machine, where Rust variables are used to represent the "infinite" registers (notice in Figure 4.5a that the Wasm code only uses two stack slots, which correspond to Rust variables `slot0` and `slot1` in Figure 4.5b). We represent Wasm's non-stack locals and function arguments as scalar-typed Rust variables (see `local0` and `local1` in Figure 4.5b). This allows us to piggy-back on `rustc`'s excellent register allocation routines to obtain good performance.

To further simplify rWasm's implementation, we observe that Rust is adept at eliminating the repeated wrapping and unwrapping of tagged unions (i.e., sum types) without any performance penalty. Hence, we implement a custom tagged `enum` in Rust that can hold any of the native Wasm types. As shown in Figure 4.5b, `tv` wraps and tags values of any scalar type into the `TV` tagged `enum`; the original values can then be extracted back to a scalar type, say `i32`, while checking the tag using `vi32()?`. Using a tagged `enum` means that all

```
(func (param i32) (result i32)
  (local i32)
  loop (result i32) ;; []
    local.get 0      ;; [a]
    i32.const 1      ;; [a, 1]
    i32.lt_s         ;; [a<1?]
    if (result i32)  ;; []
      local.get 1    ;; [s], return
    else
      local.get 0    ;; [a]
      local.get 1    ;; [a, s]
      i32.add        ;; [a+s]
      local.set 1    ;; [], s ← a + s
      local.get 0    ;; [a]
      i32.const 1    ;; [a, 1]
      i32.sub        ;; [a-1]
      local.set 0    ;; [], a ← a - 1
      br 1           ;; continue
    end
  end)
```

(a) WebAssembly code

```
fn func_0(&mut self, a: i32) -> Option<i32> {
 let mut local0 = a; let mut local1 = 0i32;
 let mut slot0: TV;  let mut slot1: TV;
 'lbl0: loop {
  slot0 = tv(local0);
  slot1 = tv(1i32);
  slot0 = tv(slot0.vi32()? < slot1.vi32()?);
  'lbl1: loop {
   if slot0.vi32()? != 0 {
    slot0 = tv(local1);
   } else {
    slot0 = tv(local0);
    slot1 = tv(local1);
    slot0 = tv(slot0.vi32()? + slot1.vi32()?);
    local1 = slot0.vi32()?;
    slot0 = tv(local0);
    slot1 = tv(1i32);
    slot0 = tv(slot0.vi32()? - slot1.vi32()?);
    local0 = slot0.vi32()?;
    continue 'lbl0;
   } break;
  } break;
 }
 Some(slot0.vi32()?)
}
```

(b) Simplified Rust Output from rWasm

```
func_0:
    test    esi, esi
    jle     .A
    lea     eax, [rsi - 1]
    lea     ecx, [rsi - 2]
    imul    rcx, rax
    mov     edx, eax
    imul    edx, eax
    shr     rcx
    add     edx, esi
    sub     edx, ecx
    mov     eax, 1
    ret
.A:
    xor     edx, edx
    mov     eax, 1
    ret
```

(c) Compiled to x64

Figure 4.5: Example compilation of a program with rWasm. *The WebAssembly program implements the sum of the first* **a** *positive integers. The status of the stack after each instruction's execution is shown as comments, where the stack grows towards the right.*

Rust variables have the same type, so rWasm's stack analysis only needs to track the overall stack size, not the types of the values on the stack at any given moment during program execution. For example, given the Wasm code, `i32.const 5, drop, f64.const 3.14`, rWasm can reuse the same Rust variable, even though the stack slot has different types during execution. Furthermore, this approach makes the polymorphic Wasm instructions `select` (which picks one of the elements of the stack based upon the top element) and `drop` (which drops the top element of the stack) trivial to implement.

To handle WebAssembly's wide range of control flow constructs, we need to compile them down to those supported by Rust. While Rust does not support unstructured `goto`s, it does have the ability to `break`/`continue` to any outer loop (the labels `'lbl0` and `'lbl1` in Figure 4.5b correspond to the respective block structure in Figure 4.5a). We use this, along with conditionals and `match` expressions to implement and emulate all the intra-function control flow constructs in Wasm. Direct function calls translate trivially to direct calls in Rust, but indirect function calls have multiple design choices, such as inlining a dispatch routine, or having multiple type-disjoint dispatch routines, or calling out to a single common dispatch routine (requiring serialization of arguments on the stack and a type check and deserialization at the dispatch routine). In practice, we found the single dispatch routine to work best, for both compile-time and run-time performance.

For WebAssembly's linear memory, there are multiple design choices both for how to implement it, as well as how to access it. The first decision is whether to implement it as an overcommitted allocation with memory-size emulation (similar to that in vWasm) or to simply utilize a heap-allocated resizable array of bytes (i.e., `Vec<u8>`). The second decision is a choice between check-and-panic vs wrapping memory access. Each of these choices involves a trade-off between compile-time and run-time performance. Based on our microbenchmarks (Section 4.6.1), we chose check-and-panic with a resizable `Vec<u8>` as our default. While this introduces explicit checks at each access, `rustc` optimizes many of them by eliminating repeated or unnecessary checks statically.

Wasm's mutable global variables might initially seem difficult to implement in Rust, since Rust requires `unsafe` to read or write mutable globals (or `static mut` in Rust parlance). However, this has a simple fix familiar to most functional programmers: the state monad. In fact, rWasm even handles Wasm's linear memory this way. In particular, we represent the emulated Wasm module as a Rust `struct` whose associated methods emulate the Wasm module's functions. This means that the module's state is passed into each function (via `&mut self`), and the globals and linear memory can be stored safely within the Rust `struct`.

Finally, rWasm models integer overflow semantics to explicitly match WebAssembly semantics. By default in Rust, integer overflows cause panics in debug builds, and wrap in release builds. Since Wasm's integer overflow semantics are to always wrap, we explicitly perform wrapping arithmetic in Rust. This ensures that we always match WebAssembly semantics, even in debug builds. We omit this in Figure 4.5b due to space constraints.

4.5.5 Extensions

Implementing a low-level (virtual) architecture emulator via lifting to a high-level language, as we do in rWasm, comes with some extra benefits. One such benefit is that it is easy to build code tracers and Inline Reference Monitors (IRMs) in the spirit of SASI [31] (which describes how to instrument Java and x86 byte code to enforce security policies expressed as security automata). Another benefit is that the Rust compiler is able to jointly optimize the IRM and Wasm module's code, since both are part of the same generated Rust source. In fact, one could even consider the sandboxing access checks to be a special case of such IRMs. Within rWasm, we currently have multiple tracers that can be enabled if the user chooses, including function-level tracing, instruction-level tracing, and memory-access tracing, taking 75, 10, and 70 lines of code respectively to implement. Anecdotally, we found it easy to debug rWasm and its output during development due to IRMs. It would not be difficult to extend rWasm with other IRMs, even with very high precision, such as byte-level granularity run-time taint analyzers. Such extensions could potentially allow one to implement various sanitizers, such as AddressSanitizer (ASan) [129], without introducing much overhead, or indeed even needing source (which compiling with ASan requires). We leave such extensions for future work.

4.6 Evaluation

We evaluate vWasm and rWasm against multiple popular Wasm runtimes. These include interpreters (wasm3 [147] and WAMR [153] in interpreter mode), JIT compilers (Wasmer [148] and wasmtime [149]), and AOT compilers (wasm2c [146], WAMR [153] in AOT compilation mode, WAVM [152]). We choose these runtimes for comparison as they are both popular, as per GitHub stars, and also support the WebAssembly System Interface (WASI) [145], allowing for direct comparisons. We also investigated Lucet [4], which uses Wasm for sandboxing, although without any guarantees.[7] Unfortunately, despite extensive efforts

[7]Indeed, one of the developers remarked, "We're just constantly fixing bugs with it" [88].

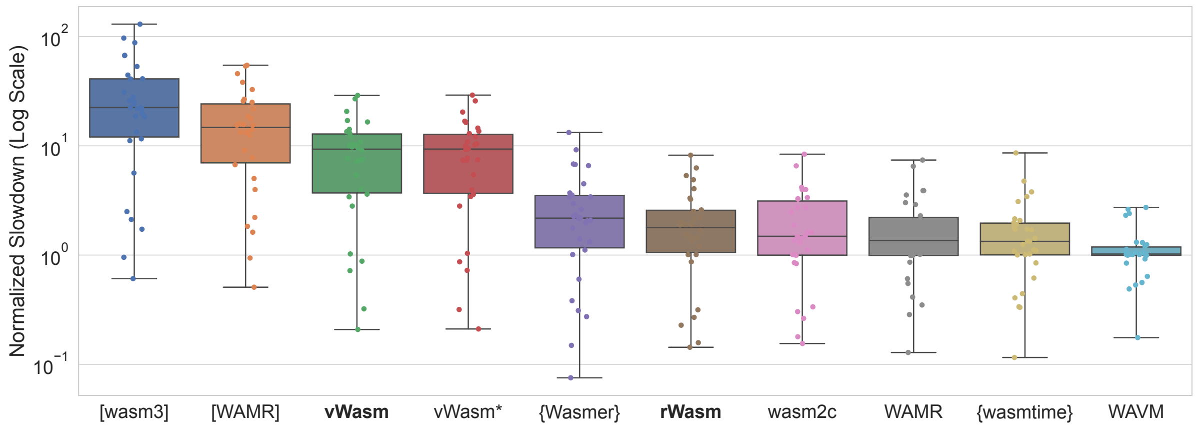


Figure 4.6: Runtime performance of Wasm runtimes. *Mean execution time of PolyBench-C benchmarks across the Wasm runtimes, normalized to pure native execution. Interpreters have square brackets; JIT compilers have braces; the rest are AOT compilers. vWasm* disables sandboxing.*

with multiple versions, we were unable to get Lucet to execute any of our 30 benchmarks, and thus we exclude it from comparison.

We evaluate these Wasm runtimes on both quantitative (Section 4.6.1) and qualitative (Section 4.6.2) metrics. All benchmarks run on a system with an AMD Ryzen 3700x processor and 64 GB of memory. We compile benchmarks to WASI-compliant Wasm binaries using Clang [18] with the `-O3` optimization level. In addition, for comparison against native (non-sandboxed) execution, we compile directly to native x64 code also using Clang with `-O3`.

4.6.1 Quantitative Benchmarks

4.6.1.1 Execution Time

As discussed in Section 4.3, run-time performance is critical for practical adoption in most applications. Hence, we measure execution time for our compilers and our various baselines using the PolyBench-C benchmark [115] suite, consisting of thirty programs, which has been a standard benchmark suite for Wasm since its inception [45].

Figure 4.6 summarizes our results, showing the normalized execution time of the benchmarks on the Wasm runtimes. Each point in the chart is the ratio of the mean time taken to execute the benchmark with the particular runtime vs. the mean time taken to execute by compiling the C code directly to non-sandboxed x64. We run each benchmark between 10 and 1000 times, based upon the time taken to run the particular benchmark. The mean of the different normalized execution time, and the 25% and 75% quartiles are shown for each runtime. Figure 4.7 shows a detailed breakdown and further analysis.

The results indicate that, unsurprisingly, compilation strictly dominates interpretation for run-time performance. Note that, as seen in the original Wasm paper [45], we too find some benchmarks execute *faster* when compiled via Wasm than when compiled directly to native code.

With respect to our compilers, we see that 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 (such as `2mm`, `3mm`, and `gemm`), rWasm is more than twice as fast than WAVM, and thus we see that relative performance can vary drastically based upon workload. vWasm consistently outperforms the interpreters on all benchmarks (by 30% on benchmarks like `reg_detect` and `fdtd-apml` to 600% on benchmarks like `cholesky` and `ludcmp`). However, while on average it is 2× to 3× faster than the interpreters, it is slower than the other compilers by 3.5× to 7.5×. Figure 4.6 also shows the execution time for vWasm with the sandboxing pass disabled. We find that the run time is marginally affected (by only 0.2%). This indicates that almost all of the slowdown, compared to other compilers, is due to the unverified portion of the compiler, which can be improved without needing to write any new proofs or even impacting existing proofs. In particular, replacing the simple register allocator and introducing standard optimizations (e.g., common subexpression elimination) should improve the performance of code compiled by vWasm significantly, without any proof effort.

Microbenchmark: Sandboxing Memory Accesses There are multiple design choices for how memory accesses are implemented (Section 4.5.4). We compare these in Figure 4.9, both on our regular AMD-based test bench, and on a system with an Intel i9-9900K with 128GB of RAM. These violin plots show the time taken to read a single 64-bit integer from memory, using different methods to confirm the read's memory safety. We represent the three approaches, namely No Sandbox (N), Check-and-Bound (C), and Wrapping with Bitwise AND (B), as rows/violins in the plot. N is shown only as an unsafe baseline to compare the safely sandboxing C and B. In each violin, the top half shows the probability density of time taken (collected from $\sim 2 \times 10^9$ samples for each configuration) to perform a

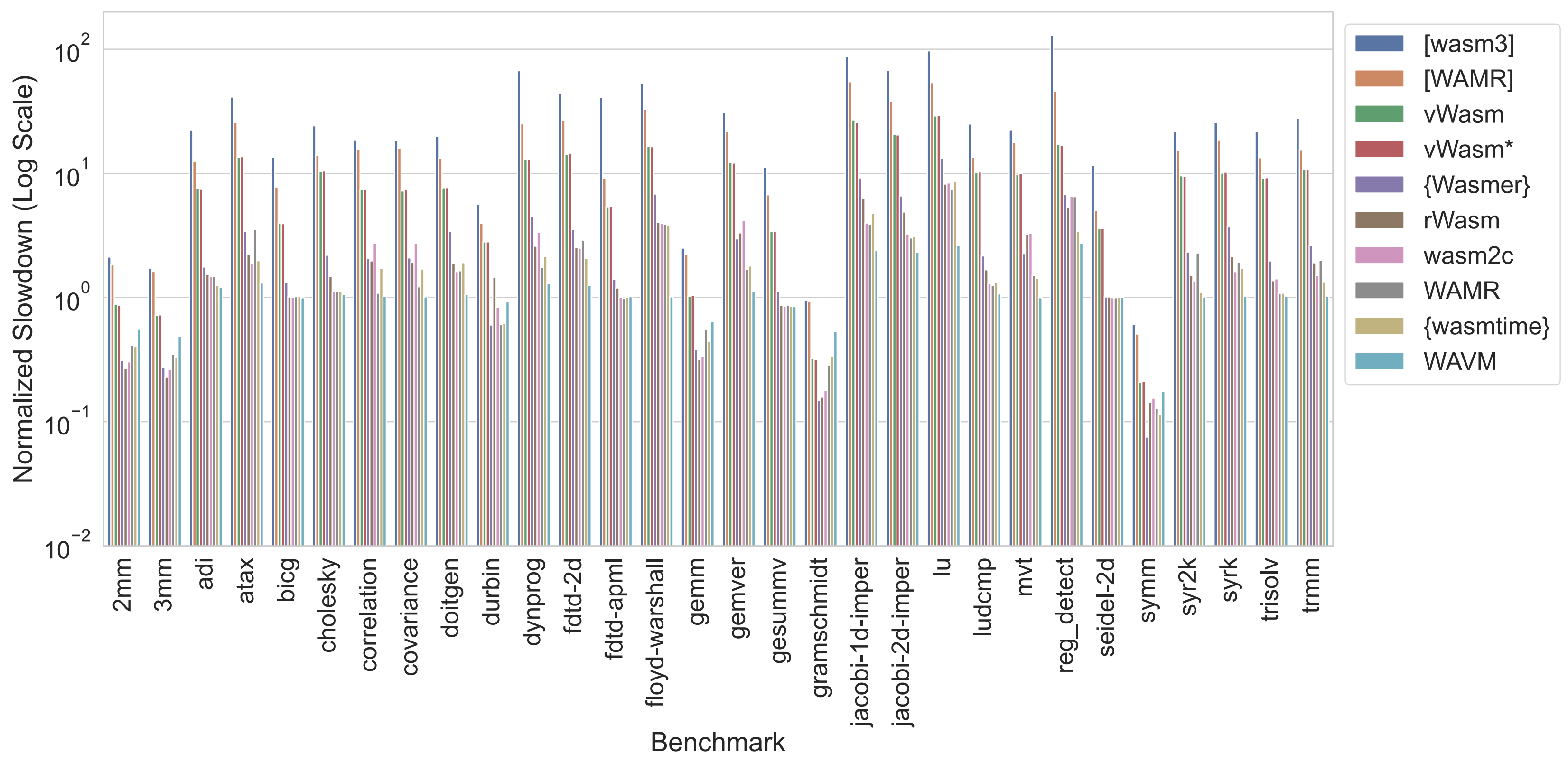


Figure 4.7: Detailed runtime performance of Wasm runtimes. *Per-benchmark breakdown of the mean execution time of PolyBench-C benchmarks across the WebAssembly runtimes, normalized to pure native execution. Interpreters have square brackets; JIT compilers have braces; the rest are AOT compilers. vWasm* disables sandboxing. All benchmarks are compiled with PolyBench-C's own internal execution time reporting (i.e.,* `-DPOLYBENCH_TIME`*), rather than relying on less accurate external measurements using* `time(1)` *or similar. Note how, for example, all interpreters and compilers on* `lu`*/*`symm` *perform strictly worse/better than native. Such differences have been seen in the past [45], and seem connected to how well compilation to Wasm goes, as well as the memory access patterns of each benchmark.*

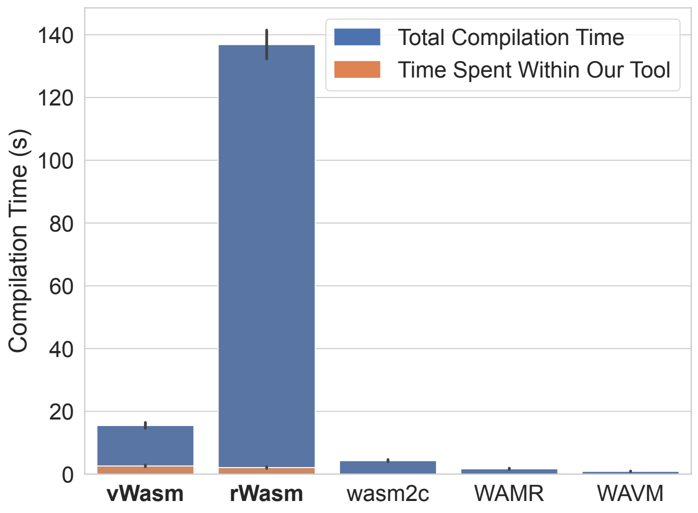


Figure 4.8: Compilation speed of Wasm compilers. *Mean compilation time for the PolyBench-C benchmarks across the Wasm AOT compilers.*

single read from resizable memory (via `Vec<u8>`), while the bottom half shows the same from a fixed-size array. On both CPUs, for fixed-sized arrays, the difference between C and B is negligible. However, on resizable memory, we find that C is three times faster than B on the AMD CPU, while on the Intel, B is only marginally faster than C. Thus, we find a significant difference in picking the better approach on modern Intel and AMD CPUs. More surprisingly, C either almost meets, or significantly beats the performance B, contrary to conventional SFI wisdom.

4.6.1.2 Compilation Time

For some scenarios (e.g., Web apps), compilation time matters, since it sits on the critical path for an impatient user. For other scenarios (e.g., installing dynamic client code at a CDN), compilation time happens off the critical path.

For completeness, Figure 4.8 shows the mean compilation time needed for each of the Wasm compilers. For our compilers, we also show the split between time spent within our implementation, and time spent in the tool that runs after.

For vWasm, approximately 17% of the 16 seconds is spent within the compiler, and the rest is spent in the assembler. For rWasm, 1.5% of the 137 seconds is spent within our compiler, and the rest is spent within the Rust compiler, which is known for slow performance. We note however, that any improvements in the `rustc`'s compilation time

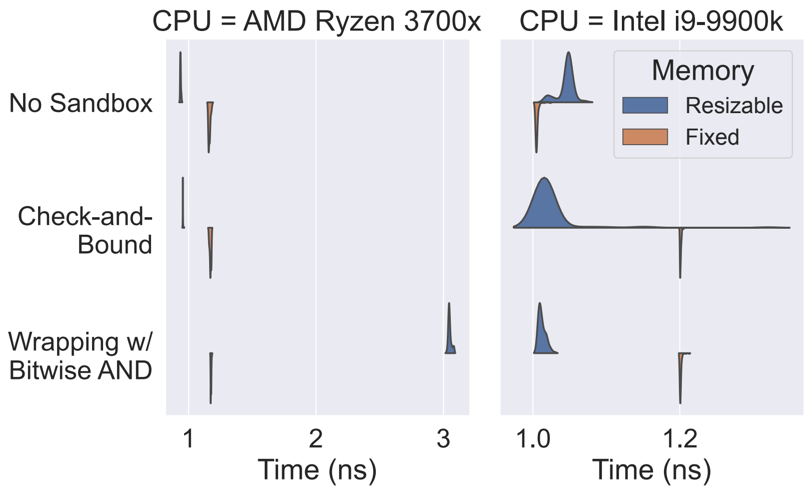


Figure 4.9: Design choices for sandboxing memory accesses

will automatically improve our overall compilation time without requiring any changes to rWasm.

Faster Compilation with Append Optimized Lists As discussed in Section 4.4.4, we introduce an efficient implementation for functional lists with operations proven functionally equivalent to standard functional lists. When measured on the compilation times for PolyBench-C, they reduce compile times from 25+ seconds to 2.5s, which is an order of magnitude improvement. The impact on verification time and effort (apart from proving the AOL functionally correct) is negligible, since AOLs provably meet the specification for standard functional lists, meaning that they behave as a “free” drop-in replacement.

4.6.1.3 Development Effort

Next, we quantify the development effort needed to implement both vWasm and rWasm. The former took approximately two person-years to develop, including both code and proofs, while the latter took one person-month. This stark contrast is a testament to the daunting amount of work formal verification requires, even with modern, automated tools like F*. It also illustrates the significant benefit of rWasm’s carefully leveraging Rust’s investment in safety. We describe the development effort qualitatively in Section 4.6.2.

As another quantitative measure, we include lines of code for both tools in Table 4.1, split by high-level components, along with total time taken for F* to verify the components

vWasm Component	Lines of Code	Verif. Time (s)
x64 Semantics	2068	114
Printer	1458	45
Parser	558	19
Frontend	2747	57
Register Allocator	1822	87
Optimizer	185	8
Sandboxing	3607	450
AOL	737	9
Layered Effects	443	8
Misc	1160	22

rWasm Component	Lines of Code
AST + IR	384
Parser	888
Stack Analysis + Printer	2157
IRMs	155
Misc	109

Table 4.1: Development Effort. *The first two components of vWasm are its primary TCB.*

of vWasm. We note however, that this only shows overall time taken to verify; it is not indicative of the interactive verification cycle, which is what comprises the majority of vWasm's development time. To keep the interactive proof engineering cycle tolerable, most proofs in our code base take under ten seconds to verify, and even the most time consuming proofs are checked in under two minutes.

4.6.2 Qualitative Evaluation

Here, we qualitatively compare vWasm and rWasm against one another, as two important points in the design space (Section 4.3). We summarize the comparison in Table 4.2.

Safety The level of assurance provided by both vWasm and rWasm is extremely high, since both provide provable safety. However, only the former is formally verified and reasons directly about the generated assembly code. Thus, one might argue that it is theoretically safer. From a more practical view, we need to consider their respective TCBs to understand

Property	vWasm	rWasm
Safety Guarantee	Theoretically stronger	Provable w/ non-standard TCB
Initial Implementation	Less efficient	More efficient
Maintenance	Less efficient	More efficient
Static Properties	More extensible	Less extensible
Run-Time IRMs	Less extensible	More extensible

Table 4.2: Summary of the Qualitative Comparisons

safety. The TCB for vWasm is standard in many verification papers, in that it includes the verification tool (and its dependencies) as well as the model we are verifying against (here, the x64 machine model). For rWasm, the TCB is non-standard since it includes trusting the compiler of a language that is only a decade old, but is usually not part of verification. However, Rust is committed to memory safety, and is trusted for many security-critical applications in the industry, and thus may be considered safe enough. The decision for which to pick, purely based upon safety, thus relies on which TCB one considers more trustworthy, since the two are practically disjoint, and thus not directly comparable.

As a sanity check, we also confirm that exploit attempts are caught. Specifically, we implement an end-to-end image conversion scenario using netpbm 10.26 (vulnerable to CVE-2008-0554) and libjpeg-turbo 2.1.1. We compile them to separate Wasm modules, and use them to convert GIFs to JPEGs via a trusted driver program that hosts them as separately sandboxed libraries in the same process. On an example input, the rWasm-compiled and vWasm-compiled versions show a mean slowdown of $1.301\times$ and $3.825\times$ across 100 executions respectively, compared to the equivalent native program without Wasm-based compilation and sandboxing. We also test the proof-of-concept for CVE-2008-0554 on all three versions. This causes the native version to a crash with attacker-controlled state (potentially leading to arbitrary code execution). In contrast, unsurprisingly, both vWasm- and rWasm-compiled versions are able to successfully detect the buffer overflow and terminate the module's execution, returning back safely into the driver program.

Development Effort: Initial Implementation As noted in Section 4.6.1, vWasm took much longer to implement than rWasm. The reasons for this are many fold: (i) developing verified software continues to be significantly more difficult than unverified software; (ii) a full compiler down to assembly is more complicated than one to a high-level language, due to low-level architectural details; (iii) a compiler to a high-level language supports conveniently introducing tracers (Section 4.5.5), aiding in the debug cycle.

While our focus is on safety, we also took steps to ensure the correctness of our tools. Indeed, to aid in debugging vWasm during development, we implemented a semantics fuzzer, included in our open source release, which randomly generates valid Wasm programs that check their own results during execution, in order to help identify potentially flawed semantics. The core idea of this fuzzer is to generate code of the form “if 2 + 3 is not 5, exit with failure”. This fuzzer helped us identify and fix over 15 distinct semantic correctness issues in vWasm (none of which threatened sandbox safety, but which could lead to incorrect computation results). No issues were found by the fuzzer in rWasm. Both vWasm and rWasm have now been fuzzed extensively, and they both pass all correctness and consistency checks for the benchmarks.

Overall, we conclude that the initial development effort for rWasm is significantly better than vWasm.

Development Effort: Maintenance Of course, the development effort for any software used in practice cannot be understood purely from its initial implementation effort but must also consider ongoing maintenance costs. This can be quite subtle for both vWasm and rWasm, and also somewhat speculative. For the former, there is no further maintenance effort needed if one simply wants to use it on x64 processors, since the architecture, while likely to change, will largely only introduce new instructions while keeping the existing instructions the same. However, improving the performance of code generated by vWasm would require quite some effort. Since the stages before the sandboxing pass are unverified, the effort is comparable to any other maintenance for a compiler. Wasm, as a standard, is not yet completely finished and new proposals will continue to improve upon it, adding new instructions and potentially new control-flow constructs; these should only require a small amount of effort to introduce. However, if Wasm introduces a new way to access memory, this could take a larger amount of effort to introduce into vWasm, since it might impact the sandboxing pass. Finally, despite our efforts to keep our proofs as general as possible, adding support for a new architecture (such as ARM) would be a straightforward but non-trivial amount of effort, given the inherent difficulty of writing formal proofs. In contrast, rWasm automatically supports all architectures supported by the Rust compiler; thus, support for new architectures comes to rWasm “for free” from the broader Rust community. Similarly, performance of the overall compiler automatically improves as `rustc`’s performance is improved, without any changes needed to rWasm’s code itself. Additionally, new domain-specific optimizations, new Wasm instructions, control flow constructs, ways to access memory, etc. could be added to rWasm with little effort, as seen by the ease of the initial implementation.

Static Property Extensibility Provable safety is an important property of a verified sandboxing compiler, but one might wish to prove other properties, such as traditional compiler correctness. Here, vWasm has the upper hand, as this is feasible to do in F*, and we have even structured the compiler to make such proofs possible. In contrast, proving correctness for rWasm would be a challenging task, since one would need to formally model the Rust language, show that rWasm preserves Wasm semantics in compiling to Rust, and then implement a semantics-preserving Rust compiler (or prove `rustc` as semantics-preserving). The nature of the provable sandboxing property is what puts it into the sweet spot where we obtain it "for free" when compiling to Rust, and we believe there may be other such properties where one can obtain provable guarantees in a similar fashion. However, all these properties are a strict subset of what might be proven for an implementation like vWasm, which is built in a full-blown verification-oriented language.

Run-Time Extensibility As discussed in Section 4.5.5, rWasm supports conveniently adding runtime tracers, or Inline Reference Monitors (IRMs). It does so by leveraging its emulation of Wasm in a high-level language (Rust) to succinctly inspect and modify the program's state. In contrast, implementing these for vWasm would require a significant re-architecture of the compiler, which follows a traditional compiler pipeline oriented towards progressively lowering Wasm towards machine code. Thus, run-time behavior of a Wasm module can be better observed, analyzed, and controlled when compiled via rWasm.

4.7 Related Work

Virtualization-based Isolation Hypervisors and VMMs can provide strong sandboxing, such as with the Hypervisor-Protected Code Integrity (HVCI) [50] option for drivers on Windows. However, this is heavyweight and usually requires hardware support. Lighter weight than this, most operating systems guarantee strong isolation between processes, and some even provide OS-level virtualization, used by container frameworks such as Docker [29] and LXC [79]. This can still be too expensive due to the overhead of IPC, multiple privilege-level crossings, cache flushes, etc. Instead, SFI provides sandboxing that is intra-process, costing little more than a function call. To use a sandboxed module, a developer simply links against it, easing deployment.

Language-based Isolation Some programming languages have VMs that can provide isolation guarantees (e.g., V8 for JavaScript, JVM for Java, and CLR for .NET). However, these must trust the complex implementation of the language's VM, and they restrict usage to their particular language. In contrast, by using Wasm as a narrow waist (Section 4.2), we support sandboxing for nearly all popular languages.

Validator-based SFI Software Fault Isolation (SFI) [143] is a popular technique for providing language-agnostic, lightweight, and safe software sandboxing. Traditionally, SFI solutions have an untrusted component that introduces the checks, and a trusted validator that confirms that the checks are sufficient before execution. On the Web, NaCl [159] uses this approach, and RockSalt [96] even provides a formally verified validator. They rely, however, on a custom compiler toolchain to make the emitted code easier to validate. In contrast, VeriWasm [54] is a formally verified validator that confirms checks produced by an uncustomized compiler, Lucet [4], which further optimizes code *after* inserting SFI checks. To do this without false positives, VeriWasm uses features of Wasm, as well as implementation choices specific to Lucet. This is quite challenging to do without rejecting legitimate programs or suffering soundness issues (e.g., CVE-2021-32629), due to Rice's theorem [124]. Instead, our approaches guarantee sandbox safety by construction. Additionally, while validator-based approaches like NaCl, RockSalt and VeriWasm are necessarily architecture dependent, rWasm provides architecture-agnostic provable sandboxing. Finally, while most previous SFI solutions use a fixed-size sandbox for performance, rWasm attains high performance without reserving a large, fixed-sized chunk of memory (see Section 4.5.4), making it feasible to use even in embedded environments.

Compilation-based SFI Kroll et al. [63] present a technique for Portable SFI (PSFI) that is architecture-agnostic, with performance comparable to GCC [40] with no optimizations enabled. PSFI works on Cminor, an intermediate language of CompCert [76] (which compiles from C), and works by composing a program transformer with the verified backend of CompCert. Our compilers instead are multi-lingual since they support any language that can be compiled to Wasm (Section 4.2.3). In theory, PSFI could also be extended to offer multi-lingual support by writing translators from other languages to Cminor.

Additionally, rWasm obtains competitive performance with unverified performance-optimized implementations (Section 4.6.1). Due to Rust's collection of supported target architectures, rWasm can also target more architectures with no additional effort.

WebAssembly Multiple compelling use cases have been shown for using Wasm as a sandboxing primitive. RLBox [102] provides a framework for retrofitting isolation of third-party libraries in complex pre-existing software like Firefox; eWASM [112] provides a framework for SFI using Wasm on embedded systems with resource constraints; and Sledge [38] enables low-latency serverless compute on the edge via Wasm. Employing our techniques for provably-safe sandboxing within these framework would provide greater assurance of their safety.

Given Wasm's growing prevalence, it is important to identify its performance bottlenecks compared to running purely native code. Jangda et al. [53] perform a large-scale evaluation of browser Wasm runtimes, which helps identify causes for these bottlenecks, some inherent to Wasm, and others due to implementation deficiencies. These highlight opportunities that vWasm could take to improve performance.

Recent work by Lehmann et al. [72] shows that classic vulnerabilities such as simple stack buffer overflows, unexploitable in native binaries due to common mitigations, become exploitable again inside Wasm modules. This however does not impact sandboxing runtimes for Wasm such as ours, as any exploit will only corrupt the program state within the sandbox. The environment is left unaffected, modulo calls via the trusted interface offered explicitly by the environment.

Proposed extensions to Wasm, such as MS-Wasm [28] (for memory safety), can help capture critical high-level information about the program being compiled. Other extensions like CT-Wasm [123] (for constant-time cryptography) help capture high-level invariants that one wishes to maintain. These extensions are orthogonal to the goals of sandboxing, and can help provide even stronger guarantees of safety and correctness to the native code execution.

4.8 Concluding Remarks

In this chapter, we have explored two concrete points in the design space for implementing a sandboxing execution environment, with a focus on WebAssembly. We proposed designs for these two points, implemented them as open-source tools, vWasm and rWasm, and evaluated them on a collection of both quantitative and qualitative metrics. We show that run-time performance and provable safety are not in conflict, and indeed rWasm is the first Wasm runtime that is both provably-sandboxed and fast.

In the future, it would be interesting to explore proving other, potentially stronger, properties within the framework of our two compilers. Specific to WebAssembly, there are multiple pending proposals to enhance the standard (e.g., adding multi-threading), and each could potentially have interesting consequences for the design of our compilers. In the next chapter, we will see one such extension, focused on intra-module safety—MSWasm. Additionally, while speculative attacks are out of scope, it may be interesting to employ emerging defenses in our compilers.

In summary, in this chapter we have shown that, for safely sandboxing arbitrary code, unapologetic security is achievable—principled and formal approaches can provide provable safety, without apologizing for either development velocity, runtime performance, or portability.

Chapter 5

Pre-Emptively Defending Code

Jason: *I'm telling you, Molotov cocktails work. Anytime I had a problem, and I threw a Molotov cocktail, boom! Right away, I had a different problem.*

The Good Place
S02E11 "Rhonda, Diana, Jake, and Trent"

In the previous chapter, we saw how we might defend against issues in third party code through provably software-based enforcement. In this chapter, we will see how we can expand on this to provide agility in enforcement, expanding both the scope of enforcement, as well as providing a platform to easily make contextual decisions in changing environments, such as the release of a new hardware-based enforcement technique, or the discovery of a bug in critical first party code.

Contributions. This chapter consists of our work on MSWasm presented at POPL'23 [91]. My contributions on MSWasm focused on the design and implementation of the agile ahead-of-time software-enforcement backends, in addition to aiding my co-authors on several other important parts of the overall implementation. The core design of MSWasm (the language) itself originally appears earlier in a position paper by Disselkoen et al. [28], which we expand upon in our paper, with both rigorous formalism and practical implementation. Given my work's focus on the implementation, this chapter focuses primarily on the implementation; however, other relevant portions from the paper are included here where necessary for context.

5.1 Introduction

Pragmatically, in addition to defending against issues in third party code, one might also wish to defend against issues in one's own (i.e., first party) code. Naïve approaches for this, such as treating one's own code adversarially, might need to apologize for 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 first or third 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 Chapter 4, Wasm is a great narrow-waist for providing software sandboxing. This allows for isolating the impact of a vulnerability to one Wasm module, keeping it 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. [72], 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 [100, 101, 104]. 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 (e.g., due to violation of memory layout assumptions). It gives up on *performance*: efficient memory-safety enforcement techniques rely on operating system abstractions (e.g., virtual memory [20]), 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 [7, 77]). Finally, it also makes it harder to prove that memory safety is preserved end-to-end.

With *Memory-Safe WebAssembly*, Disselkoen et al. [28] 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 [150], 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 proposal to realize the vision of MSWasm.

We describe precise and formal semantics of MSWasm in our paper [91], giving precise meaning to the previous informal design [28], 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). This chapter focuses primarily on the AOT compilers from MSWasm to machine code.

Extending rWasm (Chapter 4) 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 [115]. 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 [101]. MSWasm, however, makes it easy to change the underlying enforcement mechanism (e.g., to boost performance), without changing the MSWasm module. 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% [77], while Arm's CHERI [42] or Intel's CCC [75] 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, almost for 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, which 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, this approach unlocks easy upgrades to new enforcement mechanisms, both in software and hardware, promoting long term improvements at reduced development cost.

All our implementations, benchmarks, and data sets are available as open source.[1]

5.2 Background and Motivation

In this section, we briefly recall to the Wasm threat model. We then discuss the implications of memory unsafety in the Wasm sandbox, and give a brief introduction to MSWasm, both in terms of threat model, as well as design.

5.2.1 WebAssembly Threat Model

As discussed in Section 4.2.2, Wasm programs consist of modules, each of which consists of collections of code, data, and associated connections to the environment (or other modules). These modules form the boundaries of protection offered by WebAssembly. In particular, the runtime is supposed to ensure that no Wasm module can access any memory or APIs outside of what it is explicitly given access to (see Chapter 4 for provably achieving this sandboxing).

In short, the threat model of WebAssembly is that an attacker has the ability to invoke arbitrary functions with arbitrary inputs to one or more (possibly maliciously crafted) Wasm modules, and wishes to obtain code execution in (or sensitive data from, or corrupt, or . . .) either the host environment or other modules.

While WebAssembly's threat model *does* protect against a whole class of important attacks, this might not by itself be sufficient for some practical applications. In particular, an attacker might not care about cross-module (or module-to-host) escalation, if their intended target (say, sensitive data) is within the same module.

Indeed, memory-unsafe C programs, when compiled to Wasm, largely remain unsafe—they can run uninterrupted as long as their reads and writes stay within the bounds of the entire linear memory of the Wasm module. Worse, Wasm also lacks most mitigations we rely on today to deal with memory unsafety (e.g., memory protection bits and ASLR), so a

[1] `https://mswasm.programming.systems/`

program compiled to run within Wasm's sandbox may be more vulnerable than if it were running on bare metal [72].

Although an attacker could not use such unsafety to escape Wasm's sandbox, they can use it to corrupt or leak sensitive data *within* the sandboxed module itself. This is not to be taken as an indictment of WebAssembly, since it was not intended to guard against unsafety within the sandbox. Indeed, the entire point of sandboxing is that arbitrary corruptions are allowed within the sandbox, *without* causing damage outside of it. However, partitioning existing programs often is not simple, and thus sandboxing might not be helpful in such scenarios.

5.2.2 MSWasm Threat Model

MSWasm's threat model accounts for intra-module safety, in addition to the sandboxing safety of Wasm. Specifically, an attacker wishes to corrupt or obtain control over an MSWasm module compiled from a memory-unsafe C program, and has the ability to invoke arbitrary functions with arbitrary inputs to it. We consider vulnerabilities that can be triggered by *spatial* memory errors (e.g., buffer overflows), *temporal* memory errors (e.g., use-after-free and double-free vulnerabilities), and *pointer integrity* violations (e.g., corrupting function pointers to bend control flow). Similar to that in Wasm's threat model, the attacker can provide arbitrary modules of their own, as well as inputs to any of the modules.

5.3 MSWasm: Memory-Safe WebAssembly

MSWasm's addresses this threat model, and challenges introduced by it, by extending Wasm with abstractions for enforcing intra-module memory safety [28, 91]. Specifically, MSWasm introduces a new memory region called *segment memory*, which consists of individual *segments*—linearly addressable, bounded regions of memory representing dynamic memory allocations.

Unlike Wasm's linear memory, the segment memory cannot be accessed at arbitrary `i32` offsets through standard `load` and `store` instructions. Instead MSWasm provides new types, values, and instructions to regulate access to segments and enforce per-allocation memory safety. Segments can only be accessed through *handles*, unforgeable memory capabilities that model pointers bounded to a particular allocation of the segment memory.

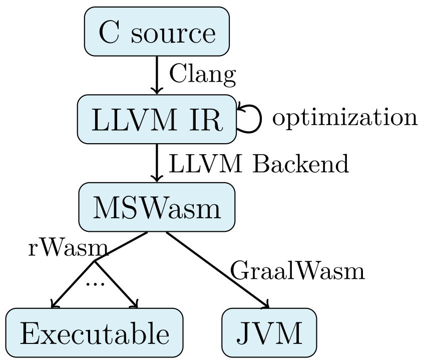


Figure 5.1: End-to-end compilation pipeline. *We first compile C to MSWasm (via LLVM), and then compile MSWasm to machine code using either our modified rWasm AOT compiler (which supports different notions of safety) or our modified GraalWasm JIT compiler.*

Handles *conceptually* consist of an offset into a particular segment, along with additional information to ensure the handle's validity. Spatial memory safety can be checked be ensuring that the handle is within the bounds of the segment, and temporal memory safety by maintaining unique identifiers for segments. MSWasm guarantees integrity of handles by using a "corrupted" flag; intuitively, attempts to forge handles (e.g., by casting an integer, or altering the bitstring representation of an existing handle in memory) result in a corrupted handle. MSWasm traps only when an invalid (out-of-bounds, freed, or corrupted) handle is *used*, not when it is created. This is primarily for compatibility, since many common C idioms create benign out-of-bound pointers [89, 90, 125].

MSWasm provides new instructions to create and manipulate handles, and to access segments safely through them; this includes versions of `segload` and `segstore` for each primitive type (analogous to `load` and `store`), in addition to `segalloc` and `segfree` to allocate and free segments dynamically at run-time. Lastly, `handle.add` (similarly, `sub`, `eq`, ...) supports pointer arithmetic, to manipulate (similarly, compare) handles within the same segment.

More details and formalism can be found in our paper [91].

5.4 Implementing MSWasm

Our prototype MSWasm compilation framework (Figure 5.1) consists of a compiler from C to MSWasm, and two compilers *of* MSWasm. In this section, we describe specifically our ahead-of-time (AOT) compiler from MSWasm to executable machine code, which demonstrates

MSWasm's flexibility in employing different enforcement mechanisms, including both software-based enforcement and hardware-accelerated enforcement. Details of our other compilers (C-to-MSWasm, and MSWasm-to-JVM JIT compiler) can be found in our paper [91].

Our prototype implementation of MSWasm extends the bytecode of Wasm with instructions to manipulate the segment memory as well as handles. In doing so, it takes a few shortcuts in the name of expediency—most notably, it replaces the existing Wasm opcodes for `load` and `store` (of each type; e.g., `f32.load`) with a corresponding `segload` and `segstore` respectively. A production MSWasm implementation would support both segment-based and linear-memory-based operations simultaneously, by using two-byte opcode sequences for `segload` and `segstore`.

5.4.1 Ahead of Time Compilation of MSWasm

To compile MSWasm bytecode to machine code, we build on the rWasm compiler [13]. As described in Section 4.5, rWasm is a provably-safe sandboxing compiler from Wasm to Rust, and thus to high-performance machine code.[2] We extended rWasm to support MSWasm as follows. We modified rWasm's frontend to parse MSWasm instructions and propagate them through to later phases. We updated rWasm's stack analysis to account for MSWasm's new types and instructions (e.g., `segload` and `segstore`, which take a `handle` as argument). Finally, we updated rWasm's backend—the code generator, specifically—to implement MSWasm's instructions and segment memory.

One of the benefits of MSWasm is that it gives Wasm compilers and runtimes flexibility in how to best enforce memory safety. This is especially important today: memory-safety hardware support is only starting to see deployment and applications have different security-performance requirements—we cannot realistically expect everyone to pay the cost of software-based memory safety. When hardware becomes available, MSWasm programs can take advantage of hardware acceleration almost trivially: in our AOT compiler, for example, we only need to tweak the codegen stage. We demonstrate this flexibility by prototyping two different software techniques, and one hardware-accelerated technique that have different safety and performance characteristics.

[2]In modifying rWasm, we were careful to ensure that we preserve its previously-established sandboxing/isolation guarantees. These guarantees, together with the internal memory-safety guarantees from MSWasm, increases the level of protection for native code generated by rWasm.

Segments as Vectors. Our default technique for memory-safety enforcement closely matches the conceptual interpretation of handles in Section 5.3, and enforces spatial safety, temporal safety, and handle integrity (rWasm$_{STH}$ in Section 5.5). We implement the segment memory as a vector (`Vec`) of segments. Each segment is a pair composed of a `Vec` of bytes (giving us spatial safety) and a `Vec` of tags, which is used to enforce handle integrity—any possibly-unsafe (i.e., not obviously-safe) operation that manipulates memory that stores a handle changes the tag of that location to mark it as corrupted. Handles themselves are implemented using an `enum` (i.e., a tagged union). To enforce temporal safety we clear free segments from memory and use sentinel value to prevent the reuse of segment indexes. A slight variation of this technique (rWasm$_{ST}$ in Section 5.5) gives up on handle integrity (we remove the `Vec` of tags and related checks) for performance.

Segments with Baggy Bounds. Our second technique is inspired by Baggy Bounds checking [1], which is a technique that performs fast checks at each handle-modifying operation and elides checks at loads and stores, enabled by expanding buffers to the next power of two at the point of allocation. This technique gives up on handle integrity and temporal safety, since accesses are not checked, but it is considerably faster (rWasm$_{S}$ in Section 5.5). To implement this technique, our compiler uses a single growable `Vec` of bytes, within which a binary buddy allocator allocates implicit segment boundaries. We implement the handles as 64-bit values storing an offset in memory and the log of the segment size (rounded up to nearest power of two at allocation). We emit bounds checks for each operation that might modify handles, ensuring that handles remain within the (baggy) bounds of their corresponding segment. Specifically, when handles stray a short distance outside their segment, we mark them as such (and they can safely return back), but we trap when they (try to) stray too far.

Hardware acceleration using CHERI. Our third technique implements segment memory using CHERI capabilities [150], by adding a new rWasm backend that emits a subset of CHERI-compatible C code, which implements handles as capabilities. More details can be found in our paper [91].

Implementation Effort. Our modifications to rWasm, for both software-only memory enforcement techniques, comprise roughly 1900 lines of additional code. The implementation of these two techniques comprise approximately 500 lines of code each in rWasm's codegen, and share the rest of rWasm's codebase. The hardware-accelerated technique also uses our modified rWasm frontend, but could not share much of rWasm's codegen with the other backends (which target Rust), instead requiring code to support a new target language—CHERI-C (even for the regular non-MSWasm-specific components of Wasm); our modifications for this backend comprise approximately 3000 lines of code. The relative ease of these modifications, both for software- and hardware-based techniques illustrates how MSWasm provides a fertile ground for experimenting with new techniques for providing performant memory safety.[3]

5.5 Evaluation

In this section we describe our performance evaluation of the MSWasm compiler. We use the PolyBench-C benchmarking suite [115] since PolyBench-C has become the de-facto suite used by almost all Wasm compilers (although some limitations of PolyBench-C are noted by Jangda et al. [53]). We compare the performance of MSWasm to the performance of the same benchmarks compiled to normal Wasm, on each of our implementations.

We compile all benchmarks from C to Wasm using Clang, and from C to MSWasm using our modified CHERI Clang compiler; in both cases we set the optimization level to `-O3`. We run all our software-based enforcement benchmarks on a single core on a Linux-based system with an Intel Xeon 8160, and our hardware-accelerated enforcement benchmarks on the ARM Morello platform [8].

Figure 5.2 summarizes our measurements (see Figure 5.3 for a detailed breakdown), normalized against the execution time of native (non-Wasm) execution. In this figure, $\text{rWasm}_{\text{Wasm}}$ and $\text{Graal}_{\text{Wasm}}$ refer to execution of normal Wasm. We distinguish the different MSWasm compilers according to their enforcement techniques: rWasm_{STH} enforces spatial safety, temporal safety, and handle integrity; rWasm_{ST} and Graal_{ST} only enforce spatial and temporal safety; and, rWasm_{S} only enforces spatial safety (in the style of baggy bounds).

As expected, and in line with prior work [100, 101], each safety enforcement techniques comes with a performance cost—handle integrity being the most expensive. For the AOT compiler, we observe that enforcing spatial safety alone rWasm_{S} has a geomean overhead of 21.4% over $\text{rWasm}_{\text{Wasm}}$; additionally enforcing temporal safety (rWasm_{ST}) results in an overhead of 52.2% over $\text{rWasm}_{\text{Wasm}}$; and, finally, further enforcing handle

[3]As a brief note, our JIT compiler required adding roughly 1200 lines of code to GraalWasm, and our compiler *to* MSWasm, built on top of CHERI LLVM required approximately 1600 lines of code (in particular to its Wasm backend, and the WASI libc).

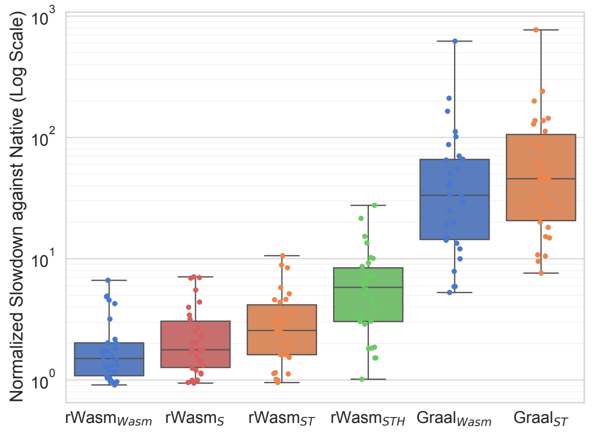


Figure 5.2: Runtime performance of MSWasm. *Performance of our implementations of MSWasm compared to normal Wasm, normalized against native (non-Wasm) execution on benchmarks from PolyBench-C.*

integrity (rWasm$_{STH}$) increases the end-to-end overhead to 197.5%. For the JIT compiler, enforcing spatial and temporal safety results in an overhead comparable to that of the AOT compiler: Graal$_{ST}$ imposes a 42.3% geomean overhead. The JIT approach is much slower than the AOT approach though—the overheads of rWasm$_{\text{Wasm}}$ and Graal$_{\text{Wasm}}$ over native (non-Wasm) execution are 71.8% and 3230.0% respectively. We also note that with increasing iterations of the GraalVM JIT, Graal$_{\text{Wasm}}$'s performance improves more rapidly than Graal$_{ST}$'s, which suggests that our implementation still has potential to make better use of GraalVM's optimizer.

Our hardware-accelerated approach, which enforces spatial safety and handle integrity (but not temporal safety), runs on an entirely distinct architecture and platform. Thus, a direct comparison against the same native code baseline used for the other techniques would not be particularly instructive. Instead, we evaluate our MSWasm-CHERI backend against native CHERI code that uses 128-bit registers to store capabilities (known as pure capability mode, as opposed to hybrid mode, which by-default uses 64-bit registers to store raw pointers) on the Morello platform, and find an overhead of 51.7%. Analysis of the benchmark programs with the highest overhead shows that the majority of the slowdown seems to be orthogonal to memory safety, caused instead by the CHERI-clang compiler missing opportunities for vectorization optimizations. We believe this can be remedied with additional engineering.

Since normal Wasm and MSWasm have different bytecode formats, our evaluation of MSWasm performance necessarily includes slowdowns caused by inefficiencies in our compi-

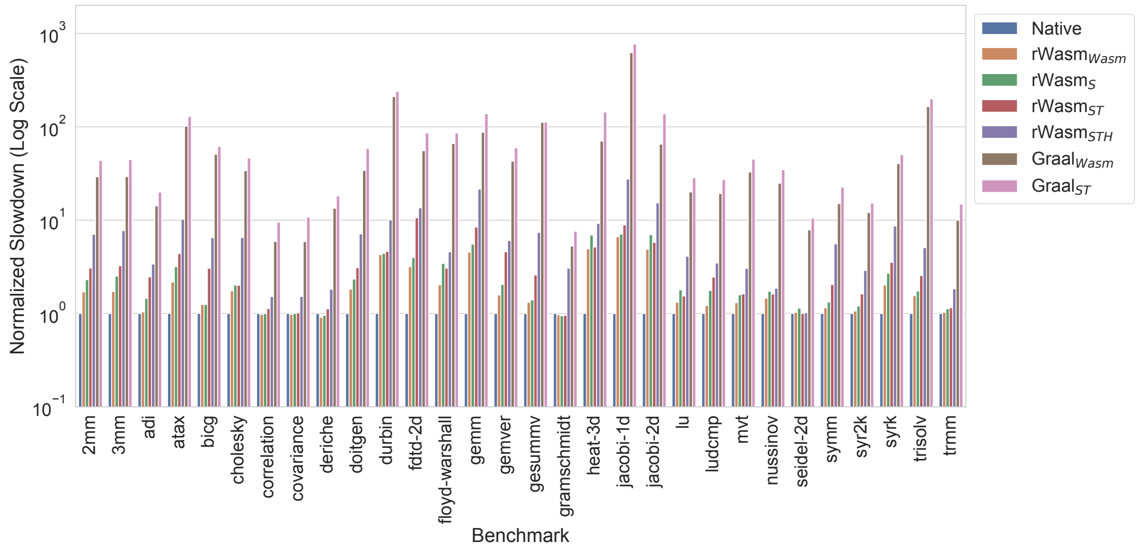


Figure 5.3: Detailed runtime performance of MSWasm. *A detailed per-program breakdown of the performance of our implementations of MSWasm compared to normal Wasm, normalized against native (non-Wasm) execution on benchmarks from PolyBench-C. Rather than relying on less accurate external measurements using* `time(1)`*, we use PolyBench-C's own internal execution time reporting (i.e.,* `-DPOLYBENCH_TIME`*).*

lation from C to MSWasm. But because MSWasm decouples memory safety enforcement from the generation of MSWasm bytecode, both parts of this pipeline (C-to-MSWasm compilation, and MSWasm to machine code) can be independently optimized, with MSWasm performance benefiting from improvements on both sides.

5.6 Related Work

Efficient memory-safety implementations. Unlike compiler-based instrumentations, compiling to MSWasm does not commit to a particular concrete strategy for enforcing memory safety: Different implementations of MSWasm can use different enforcement approaches. In particular, MSWasm enables backends compilers and runtimes to leverage efficient software- and hardware-based mechanisms, independently proposed to enforce pointer integrity [77], spatial [1, 7, 62], and temporal [70, 111] safety, to create new practical memory-safety enforcement schemes. Because MSWasm is platform-agnostic, we expect that implementations will be able to opportunistically take advantage of hardware memory protection mechanisms on individual platforms [7, 27, 65, 109] (current and proposed) to efficiently implement handles.

Software isolation via Wasm. Wasm abstractions provide an efficient software-isolation mechanism, which has been applied in many different domains. For example, using Wasm, the RLBox framework [103] retrofits isolation into the Firefox browser; Sledge [38] enables lightweight serverless-first computing on the Edge; and eWASM [112] demonstrates practical software fault isolation for resource-constrained embedded platforms. These use cases already rely on both the performance and the sandboxing safety of Wasm, and stand to benefit from MSWasm's focus on memory safety.

vWasm and rWasm [13] (Chapter 4) use formal methods and non-traditional techniques respectively to provide provable isolation between the Wasm module, running as a native library, and the host process executing it. Their focus is on provable sandboxing, and module-internal memory safety is explicitly left out of scope. As shown by Lehmann et al. [72], Wasm lacks many common defenses (e.g., stack canaries, guard pages, ASLR) against classic memory safety vulnerabilities, such as buffer overflows.

Jangda et al. [53] perform a large-scale performance evaluation of browser Wasm runtimes, comparing to native code. Our evaluation of MSWasm's performance (Section 5.5) shows that adding memory-safety protections does not fundamentally change Wasm's performance story. In particular, adding spatial and temporal safety imposes less overhead on Wasm than the overhead Wasm already incurs vs native code.

5.7 Concluding Remarks

In this work, we demonstrate that by adding a small language-level memory-safety extension to Wasm, namely the MSWasm extension, we obtain a convenient virtual machine that supports agile enforcement of memory safety. This agility allows for contextual switching between different enforcement mechanisms with varying tradeoffs between speed and security, *without* requiring changes to the MSWasm module itself.

Our PolyBench-C-based evaluation shows that MSWasm introduces an overhead ranging from 22% (enforcing spatial safety alone) to 198% (enforcing full memory safety). Our software-based implementations primarily serve to highlight that the enforcement intra-module memory safety for Wasm is possible and, moreover, that MSWasm makes it easy to change the underlying enforcement mechanism without modifying application code. This means MSWasm engines will be able to take advantage of clever memory safety enforcement techniques today and hardware extensions (such as CHERI) in the near future, progressively (and transparently) improving the safety of the applications they run.

In the future, it would be interesting to explore alternate enforcement strategies, both in software and hardware. Particularly interesting might be strategies that provide probabilistic enforcement, allowing for more fine-grained tradeoffs between speed and security, rather than the more discrete jumps of the currently implemented enforcement mechanisms. Also interesting would be to use probabilistic enforcement as an early-detection mechanism for potential vulnerabilities at almost zero amortized cost (in a similar vein to GWP-ASan [130]). Additionally, it may be interesting to explore optimizations enabled by the particular structure of MSWasm, for example, the usage of segments rather than linear memory could (in theory) allow for transparent defragmentation of memory allocations.

In summary, in this chapter, 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.

Chapter 6

Handling Untrusted Data

Words are abstraction, break off from the green; words are patterns in the way fences and trenches are. Words hurt.

This is How You Lose the Time War
Amal El-Mohtar *and* Max Gladstone

So far, we've been working on ways to deal with issues that arise with the code itself, whether it be from writing ultra-high-performance code, or running arbitrary code. In this chapter, we switch gears to deal with untrusted data. In particular, we focus on safely dealing with the serialized representations of data as they would exist either in files or on the wire.

Contributions. The work presented in this chapter spans two papers that are yet to appear. The intrinsic data format framework (VMarshal) will make an appearance in our paper on Verus as a Practical Foundation for Systems Verification [68], as part of a larger macro-benchmark evaluation of Verus for practical systems verification. The design and implementation of VMarshal, along with its description and evaluation are my own, albeit with helpful comments from co-authors on the paper. The extrinsic data format framework (Vest) will be a part of our paper on the Owl compiler [134], which compiles high-level Owl protocols [39] to Verus code and proofs. The core design of Vest is my work, implemented in practice by Yi Cai, with additional guidance from co-authors on the paper.

6.1 Introduction

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 the 15 "most challenging weaknesses that exist today" [94].

Especially in high-assurance software, such as with formally verified security protocol implementations [10, 24], one might wish to use formally verified parsing and serialization. Prior works [23, 55, 119, 137, 144] 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 in more detail in Section 6.3.

Broadly, 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 inform the choice of the (otherwise flexible) 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.

With this split, we design and implement two separate parser/serializer frameworks (Sections 6.4 and 6.5), reducing burden for users. Both our frameworks prove various properties about the parsers and serializers, such as high-level data surviving a round-trip through them. The goal for these frameworks is not to prove that the implementation matches a particular grammar specification (an orthogonal concern), but to prove a collection of properties similar to ones in prior work, such as non-malleability, which aid in proving important higher-level properties in verification projects. Additionally, for broad applicability, so that even unverified contexts can take advantage of these parsers, we use Verus [68, 69], 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 a foreign function interface (FFI) or awkward API.

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.

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.

6.2 Background

Here, we discuss some background about parsing techniques, as well as Verus, the systems verification framework that powers our pair of parser-serializer frameworks.

6.2.1 Parsing Techniques

In contrast to serialization, which is *relatively* straightforward (one "merely" has to pick the right bytes and place them into the right spot), correct and performant parsing is significantly more involved. Thus, it is unsurprising that there has been a lot of research over the years on different techniques to perform parsing.

At a high-level, a parser takes a string of some alphabet (typically bytes, ASCII/Unicode characters, or tokens—if lexing has been performed), and produces a higher-level representation (typically a syntax tree, or structured data), if the string is a member of a specified grammar that describes the recognized language. The grammar itself may be implicit, or explicitly stated using formal grammar rules.

Parsers themselves can thus be classified into top-down or bottom-up parsers. The former focuses on attempting to find an expansion of grammar rules from the start symbol that matches the string. In contrast, the latter starts from the string and attempts to rewrite lower-level symbols to higher-level ones, until it is able to reach the start symbol.

A popular technique, hand-written recursive-descent parsing, is top-down and consists of a series of mutually-recursive functions, often written in an imperative fashion, representing the non-terminals of the grammar (or auxiliary helper functions). Recursive descent parsers can provide excellent error reporting, and allow for great flexibility for tweaking the parser—after all, they are just a collection of normal functions in the programming language, merely implemented in a consistent style. However, this is also their downfall—with an implicit grammar, and nothing to enforce consistency, they can be notoriously difficult to write correctly, and especially in memory-unsafe languages like C, seem to be a common source of vulnerabilities and exploits.

Parser generators, as popularized by Yacc, Bison, ANTLR, and more, provide a stark contrast. With such tools, the programmer writes an explicit grammar, and uses the tool to generate a parser, typically bottom-up. The explicitness of the grammar provides a succinct specification of the grammar, without the distraction present in the parser code (unlike in recursive descent parsers). However, such parsers lack the flexibility of easily adding custom code, and require an extra build step, and thus appear to be less frequently used compared to hand-written recursive-descent parsers. Nonetheless, due to having an explicit grammar, syntactic ambiguities are detectable at compile time. An alternate approach, used by modern parser generators such as pest [25], is to use a style of grammar the eliminates ambiguities entirely—Parsing Expression Grammars (PEGs). These replace the regular alternation (aka choice) in grammars with ordered-choice, forcing a resolution on what could otherwise be ambiguous. This order-dependence makes PEGs essentially imperative and can make them non-trivial to use correctly for some grammars; nonetheless, they are well suited for grammars where multiple interpretation alternatives can be disambiguated locally (such as tag-length-value schemes used in security protocols). As an example of practical usage, the commonly-used reference implementation of Python, CPython, switched to an entirely PEG-based parser starting from Python 3.10 [140].

Functional programming often uses yet another parsing technique—parser combinators, as popularized by Parsec [73]. These are a variant of recursive descent parsers, which factor out common repetitive patterns into higher order functions, which can take and return parsers as input/output. In practice, parser combinators provide both flexibility and readability, yet suffer the same shortcoming as regular recursive descent parsers—implicitness of grammar. Ambiguities in the grammar cannot be caught easily at compile time when the grammar is implicit.

```
spec fn fib(n: nat) -> nat
    decreases n,
{
    if n <= 1 {
        n
    } else {
        fib((n - 1) as nat) + fib((n - 2) as nat)
    }
}

proof fn fib_monotonic(a: nat, b: nat)
    requires a <= b,
    ensures fib(a) <= fib(b),
    decreases b, b - a,
{
    if (a < 2 && b < 2) || a == b {
    } else if a == b - 1 {
        fib_monotonic((a - 1) as nat, (b - 1) as nat);
    } else {
        fib_monotonic(a, (b - 1) as nat);
        fib_monotonic((b - 1) as nat, b);
    }
}
```

(a) Specifying and writing proofs about the purely-Mathematical function `fib`, representing the n^{th} Fibonacci number

```
exec fn fibonacci(n: u64) -> (r: u64)
    requires fib(n as nat) <= u64::MAX,
    ensures r == fib(n as nat),
{
    if n <= 1 {
        return n;
    }
    let (mut a, mut b) = (0, 1);
    for i in 1..n
        invariant
            a == fib((i as nat - 1) as nat),
            b == fib(i as nat),
            fib(n as nat) <= u64::MAX,
    {
        proof {
            fib_monotonic(i as nat + 1, n as nat);
        }
        let x = a + b;
        a = b;
        b = x;
    }
    b
}
```

(b) An executable implementation of the n^{th} Fibonacci number, proven equivalent to `fib`

Figure 6.1: Example Code and Proofs written in Verus

6.2.2 Verus: A Systems Verification Framework

Formal verification with SMT-based automation (Section 2.2) is incredibly useful, and many large scale projects have been built with tools like F* [136], Dafny [74], and more. Unfortunately, SMT-based automation has not scaled well enough for low-level systems code in the past. In particular, software that requires a lot of low-level memory reasoning tends to overwhelm classical tools and frameworks such as Dafny and Low* [117] (a subset of F* focused on verifying C), due to their encoding of memory reasoning into SMT.

Interactive theorem provers, such as Coq [138] and Lean [22], allow a lot more control over details, and thus could be used to represent and work with low-level code. However, this often comes at the cost of automation, often requiring manually working through straightforward-but-tedious proofs,[1] or scaling up automation by writing complex tactics that can then fall prey to the same sorts of woes that befall SMT-based automation.

Along a similar vein, in regards to memory safety, garbage-collected languages provide

[1]Sometimes referred to as the tactic *apply_grad_student*

an incredible productivity boost, since they do not require the programmer to manually reason about memory. This often however comes at the cost of unpredictable and lower performance. In contrast, manually managing memory (such as in C) provides the opposite tradeoff, and often was the only practical and pragmatic choice for various low-level software situations.

In recent years however, with Rust [83], we have the best of both worlds—the convenience of automated memory management, as well as the low-level control and performance of manual memory management. It achieves this by relying on linear types, lifetimes, and borrowing.

Inspired directly by this, a new verification tool was born—Verus [68, 69]. By layering verification conditions on top of (safe) Rust, Verus leverages the power of the memory reasoning provided by Rust, rather than encoding memory reasoning to SMT. By using Z3 [21], Verus leverages the power of automation provided by SMT-solvers for everything else.

Figure 6.1 shows a toy example of code and proofs written in Verus. Since it is built on top of Rust, its syntax is very similar to Rust, with some changes to support verification. Functions come in three flavors, `spec`, `proof`, and `exec`; these correspond to mathematical specifications (pure functions), lemmas (which can require pre-conditions to be invoked, and provide their `ensures`-clauses when invoked), and executable code. Only `exec` functions exist in the final code at run-time, and the other two exist to aid them. Similar to `proof` functions, `exec` functions also support pre- and post-conditions, written as `requires` and `ensures` clauses. They additionally can have non-executable code such as proofs (via executing `proof` lemmas), invariants, assertions, etc. inline, to support the solver in showing that the specified properties about the code actually hold. In our toy example, we show that the executable function `fibonacci` (which operates on machine integers, and uses the faster linear-time algorithm to compute Fibonacci numbers) is equivalent to the mathematical definition `fib`, which operates on mathematical (arbitrarily-sized) natural numbers. The proof `fib_monotonic` is a lemma that proves that Fibonacci numbers are monotonic, and this proof is used by the executable `fibonacci` to prove the absence of integer overflows in its implementation.

6.3 Classifying Data Formats

Broadly speaking, software systems use serialized data in two fundamentally distinct ways, driven primarily by the intended scope of data exchange, and interoperability requirements. When data is used primarily only internal to a specific program or system, a format tailored to such a system suits better; other uses of serialized data must interoperate with the outside world, and thus usage of the data must conform to external constraints and specifications. We classify the former as *intrinsic*, and the latter as *extrinsic* data formats. We discuss each in further detail below.

6.3.1 Intrinsic Data Formats

When a software system communicates with its own components over a network, or saves its state for future executions, the data format it employs is typically intrinsic. In such cases, the program can afford to optimize its data representation for efficiency or convenience without needing to adhere to external standards, or accommodate interoperability requirements. In particular, by having a format intimately tied to the internal data structures of the program, the programmer need not concern themselves with the specific on-wire data representations.

Intrinsic data formats are commonly used in a variety of scenarios, but primarily focus on shorter-term use cases. For example, when working with stable-but-expensive computations over non-trivial data structures, one might wish to cache intermediate results. In such a scenario, it does not matter *how* the data is cached, but *that* it is cached. Since the cache is likely invalidated when the data structure changes, it is not important (and perhaps even counter-productive) to have a particularly stable format, as long as the format is stable enough for the use case.

Looking at practical libraries supporting intrinsic data formats, Python's `marshal` library supports the marshalling and demarshalling of Python values to/from a binary format with the documentation stating "Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does)." [35].

With Python's `pickle` library, we also see that intrinsic formats can (without much additional effort) support longer term use cases too, by including a versioning field, and being explicitly designed to provide backwards-compatibility as a guarantee [36]. This backwards guarantee comes from the `pickle` library internally treating the data format as extrinsic, yet providing an interface that allows users to treat the format as intrinsic. Moreover, in practice, Python recommends users pick `pickle` over `marshal` due to this backwards-compatibility guarantee; as far as the user is concerned, the core use-case for both is scenarios where the programmer does not particularly care about *how* the data is serialized, but *that* it is serialized and deserialized.

In short, a key advantage to (and a defining aspect of) intrinsic data formats is their simplicity of usage for the programmer. The programmer does not need to keep a particular format in sync with the high-level data structures in their program, but instead can focus almost entirely on working on high-level data. This allows for rapid iteration.

In Section 6.4, we discuss our approach towards a formally verified intrinsic data format framework, which provides safe parsers and serializers without apologizing for usability, flexibility, or adaptability to changes.

6.3.2 Extrinsic Data Formats

Extrinsic data formats, in contrast to intrinsic data formats, are characterized by their necessity for communication and interoperability with external entities and systems. Stability and compatibility is essential in heterogeneous environments, and thus the format cannot be arbitrary (due to tight coupling to particular software), and must instead align to agreed upon protocols and formats.

One can list countless examples where interoperability is crucial, simply by looking at different network protocols, or files saved by software. Network protocols like TCP, TLS, etc. necessarily require interoperability across different machines, and are thus standardized. Similarly, `.zip` or `.pdf` files are commonly transferred across machines, or even opened with differing versions of software as time passes, and thus necessitate standardized formats.

Additionally, as discussed earlier, even an intrinsic data format internally might treat the data format as extrinsic, to help achieve longer-term storage or data transmission requirements. Thus, unsurprisingly, when discussing data formats, it is almost always extrinsic data formats that people appear to have in mind. Almost by definition, the intrinsic data format is meant to hide away the fact that the high-level data becomes a low-level serialized form, and back again.

Thus, a lot of focus on the parsing and serializing of data formats (including formally verified parsers and serializers) has focused on extrinsic data formats. In Section 6.6, we discuss related work in the area.

In Section 6.5, we discuss our approach towards a formally verified extrinsic data format framework, which provides safe parsers and serializers without apologizing for usability, expressiveness, or performance.

6.4 VMarshal: A Framework for Intrinsic 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. We use marshalling/demarshalling synonymously with serializing/parsing respectively in this section. Below, we describe the high-level correctness properties provided by VMarshal, followed by the design and implementation that provides them.

6.4.1 Verified Properties

The most fundamental property that a marshalling library must satisfy is that after making a round-trip through serializing and parsing, the high-level data should be effectively unaltered. That is, for a given pair of parser p and serializer s for a high-level type T, we require that $\forall(x : T).\, p(s(x)) \simeq x$.

Notice that we cannot claim that this round-trip is the identity function, but only that it should produce equivalent high-level data. This allows us to ignore possibly-irrelevant low-level details, while maintaining important high-level information. For example, a heap-allocated object need not be at the same position, or a reference-counted object need not have the same reference count; however, any important high-level information must remain identical. Thus, the equivalence for this depends upon the specific high-level type in question.

Perhaps a less fundamental property is the other direction of the round-trip—that the serialization of a parsed value does not change the serialized data (byte representation)—i.e., $\forall(b : \text{u8}^*).\, s(p(x)) = b$. While not crucial in general for serializing and parsing (e.g., whitespace-insensitive formats), this is a prudent property for intrinsic data format libraries to maintain.

From the round-trip properties, it is straightforward to see that it implies that no two differing high-level objects can serialize to the same low-level data. However, a slightly stronger property (called the *strong-prefix* property) is useful to maintain—that neither of the low-level data representations is a prefix of the other; i.e., $\forall(x_1, x_2 : T).\, x_1 \not\simeq x_2 \implies s(x_1) \neq s(x_2)[..|s(x_1)|]$.

In VMarshal, we prove all the aforementioned properties for any parser/serializer pair produced by our framework.

6.4.2 VMarshal: Design and Implementation

VMarshal is a macro- and trait-based library in Verus that automatically generates parsers and serializers for arbitrary Rust `struct`s and `enum`s. At a high-level, this consists of a Verus trait (called `Marshallable`) that specifies the requirements that must be satisfied by any parser/serializer pair, implemented as an instantiation of the trait. With pre-defined macros, 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`.

At its core, the `Marshallable` trait (Figure 6.2) defines some functions (read, obligations) for each type T that implements it:

- `view_equal` defines the equivalence relation ($\simeq$). For VMarshal, we only need to show that the defined relation is symmetric (`lemma_view_equal_symmetric`).
- `marshallable` defines a specification (mathematical definition) function that limits which elements of the type T support marshalling. This allows us to restrict to (say) marshalling only types whose serialized form is less than 2^{64} bytes. The function `is_marshallable` is the run-time executable version of `marshallable`.
- `ghost_serialize` defines the mathematical serialized representation (as abstract sequences of bytes, `Seq<u8>`) of an element of T, if it is `marshallable`.
- `serialize` appends the serialized representation of a `marshallable` value to a growable set of bytes (`Vec<u8>`), proving that it changes nothing else in the bytes, and appends exactly the `ghost_serialize`d data.
- `deserialize` parses bytes from an offset, returning the parsed value and ending offset if it succeeds. It guarantees that the parsed value serializes to the exact bytes that it has parsed, proving one side of the round-trip properties.
- `lemma_serialization_is_not_a_prefix_of` proves the strong-prefix property.
- `lemma_same_views_serialize_the_same` proves that the equivalence relation is "reasonable"; specifically, $\forall (x_1, x_2 : T).\, x_1 \simeq x_2 \implies s(x_1) = s(x_2)$.
- `lemma_serialize_injective` proves the converse; specifically, it shows the injectivity of `ghost_serialize`—$\forall (x_1, x_2 : T).\, s(x_1) = s(x_2) \implies x_1 \simeq x_2$. This, along with the property proven by `deserialize` finishes the round-trip.

For primitive types, such as `u64`, hand-written implementations discharge the implemen-

```
pub trait Marshallable : Sized {
  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);
  spec fn marshallable(&self) -> bool;
  exec fn is_marshallable(&self) -> (res: bool)
    ensures res == self.marshallable();
  spec fn serial(&self) -> Seq<u8>
    recommends self.marshallable();
  exec fn serialized_size(&self) -> (res: usize)
    requires self.marshallable(),
    ensures res as int == self.serial().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.serial();
  exec fn deserialize(data: &Vec<u8>, start: usize) -> (res: Option<(Self, usize)>)
    ensures res matches Some((x, end)) ==> {
        ∧ marshallable(x)
        ∧ start <= end <= data.len()
        ∧ data@.subrange(start as int, end as int) == x.serial()
    };
  proof fn lemma_serialization_is_not_a_prefix_of(&self, other: &Self)
    requires
      !self.view_equal(other),
      self.serial().len() <= other.serial().len(),
    ensures
      self.serial() != other.serial().subrange(0, self.serial().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.serial() == other.serial();
  proof fn lemma_serialize_injective(&self, other: &Self)
    requires
      self.marshallable(),
      other.marshallable(),
      self.serial() == other.serial(),
    ensures
      self.view_equal(other);
}
```

Figure 6.2: The `Marshallable` trait.

```
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 6.3: 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.*

tations and proof obligations for `Marshallable`. The implementation of `Marshallable` for `Vec` is also hand-written (once and for-all). Due to Rust's trait support, this implementation is parametric in the type of elements of the vector, and thus, `Vec<T>` automatically implements `Marshallable` whenever `T` is `Marshallable`.

Arbitrary `struct`s and `enum`s can automatically implement `Marshallable` through macros. Figure 6.3 shows an example of how a user of the framework can use one such macro to automatically derive `Marshallable` for a user-defined `enum`. The `#[..]` annotations guide, and provide fresh identifiers for the automatically generated proofs. These are currently a compromise due to the current `macro_rules!`-based definition of `derive_marshallable_for_enum`. A procedural macro [122] would allow these annotations to become optional.

Both macros `derive_marshallable_for_enum` and `derive_marshallable_for_struct` work in a syntax-driven fashion, automatically deriving both the necessary implementations and proofs for the functions necessary for the `Marshallable` trait, by invoking the relevant functions of the constituent elements. For example, a `struct`'s parser consists of sequentially parsing each of its fields, with even a single failure causing immediate failure of the entire `struct`.

An additional macro, `marshallable_by_bijection` helps automatically implement

```
marshallable_by_bijection! {
    [EndPoint] <-> [Vec::<u8>];
    forward(self) self.id;
    backward(x) EndPoint { id: x };
}
```

Figure 6.4: `Marshallable` by proving a bijection

`Marshallable` by proving a bijection with a type that already implements `Marshallable`. To use this macro, the user must write forward and backward functions, which the macro then automatically verifies to be inverses, and then uses to implement `Marshallable` by inserting the necessary stubs for each of the functions and proofs. Figure 6.4 shows an example usage of `marshallable_by_bijection`. While `derive_marshallable_for_struct` would have been perfectly suitable for such a `struct`, the proof by bijection provides versatility in more complex cases by allowing some finer control over the produced output; in particular, it explicitly documents and proves that the serialized format of the two types is identical and inter-changeable.

6.5 Vest: A Framework for Extrinsic Data Formats

Our framework for extrinsic data formats, called Vest, is also a library in Verus; however, it takes a different approach. At a high-level, Vest is a parser generator library that uses a domain-specific language (DSL), inspired by Nail [9], as its input, and produces verified serializers and parsers using parser combinators. The proof-producing compiler from the DSL to the Verus code and proofs does not need to be trusted, nor verified.

Following the structure of subsections in VMarshal, below, we describe the high-level correctness properties provided by Vest, followed by the design and implementation that provides them.

6.5.1 Verified Properties

Vest proves a number of properties, some reminiscent of those in VMarshal (Section 6.4.1), and others going beyond.

The round-trip property through serializing and parsing (i.e., $\forall (x : T).\, p(s(x)) \simeq x$) is of course, fundamental; thus Vest proves it on all data formats.

The other direction of the round-trip (i.e., $\forall(b : \text{u8}^*).\, s(p(x)) = b$) depends upon the specific grammar of the data format in use. In particular, if the data format allows for ambiguities in serialized representation (i.e., there exist two different serialized representations b_1 and b_2 such that $p(b_1) = p(b_2)$) then obviously, the parse-serialize round-trip itself cannot hold. Such a grammar is considered to be *malleable*. When possible, Vest proves non-malleability (i.e., $\forall(b_1, b_2 : \text{u8}^*) p(b_1) = p(b_2) \implies b_1 = b_2$), but it also supports malleable grammars. In particular, the user can choose to have a weakened form of bijection by allowing malleability through *constants* (for example, in a text-based protocol, the high-level constant `WHITESPACE` could refer to any amount of whitespace). Any parts of the format that are non-malleable would *still* be proven non-malleable; the malleability only "infects" parts of the grammar that are malleable.

Similar to non-malleability, when possible, Vest additionally proves the strong-prefix property (i.e., $\forall(x_1, x_2 : T).\, x_1 \not\approx x_2 \implies s(x_1) \neq s(x_2)[..|s(x_1)|]$); and this too is proven on as many parts of the format that it holds on.

The above properties are all similar to those in VMarshal. We will discuss this further shortly in Section 6.5.2, but Vest's design means that rather than dealing with complete byte-strings directly (as VMarshal would), it instead deals with byte-*streams*; this means that it must show a few additional properties. Byte-streams are essentially a slice into a (possibly larger) byte-string; and multiple byte-streams might be in use on a particular byte-string. Thus, some of the additional properties that Vest must show deal with well-behavedness of such streams (e.g., similar to a framing property, serializing into a stream does not impact the underlying byte-string anywhere except the stream itself), and determinism (for applicable grammars, serializing onto a stream is independent of the location of the stream within the underlying byte string).

In short, Vest attempts to prove as many properties as it can show to hold on any particular extrinsic data format; if properties only hold for a subset of the grammar, then it proves it for that subset. This allows users of grammars with stronger properties to take advantage of them, while still supporting grammars that perhaps were not designed with such properties in mind.

6.5.2 Vest: Design and Implementation

We have designed Vest as a compiler from a DSL to Verus code and proofs.

The DSL, inspired by Nail [9] (an unverified parser generator library), helps define an extended Parsing Expression Grammar (PEG) [34], describing both the low-level data format and the corresponding higher-level data types.

Figure 6.5 shows an example grammar defined in our DSL. Here, `alnum` defines a parser/serializer pair for alpha-numeric characters by specifying the underlying type (`u8`), as well as a predicate to restrict it to only valid characters. Next, `element` defines a parser/serializer pair for one-or-more alpha-numeric characters (`Vec1` rather than `Vec`). Following this, `domain` defines a parser/serializer pair for domains, which consist of one-or-more elements separated by the dot character. Finally, the `defn` defines a struct consisting of a domain, followed by an ordered choice; this ordered choice is equivalent to a high-level tagged union. The `wrap` combinator, as in Nail, allows for conveniently supporting constants in serialized data *without* introducing indirection in the high-level application types (i.e., it flattens what would be a trivial `struct`).

The compiler from the VEST DSL to Verus, inspired by EverParse [119, 137], is neither verified nor need be trusted. It uses pest [25] to parse the DSL, and produces Verus code and proofs, driven entirely syntactically. The produced Verus parsers and serializers are written in the form of parser combinators. Each produced serializer, parser, and proofs for the various properties (Section 6.5.1), can depend on each other recursively, until eventually bottoming-out to a set of hand-written verified implementations for basic primitive combinators (such as `pair`, which represents a high-level pair, represented at the lower-level as a concatenation).

The executable serializers and parsers work with byte-streams, which are an abstraction of slices on a byte-string. We do this to eventually support the introduction of arbitrary (verified) transformations across streams, where a stream can refer to another stream, similar to the unverified trusted transformations in Nail. Byte streams and transformations allow the introduction of useful functionality such as checksums to be specified directly within the DSL, without necessitating an extra level of proof to be written after parsing and receiving *both* the data and its checksum. For now, support for this feature is still in an early stage.

With respect to the actual implementation, VEST initially began with the executable operations (parsing and serializing) consuming and producing streams. For certain applications, this can have a non-trivial overhead due to allocation and copying. Thus, we are in the process of introducing in-place versions of each executable operation, thereby making the framework allocation- and copy-free, wherever possible.

```
...
alnum = u8 | {'a'..='z', '0'..='9'}
element = Vec1<alnum>
domain = Vec1<element> | u8 = '.'
defn = {
  hostname: domain,
  r: choose {
    A(wrap(u8 = 'A', ipaddr)),
    CNAME(wrap([u8; 4] = "CNAME", domain)),
    MX(wrap([u8; 2] = "MX", domain)),
  }
}
```

Figure 6.5: Example usage of the VEST DSL.

6.6 Related Work

Parsing and serializing of data formats has a rich and long history.

Intrinsic data format parsing support has existed in multiple programming languages. For example, Python supports it via pickle [36] and marshal [35]. Specific to Rust, the most popular approach is serde [26], a generic *ser*ializer and *de*serializer of Rust data structures, which uses macros and Rust's trait system to provide serializers and parsers to/from popular formats, such as JSON or TOML. While the user chooses the specific format, the schema is driven entirely by the application-level data types, thereby making for a good user experience. Our marshalling framework, VMARSHAL, takes some inspiration from this good user experience, but also additionally provides provable guarantees.

Extrinsic data formatting is more popular. Nail [9] is a security-focused tool for parsers and serializers for extrinsic data formats. Nail uses a DSL to define not just the data format, but also the higher-level objects modeled by the data; it also introduces *dependent fields* and *stream transforms* to help it express protocol features not supported by prior approaches. Our framework, VEST, takes inspiration from the unverified Nail tool and its grammar for our DSL, and builds upon its ideas to provide provable guarantees with verification.

EverParse [119] is a framework for generating parsers and serializers in F* and Low*. EverParse uses an untrusted and unverified compiler to parse tag-length-value protocol formats and produce code and proofs that use parser combinators. The code is specifically designed to yield zero-copy parsers when compiled from Low* to C. They prove safety (no use-after-free, no integer overflows), functional correctness (round-trip inverses), and non-malleability (valid messages have unique representations). EverParse3D [137] is an evolution of EverParse that focuses on parsers and validators aimed at usage in an unverified C/C++ codebase for hardening attack surfaces. They expand on the allowed set of formats, but forego the serializers. VEST takes inspiration from both of these, supporting greater expressivity than EverParse, and additionally supporting automatically deriving serializers, unlike EverParse3D.

Comparse [144] focuses on secure formats for cryptographic protocols. By focusing on a specific use case, they allow the user to focus on crucial cryptography-specific properties, such as the absence of data-dependencies, which help protect against protocol-confusion attacks [85].

Narcissus [23] is a Coq library that supports deriving parsers and serializers in the style of parser-combinator libraries. It uses Coq tactics to derive these, and proofs that they form inverses of each other. A user of Narcissus works with these combinators directly in the language, while a VEST user writes in a DSL, allowing for greater separation of concerns.

In a different direction, Jourdan et al. [55], build a formally-verified parser validator in Coq, which can check if a context-free grammar and a parser (written as a finite state automaton) agree. This requires hand-rolling a parser, and does not directly connect to a serializer, but can be used complementary to other approaches such as ours.

6.7 Concluding Remarks

In this chapter, we have explored how splitting the task of dealing with low-level data formats into *intrinsic* and extrinsic formats helps provide convenience to programmers using these frameworks. In addition, this split allows for the developer of each framework to focus on a clean design for it, unburdened by the competing concerns of the other.

For VMARSHAL, in the future, it would be interesting to explore support for easily switching to alternate high-level formats, similar to what is possible with the (unverified) serde crate [26]. Additionally, the proofs required for writing parsers and serializers require a lot of “hand holding” with respect to sequences and higher-order functions. It would be interesting to explore techniques to simplify these straightforward but tedious proofs.

For VEST, it would be interesting to explore the complexities in building a non PEG-based approach. In particular, the ordered-choice operator gives us significant convenience in the implementation of the parser, but leads to a more complex round-trip proof; perhaps, a different approach might hit a better sweet-spot for the proofs. Also, we are exploring the implications of stream-transformation functions (as in Nail) to understand their implications on verified serializer and parser frameworks. Additionally, it may be interesting to further strengthen the properties provided by Vest to also prove equivalence of the implementation to the provided extrinsic format, as specified in the DSL, by having formal semantics of the DSL itself.

More broadly, it would be interesting to explore parsers in various verification languages derived from the same grammar. Parser differentials (inputs that parse differently on ostensibly equivalent parsers) are a notorious cause of security bugs, and it would be interesting to see if we can (at minimum) prevent them from happening if two communicating programs are verified (despite being verified in different languages).

In summary, in this chapter, we have seen a principled design for two formally verified frameworks—one for intrinsic and the other for extrinsic data formats—which automatically generate verified parsers and serializers. With these, we can safely deal with arbitrary untrusted data without apologizing for developer time, approachability, or expressivity.

Chapter 7

Dealing with Legacy Code

"She managed to come up with the kind of predictions that you can only understand after the thing has happened", said Anathema. "Like 'Do Notte Buye Betamacks.' That was a prediction for 1972."

Good Omens
Neil Gaiman *and* Terry Pratchett

So far, we've been dealing with situations where we are working with *relatively* new code. However, when securing software systems, one must often also deal with software where the source is unavailable or even unusable, and thus we must rely only on the executable/binary code. Unfortunately, despite decades of advancement in the art and science of decompilation (i.e., reverse compilation), a lack of high-quality source-level types plagues decompiled code. Accurate type information is crucial in understanding program behavior, which is crucial towards securing it. Unfortunately, existing decompilers rely heavily on manual input from human experts to improve decompiled output. We propose TRex, a tool that performs automated deductive type reconstruction, using a new perspective that accounts for the inherent impossibility of recovering lost source types. Compared with Ghidra, a state-of-the-art decompiler used by practitioners, TRex shows a noticeable improvement in the quality of output types on 115 of 125 binaries. By shifting focus away from recovering lost source types and towards constructing accurate behavior-capturing types, TRex broadens the possibilities for simpler and more elegant decompilation tools, thereby making it easier to analyze and understand binary code.

Contributions. The work presented in this chapter is primarily mine, and will be part of a paper on TRex [14].

7.1 Introduction

Reverse engineering of machine code is useful in many contexts, including binary hardening [156], malware analysis [158], program comprehension [17], vulnerability discovery [78], and more. Many tools have been built to assist reverse engineers, including disassemblers, debuggers, and decompilers. Decompilers, such as Ghidra [106], Binary Ninja [141], and Hex-Rays [126], are popular amongst reverse engineers, since they provide a high-level source-code-like view of low-level machine code.

Unfortunately, despite decades of advances in the science and art of decompilation, the quality of decompiled output leaves much to be desired. A particularly persistent, unsolved problem is providing high-quality source-level type information for decompiled code. These types benefit both human understanding and the decompiler itself in producing improved output [99]. However, state-of-the-practice decompilers often struggle to recover any meaningful types from binaries (Section 7.5.1). Instead, they heavily rely on human experts to manually provide better type information. This is perplexing, given the many years of academic research devoted specifically to producing high-quality source-level types. We speculate that this gap between practice and academia may be attributable to factors including: the unavailability of academic tools, the use of inconsistent benchmarks, and the inherent complexities of decompilation not fully captured in research papers.[1]

To explore this gap in decompilation performance, we decided to develop a new tool to perform better automated deductive type inference. We focus on deductive inference, rather than on approaches based on machine learning [16, 17, 81], since the latter can be hard to depend on for correctness in scenarios that differ drastically from their training corpus (Section 7.2.1). As a step towards reversing the unfortunate tendency of closed-source tools and non-reproducible benchmarks, we intend to open-source our tool and release reproducible scripts for the benchmarks.[2]

As a starting point, we analyzed the requirements for automated initial type inference from the perspective of practitioners in the field. Here, we discovered a crucial mismatch in

[1]Indeed for one paper [105], other employees at the same company attempting to reproduce the work state “It is a powerful system but difficult to understand” and the “presentation is very dense and subtle” [41]

[2]`https://github.com/secure-foundations/trex`

expectations—prior techniques were attempting to recover source-level types, while reverse engineers "merely" desire output that captures the behavior of the program. Even worse, *type recovery is often impossible*—in Section 7.2.2, we construct a collection of pairs of C programs where each element of a pair has drastically different types from its partner, but both programs compile to the exact same assembly code; hence, neither element of the pair can be *the* valid ground truth. Rather than attempt the impossible, we believe that the goal must shift to the construction of accurate behavior-capturing types. We call this new perspective Type Reconstruction (Section 7.2.3).

We adopt this new perspective in constructing TRex,[3] a tool for automating the inference of source-level types for binary code. Building from the perspective of Type Reconstruction has fundamental consequences for the design of type inference tools, and in TRex's case, results in a noticeable quality improvement in the C-like types it produces, compared to popular state-of-the-art decompilers.

Focusing on Type Reconstruction aids TRex in various ways. For example, since C is the de-facto output language for decompilers, many prior tools succumb to the temptation to use C-like types internally during analysis. However, Type Reconstruction tells us that C-like *nominal* types are ill-suited to the task, and instead we must use *structural* types (where available behaviors/features of the type define it; not to be confused with `struct` types) for the analysis even if the output types are in C, thereby better capturing what reverse engineers expect to see from their tools (Section 7.3.1).

Thus, TRex internally uses and can produce types that are far more expressive than C; these types can be used for further downstream analyses. It also produces human-readable C-like types projected from the more precise internal machine-readable types, through a separate analysis phase. Phase separation is not limited to just this distinction between machine-readable and human-readable types, but extends further, both in the internal design of our approach, and also how it integrates with over-arching binary analysis frameworks. TRex makes only a small number of assumptions on its input and output, making it easy to adapt to 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, thereby supporting all architectures supported by Ghidra, but also note that Ghidra-specific code only consists of approximately 1000 lines of primarily boilerplate code.

In the process of constructing TRex, we have encountered challenges and discovered

[3]TRex: **T**ype **R**econstruction for **ex**ecutables

insights about type reconstruction that to the best of our knowledge, are either novel, or are tacit "oral tradition" in the community and remain unpublished. These insights help drastically reduce the complexity of the design and implementation of the tool, while still providing expressive, high quality output. For example, binary analyses must actively manage conservativeness—a fully conservative tool would be a surprisingly bad idea (Section 7.3.2). Additionally, in exploring the difficulties of key phases of Type Reconstruction, we have discovered an algorithmic hardness result for a phase we call *type rounding*, which we show to be NP-Hard (Section 7.3.3.5).

In Section 7.5, we evaluate TRex. Given our insight regarding the lack of a singular valid ground truth, we evaluate TRex both qualitatively and quantitatively to better understand its benefits in comparison against other tools available to practitioners. Our quantitative evaluation uses a new metric that attempts to capture the expectations of reverse engineers, while working around the inherent difficulty of objectively evaluating type reconstruction. TRex outperforms the open-source state-of-the-art on a collection of benchmark binaries picked by prior work (that we create reproducible variants of, to facilitate future comparisons), achieving an average score that is 15.81% higher than Ghidra, which itself achieves a 26.46% higher score than a trivial baseline.

Overall, this chapter makes the following contributions:

1. We propose Type Reconstruction, a new perspective on automated type inference from binary code, which accounts for the impossibility of recovering lost source-level types.
2. We build a new open-source tool, TRex, which takes arbitrary lifted disassembly from any architecture and produces C-like types (covering all C primitives and aggregate types, including structs, arrays, unions, and recursive types) for human analysts, and more-detailed machine-readable output for downstream analyses.
3. We discover and document useful insights, such as the (im)possibility of traditional ground truth for type reconstruction, algorithmic hardness results, and more.
4. We demonstrate an improvement on 115 of 125 benchmark binaries compared with existing state-of-the-practice. We also propose a new metric that better captures output quality from a reverse engineer's perspective.

In short, in this chapter we demonstrate that principled design helps us produce a new formalism for better comprehension of binaries that need not apologize on either elegance, output expressiveness, or quality.

Type . . .	Multi-Language	Behavior-Capturing	Technique
Inference		∼	Deductive
Prediction	✓		Learning
Recovery	✓		Deductive
Reconstruction	✓	✓	Deductive

Table 7.1: Our Categorization of Automated Type Inference for Binary Code. *See Sections 7.2.1 and 7.2.3 for details.*

7.2 Automated Type Inference for Binary Code

Automated source-like type inference from binary code has been explored for decades. Caballero and Lin [15] provide a detailed survey of techniques leading up to 2016, categorizing approaches along multiple axes: static vs dynamic, value vs flow-based, primitive vs nested types, etc. In this section, we propose an alternate categorization—summarized in Table 7.1. We explore prior approaches and their core underlying flawed assumption of ground truth, finally leading us to our proposed new perspective—Type Reconstruction.

7.2.1 Prior Approaches

Type Inference, when not qualified with any modifiers (such as "from binary code"), is well studied for programming languages, and commonly refers to the single-language task of automatically inferring types given a source program. It has been incredibly useful in practice for programming; however, due to its assumption of a single language, techniques from it, such as Hindley-Milner type inference [47, 93] are not (directly) applicable to decompilation, which focuses on situations with two languages in play.

Some recent techniques [16, 17, 81] for automatically recovering source-like types from compiled code utilize learning-based approaches. We call such techniques *Type Prediction*. Despite showing increasingly impressive success at predicting source-level types, it can be hard to depend on the correctness of predicted output, especially in scenarios that might differ drastically from their training corpus. Fundamentally, such techniques come with no guarantee about usage on out-of-distribution binaries. If the usage scenario is similar to the ones they are trained on, then there is some expectation of good results. Unfortunately, most reverse engineering happens on source-unavailable code where it would be difficult to make the assumption that the target is in-distribution. This problem perhaps reaches its most extreme for malware, where training data is almost always unrepresentative of the binary being analyzed.

Thus, we focus on reasoning-based (*deductive*) techniques going forward. These techniques derive 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 between the accuracy and conservativeness of the types produced. While rarely (if ever) explicitly concretized as such, the results of these approaches can, in theory, be traced through a series of deduction steps, and thus produce reasonable types.

We use *Type Recovery* to specifically denote deductive approaches that attempt to recover the source-level types in the program that was compiled to the target binary. This is the primary focus of almost all the techniques in the aforementioned survey [15].

A key observation of our work is that all existing approaches for Type Prediction and Type Recovery attempt to recover *the* source-level types from the original program, which we demonstrate is fundamentally impossible.

7.2.2 On the Impossibility of Type Recovery

Type Prediction and Type Recovery both assume that there is an objectively correct ground truth. Indeed, it seems obvious that if some initial source code is compiled to a binary, then that original source code should provide the ground truth for its compiled machine code. Yet, this view on type inference (or decompilation, more generally), is fundamentally flawed. We demonstrate this through a collection of hand-crafted examples that show how severe the lossiness of the compilation process really is.[4] Furthermore, this lack of a singular ground truth implies that source *recovery* (even modulo comments and naming) is impossible. We believe that attempting to achieve the impossible leads one down a rabbit hole of increasingly sophisticated and complex techniques to recover the specific kinds of types that appear to be common in specific evaluations. Indeed, recent work [84, 105] on Type Recovery has continued to show increasing sophistication, both in

[4]Tested with GCC 13.2 with `-O2` on x86/64

```c
uint8_t foo(uint8_t x) {
  return ~x;
}
int32_t bar(int32_t x) {
  return ~x;
}
```

(a) Confounding sizes

```c
char* foo(char* x, size_t i) {
  return &x[i];
}
int64_t bar(int64_t a, int64_t b) {
  return a + b;
}
```

(b) Confounding pointerness

```c
int foo(int x) {
  volatile int y = x;
  volatile int z = x;
  return y + z;
}
struct S { int y; int z; };
int bar(int x) {
  volatile struct S s;
  s.y = x;
  s.z = x;
  return s.y + s.z;
}
```

(c) Confounding stack shape

```c
uint32_t foo(uint32_t n) {
  uint32_t r = n;
  while (n-- > 0) {
    r++;
  }
  return r;
}

uint32_t bar(uint32_t n) {
  return n * 2;
}
```

(d) Confounding operations

Figure 7.1: Confoundable C types. *Each pair of functions compile to the same assembly code, demonstrating the flaw of assuming a single ground truth.*

techniques and the expressivity of types produced. At its extreme, this could effectively become a collection of idioms hyper-specialized to a particular (version of a) compiler. Nonetheless, even this recent work suffers from the same common assumption.

Figure 7.1a demonstrates a pair of functions that differ in sizes of arguments and return types, yet compile to the exact same assembly code. Similarly, Figure 7.1b show a pair of functions that differ on their usage of pointers, yet compile to the same machine code. Colocation of variables on the stack can be intentional (using a `struct`) or incidental, as shown in Figure 7.1c. Clearly, memory usage of "the" ground-truth cannot be relied upon. Worse, optimizations can confound operations performed on variables too—arithmetic operations, or even entire loops, can be switched out (Figure 7.1d). Finally, code with undefined behavior (UB) allows the optimizer to wreak havoc.

With all these examples, it can initially feel like all hope of type inference is lost.

However, reverse engineers often do not require perfect type recovery.

7.2.3 Type Reconstruction: Capturing Behavior

Recognizing that recovering *the* ground truth is impossible, we turn to understanding what reverse engineers really want from decompilation. We argue that rather than perfect recovery of source, in practice, reverse engineers desire output that captures the binary's observable behavior, i.e., a summary of observed/allowed operations on each variable. Types are "merely" an encoding of this information into an easily digestible form. Type inference, even if the produced types are wildly different from the original types, is still useful when they are compatible with the observed behavior of the program, since this allows the reverse engineer to understand what is *actually* being executed, or how to interface with it.

Hence, we propose a new perspective—*Type Reconstruction.* Like Type Prediction and Type Recovery, Type Reconstruction takes in low-level code and produces high-level types. In contrast to the prior approaches however, it does not attempt to infer or recover *the* ground truth, but instead focuses on *(re)constructing* types compatible with observed behavior.

Specifically, the goal of Type Reconstruction is to construct the "nearest" source-level types that capture observable behaviors, using deductive techniques and thereby producing types that can, at least in theory, be reasoned about. We believe that adopting this more realistic goal is essential to improving type inference in practice. This goal naturally accounts for different compilers' interpretation of source code and does not attempt to codify any particular compiler's behavior(s). Instead, it focuses on providing types that are useful to reverse engineers.

Recognizing that there is no singular ground truth does come with a non-trivial downside—we must redefine what it means for a Type Reconstruction tool to produce "correct" or "accurate" types. In short, how should one evaluate the effectiveness of Type Reconstruction? We detail our specific evaluation further in Section 7.5, but morally, we attempt to capture features of types that reverse engineers would prefer the tool got correct.

While priorities on different aspects might differ, we believe that there are some common aspects that reverse engineers find helpful to know from their types. For example, whether a variable (register or piece of memory) can be dereferenced (i.e., is a pointer) is crucial; knowing if something is a `struct`, and if so, understanding fields within it is important; knowing if some memory has a function pointer could be very important for vulnerability analysis. We also note that even if the original source code did not have a `union`, reconstruction of a `union` can help surface memory reuse that can help with discovering certain vulnerabilities, such as type confusion bugs. Thus, any evaluation must capture these behaviors. In Section 7.5.2, we propose a metric that attempts to quantitatively capture these preferences. However, given the differing priorities of reverse engineers, a single number or metric is insufficient, and thus we believe we must also observe improvements qualitatively (Section 7.5.1).

7.3 TRex: System Design

TRex is the first tool explicitly designed with the goal of Type Reconstruction. We describe some of our high-level design decisions, followed by TRex's architecture below.

7.3.1 Structural Types Capture Behavior

Since the de-facto output language of decompilers is C, it is quite tempting to use C-like nominal types during Type Reconstruction. However, we believe that this would be a mistake, leading to the need for complicated mechanisms, such as those used in prior works on Type Recovery. As an example, we might discover that a certain location supports 64-bit integer addition; while this location could be thought of as an `int64_t`, `uint64_t`, or `undefined64`, if the analysis were to internally represent this location as any of these, then it is necessarily introducing inaccuracies; in particular, both the `int_t` types claim that the type does not support pointer dereferencing, while the `undefined64` (even though it captures the size correctly) does not capture the observation that the type supports integer addition. This means that later analyses that use these types cannot rely upon them and must either rely on side information, or re-analyze the code that led to the observation of the 64-bit addition.

We argue instead that the types most natural to Type Reconstruction are behavior-capturing types, i.e., *structural types*.[5] In particular, rather than attempting to match behaviors in the executable to C-like types, the types themselves need to capture behaviors precisely, even if the behaviors could not be represented as C-like types. Structural types are freed from the constraints of human-readability, and working with structural types

[5]In this paper, we are drawing a contrast with nominal types, where structural types mean a static version of duck-typing [118]. This usage of structural types should not be confused with the alternative definition used in some PL circles to contrast with sub-structural type systems.

```
StructuralType {
  COPY_SIZES  {8, 16, 32}
  INTEGER_OPS   {
    Add_32, Sub_32, Mult_32, UDiv_32, SDiv_32, URem_32, SRem_32,
    And_32, Or_32, Xor_32, Eq_32, Neq_32, ULt_32, SLt_32, UCarry_32,
    ...
  }
  POINTER_TO None
  COLOCATED_STRUCT_FIELDS None
  ...
}
```

Figure 7.2: Example Structural Type. *Simplified for presentation purposes. Equivalent to a signed 32-bit integer (`int`) in C.*

as far as possible during type reconstruction allows us to maintain high precision and conservativeness (Section 7.3.2).

Figure 7.2 shows an example structural type, equivalent to C's `int` type. Clearly, exposing the full precision of structural types to users is untenable and would detract from understanding. For example, if a type supports 64-bit addition, subtraction, multiplication, right-shift, and more, then for practical purposes, it is reasonable to believe that it would support other operations such as division too, and that it is best shown to humans as an `int64_t`. Thus, Type Reconstruction, despite best conducted with structural types, must still deal with nominal C-like types. Since the structural types are more precise, we capture this through a phase that performs "type rounding" (introducing the division operation, in the above example), followed by projection to C-like types (Section 7.3.3.5).

Overall, we make the design decision to stick with structural types *as far as possible*, only switching to nominal types in the last stages of reconstruction. This maintains a high degree of fidelity with the actual observable behavior of the machine code, as far as possible. The output of our tool can be consumed either by a downstream tool (which can use the more accurate structural types) or by a human (who can use the easier-to-read nominal types).

In addition to using structural types, decisions must still be made regarding the expressivity of such structural types. We choose to support both aggregate and recursive types, since their expressivity better captures machine-code behavior, compared to only supporting primitive types. However, we do not support polymorphism (neither parametric nor ad-hoc). This is because we have observed polymorphism to be useful only in a small number of toy examples—in practice, compilers monomorphizing and optimizing code leads to sufficiently different code and types, warranting separate attention rather than collapsing via polymorphism.

7.3.2 Conditional Conservativeness

Binary analysis is difficult, even undecidable in many cases [67], and Type Reconstruction is no exception. Tools thus must make tradeoffs between soundness and completeness—put differently, there must be a balance between the conservativeness and utility of the output from a tool. Naïvely, one might wish for a tool that never makes any non-conservative leaps, so that it cannot mislead (except by omission). However, a hypothetical fully-conservative analysis would quickly find itself being unable to provide anything useful.

For example, even a single `call` instruction would lead to the halting problem via "does the callee return?"; thus, a fully conservative analysis might not be able to progress beyond a `call`. Nonetheless, we would like at least some guarantees from our tools. Thinking about the composability of guarantees from analyses, we realize that *conditional conservativeness* appears to be a good tradeoff (especially in the long run, as various analyses are built and improved). We define conditional conservativeness to mean conservativeness only under the condition that the "upstream" analysis provides true facts. Said differently, a conditionally conservative analysis will output only true facts within the axiomatic system set up by the upstream analysis. Returning to the example situation of the `call` instruction, an upstream analysis is in charge of reachability, and the downstream analysis (say, Type Reconstruction), only needs to be conservative if the upstream analysis produced true facts.

Unfortunately, even conditionally conservative analysis is untenable in practice, since it still completely cuts off "obvious" inferences that are technically non-conservative. As an example, if a type is seen to support division, it is reasonable (but non-conservative) to assume it supports the modulus operation.

In light of these observations, while it might seem tempting to give up on conservativeness (even conditional), we believe that there is a better approach—managing the loss of conservativeness. In particular, tracking (and supporting the toggling of) the introduction of non-conservativeness introduced by different analyses becomes essential to building larger analyses and tools. Specifically, tools may make opportunistic inferences, but they must also support disabling those inferences. We believe that this design forms a useful blueprint for various binary analyses to adopt, in order for us to improve each component without compromising the guarantees provided by the overall composition.

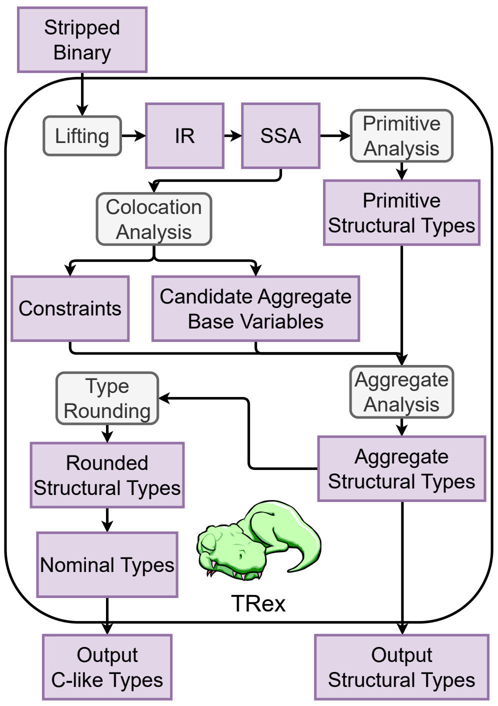


Figure 7.3: Phases in TRex

For TRex, we choose a largely conservative stance for decisions made by the tool, with some opportunistic inferences when purely conservative inference would be incredibly unhelpful. This means it can take advantage of common patterns, improving the default utility of its output, while supporting a more conservative analysis when an analyst (or another analysis) discovers that a particular opportunistic inference is being unhelpful. For example, after a `call` instruction, without a global analysis confirming that no memory corruption vulnerabilities exist, any *purely* conservative analysis cannot assume that the stack pointer is returned back to where it was prior to the call. This clearly would not be helpful for most executables. Thus, in TRex, we opportunistically assume that the stack pointer is indeed returned to the expected value. However, we also recognize that this introduces non-conservativeness, and this opportunistic assumption (like all other such similar introductions of non-conservativeness) can be disabled with a flag.

7.3.3 TRex Architecture and Analysis Phases

TRex consists of multiple phases that we summarize in Figure 7.3 and describe below. Analysis begins by lifting from disassembly, and it is flow-sensitive, but path- and context-insensitive. We additionally choose to perform type reconstruction intra-procedurally—

we do not yet propagate types inter-procedurally (although this could be added with straightforward, albeit non-trivial, engineering work). The output from TRex is either C-like types (useful for human analysts) or precise-but-verbose structural types (useful for downstream analyses that can aid in better decompilation [99]).

7.3.3.1 Input, Lifting, and SSA

An important decision is the choice of input to a Type Reconstruction tool. A plausible input could be from an existing decompiler (potentially with type information stripped away, if the decompiler already does some type inference), since that allows one to take maximum advantage of the rest of the analyses that already exist in the decompiler. Another plausible input is the disassembly (or somewhat equivalently, an architecture-independent lifting of it), since that gives the highest level of detail about what is actually happening at the machine level. Of the two, we opted for the latter, so as to obtain a higher degree of assurance in our tool's output. In practice, TRex takes disassembly-equivalent lifted code from a binary analysis framework as input, and lifts it to its own internal intermediate representation (IR) that we discuss further in Section 7.4.1. From this IR, to consistently track variables without worrying about mutation or (immediate) aliasing, we compute the static single-assignment (SSA) form of the code. Future stages reconstruct types for all variables produced in this stage.

7.3.3.2 Primitive Analysis

We begin the type-related analyses in TRex with primitive analysis, which performs a walk through the SSA IR. During this walk, we start to assign types to each variable (initialized with the empty type at first) by iteratively adding observed operations to relevant types (such as "this variable supports 32-bit addition"). This begins to provide us with the primitive size and operation information about the variables that will be useful for future phases. Also, at this point, we might merge some types together, either always (for SSA ϕ-nodes), or opportunistically (depending on the operation performed, e.g., comparison operations). In this phase, we also perform a global value numbering pass to identify further opportunities to merge types.

Naïvely, one might expect a lot of merging, but we note that most operations do not cause merging. For example, an instruction like $a < b$ requires both a and b to be the same type; but $a \leftarrow b + c$ does not necessarily mean that a, b, and c are the same type—the addition operation is as valid for pointer offsets as it is for integers.

7.3.3.3 Colocation Analysis

Having recognized primitive structural types, we begin collecting constraints that will help us discover aggregate types. These constraints recognize a dereference of a variable at an offset from another variable. Representing the constraints explicitly, rather than implicitly by combining this phase with the next, simplifies the implementation of both. The constraints themselves store both a static (constant) or dynamic (symbolic variable) offset, along with the base variable and the accessed variable. Dynamic offsets provide an indication of scanning across memory, which helps identify arrays. The constraints themselves may be derived either from a single instruction, multiple instructions, or even a collection of constraints and instructions. To aid this phase, we also implement an on-demand constant-folding analysis to identify static offsets that might otherwise be imprecisely considered dynamic. Along with the constraints, from the colocation analysis we also obtain the base variables for candidate aggregate types.

7.3.3.4 Aggregate Analysis

Using the constraints and types from prior analyses, the aggregate analysis recognizes when different types are consistently colocated, and thus can be grouped into the same `struct` or array. It finalizes the offsets and sizes of structural types, including non-aggregate types. The structural types it is identifying are analogous to C `struct`s (including support for flexible array members—unsized arrays at the end of a `struct` [52]) and C arrays.

Unfortunately, we find that in the general case, this phase is impossible to determine precisely, at least with static analysis. For example, if a function takes a pointer to a `struct` of two integers, and only touches the first, then it is impossible to even detect that there was any colocation possible. Worse, simply because the compiler places two variables beside one another consistently, this may not reflect programmer intent. Instead, colocation of variables is influenced greatly by the compiler's interpretation of memory. To accurately tease apart such variables, one needs to find a contradiction to the colocation hypothesis. A naïve fix is to instead only consider colocation when there is an obvious offset operation in play, recognizing that compilers rarely perform such offsets unless they are from a single `struct`. However, this cannot work in practice, since compilers often over-read bytes, or even cross across fields in `memcpy`-like operations, causing spurious fields of invalid sizes to appear. Recognizing this inherent difficulty in aggregate analysis, and that a purely conservative analysis would be forced to assume non-aggregate always, we instead opt for a more pragmatic approach, with careful speculative-but-safe non-conservativeness, combined with the ability to switch such decisions off (Section 7.3.2).

7.3.3.5 Type Rounding

Next, we begin the process of connecting the extremely detailed structural types to more human-readable nominal types. We call this the type *rounding* phase, to evoke the similarity to rounding a real number to an integer. For the types, this phase rounds *up* structural types to their nearest (union of) primitive types. For example, if a type has been seen to support integer multiplication, it is relatively natural to assume it would also support division, since there is no C primitive type that supports multiplication but not division.

Type rounding can round types up to any collection of primitive types; however, since C is the lingua franca of decompiler output, we build in support for C; our implementation is agnostic to this choice, and other sets of primitive types could easily be supported. Indeed, one primitive that we include by default that does not directly exist in C is `code`, to allow representing pointers to executable memory (such as function pointers, or the return address slot on the stack) as `code*`, rather than `void*`. Note that despite rounding up to C-like types, the output of this phase is still structural types.

Clearly, this phase necessarily makes opportunistic non-conservative inferences with respect to the claims of observable behaviors that structural types capture. Nonetheless, we would like the smallest number of non-conservative inferences possible. More precisely, we define type rounding as an optimization problem to find the smallest subset of primitives that, when unioned, form a supertype of the input type. Unfortunately, this problem of type rounding is at least NP-Hard. We can show this by a reduction from Set Cover [57], an NP-Hard problem to find the smallest collection of sets that cover all elements in the union of those sets. Specifically, if we transform each set into a primitive type, with a structural type representation that has the relevant elements represented as observable behaviors, then finding the Set Cover is equivalent to performing a type-rounding of the structural type that contains all behaviors, and using the resultant union type to recover the relevant primitives, and thus sets. This inherent theoretical complexity of type-rounding implies that, in the general case, this phase is difficult to perform.

Nonetheless, for the purposes of decompiling to C-like types, we observe that our greedy approximation algorithm is both performant and produces sufficiently good results. Our algorithm for type rounding starts by representing the types using vectors and matrices, reducing the problem of rounding to that of finding the smallest x such that $A \times x \geq b$, where b is the vector of indicator variables for each component of a structural type (e.g.,

$\texttt{Add}_{32}$, $\texttt{Add}_{64}$, ..., would have separate rows), A is the matrix consisting of similar vectors for the allowed primitive types, and x selects which primitives are to be `union`ed. Starting x at all 1s (representing a union of all primitives), we repeatedly flip the most *expensive*-but-*unnecessary* primitive to 0, until we cannot anymore, returning the union of all remaining primitives. A primitive is considered unnecessary iff its removal from the union does not prevent the union from being a supertype (i.e., flipping its element in x to 0 does not negate the inequality $A \times x \geq b$); cost is computed as the number of enabled indicator variables in A for that type.

7.3.3.6 Nominal-Type Reconstruction

This phase converts the large and complex structural types to human-readable C-like types. This both identifies and provides names to all `struct`s and `union`s. A recursive walk through the graph of all structural types helps identify all C primitives, `struct`s, `union`s, pointers, and arrays. An interesting case that requires careful management during the graph walk is that of `struct`s, since they can be easily confused with their 0^{th} offset. We handle this by maintaining two names for each type, which are identical for all types other than `struct`s, where one refers to the 0^{th} field and the other refers to the `struct` itself. This graph walk approach works well even for recursive types, including `struct`-of-`struct`s, with the minor caveat that generally speaking, a `struct` that contains a `struct` as its 0^{th} field is indistinguishable from the flattened `struct`; thus the flattened version is picked during reconstruction. However, all non-zero offset fields *can* be distinguished and are handled as expected.

7.3.3.7 Output

TRex can output types from any intermediate phase, although the most useful phases are probably after Aggregate Analysis (for further downstream analyses that want the most precise types) or after Nominal-Type Reconstruction (for human analysts, who want C-like types). When outputting types, a user can also pick a subset of variables to output types for, possibly chosen through a different analysis.

7.4 Implementation

TRex is implemented in Rust, consists of approximately 9,600 source lines of code, and took approximately 2 person-years of effort to build. It is designed to be agnostic to the choice of binary analysis framework to plug into; for this paper, we use Ghidra. This involves a tiny script in the binary analysis framework (∼100 SLoC of Java for Ghidra) that outputs lifted code (P-Code for Ghidra) into a file that can then be ingested by TRex's lifter (∼900 SLoC of boilerplate Rust) into its own internal intermediate representation. Following this, the analysis proceeds in a series of phases (Section 7.3.3) that work on a graph representation of structural types, finally outputting either human-readable C-like types for reverse engineers to use, or machine-readable representation for further downstream analysts to use. We describe important components in more detail below.

7.4.1 Intermediate Representation in TRex

Choosing an intermediate representation (IR) for code is critical for expressivity and ease of building any static analysis tool. For TRex, we design a custom IR that is inspired by, but distinct from, Ghidra's P-Code IR. We pick this custom IR since P-Code has idiosyncrasies specific to Ghidra, and we have somewhat different goals that are better served by a custom IR. For example, Ghidra conflates various kinds of `call`s and `branch` instructions into a fairly small set of instructions (some with implicit overloaded semantics), which we break up into separate instructions. Also, Ghidra mixes constants, addresses, and registers into a single kind of type called a `varnode` (distinguished by magic constants picking an address space for each); since we implement TRex in Rust, we use algebraic types to handle these more cleanly. Also, Ghidra has (what we believe to be) shortcomings in its lifting that we fix up during the lifting to our IR, such as using an explicit `NOP` instruction in the IR (which Ghidra does not have), which is useful for maintaining alignment between the real machine instruction-pointer (e.g., `rip`) and the IR-internal program counter, rather than relying upon implicit fallthroughs at a jump target if there is a lack of instructions. We also explicitly support (a small number of) vector instructions, which Ghidra instead handles as architecture-specific magic constants in its catch-all `CALLOTHER` instruction. Additionally, we support an explicit `HAVOC` instruction that denotes a conservatively-under-specified clobbering of its output (used, for example, when lifting rare vector operations for which we have not yet found the need to have more precise semantics). Finally, having our own custom IR allows us to be Ghidra-agnostic, such that lifters from other frameworks' IRs could potentially be written to our IR.

The in-memory representation of our IR uses a Rust vector (`Vec`) of the (lifted) instructions in the program and properties about them such as maps between IR addresses and machine addresses (since a single machine instruction might expand to multiple IR instructions), and maps to maintain basic-block and function information. A special `FunctionStart` and `FunctionEnd` instruction denotes the *singular* start and end of a function, rather than allowing multiple entries and exits from functions.

Our static single-static assignment (SSA) IR is implemented as a view on the core IR that assigns SSA variable names to it. Unlike textbook SSA, we maintain ϕ-nodes in a separate list, thereby allowing a convenient map between instructions in both forms.

7.4.2 Representing Types in TRex

Each structural type is a precise representation of observable behaviors for a particular variable. The structural type consists of information about size, colocation, dereferencing, and operations observed upon it, represented as sets, booleans, and indices. Figure 7.2 shows an example structural type.

The set of operations within a type precisely capture observable behavior; each is roughly analogous to a machine operation—for example, each of the following is an entirely distinct operation that could exist in a structural type: $\texttt{Add}_{32}$, $\texttt{Add}_{64}$, $\texttt{Sub}_{32}$, $\texttt{Copy}_{32}$. Note how the existence of 32-bit addition ($\texttt{Add}_{32}$) does not (immediately) indicate that the type supports 32-bit subtraction ($\texttt{Sub}_{32}$). We also note that while it might seem superfluous to maintain size information with the operations (instead of maintaining size information about the whole type), this is crucial to handle situations such as the `union(u32,f64)` being distinct from `union(f64,u32)`.

We represent dereferencing via a points-to relation on the type by maintaining an `Index` into the graph of all structural types (Section 7.4.3). Aggregate types (i.e., `struct`s) maintain colocation information as a map from non-zero offsets to `Index`es into the graph. The offsets are non-zero, since the operations for the first field of a struct are the same as the struct itself, modulo colocation; thus offset 0 always refers to the “current” type. Finally, we do not represent `struct` padding in the structural type explicitly; this allows us to distinguish between a `struct` padding byte (unused) and an `undefined1` (used single byte, but nothing more known; cannot be a `char`, `short short`, etc.).

7.4.3 Joinable Containers

Since the design of TRex requires directly dealing with possibly-recursive types, we must represent the structural (and later, C-like) types in some graph structure. Rust's type system is famously known to be graph unfriendly (although good graph libraries exist).

Additionally, we require the ability to merge separate, partially-specified structural types together when we recognize them as equal (for example, two different instructions might help us learn that some location X's type supports 64-bit integer addition, and Y's type supports dereferencing, and then a third operation might show that X and Y have the same type; in this case, we need to merge the types of X and Y). Furthermore, while in the process of merging two types, we might further discover other types that require merging (for example, p_X and p_Y might point to X and Y; once we find that p_X and p_Y must have the same type, we must also merge the types of X and Y).

Hence, we not only need a graph structure, it must also support this (potentially recursive, and deep) join/merge operation. To solve this, we implemented a custom graph data structure that supports safe access to graph members by storing members into an arena with strongly-typed `Index`es. This structure is parametric in its objects, only requiring that the objects support *some* join operation.

Inspired by the Disjoint-Set (aka Union-Find) data structure, we maintain a canonical internal index for each external `Index`, so that merge operations do not require a global scan to update all indices.

During any join operation, we maintain a queue to schedule further join operations during the process of each of the joins, repeatedly processing each join until it terminates. We guarantee termination by explicitly checking for repeats (which can happen when joining two recursive structural types), and noticing that modulo loops, the join operation must strictly decrease the total number of distinct structural types available.

The arena itself behaves (as expected) as an arena allocator, handing out new strongly-typed `Index`es upon allocation request; freeing is performed en masse, deallocating the entire arena at once. However, this is not the only possible deallocation operation, since the arena also supports garbage-collection and compaction, which can be explicitly invoked by providing it with a series of roots to keep alive. By maintaining a globally-unique arena identifier, indices into one arena are checked to not be accidentally used with another arena, and sentinels confirm that no use-after-free can occur with these `Index`es after invoking a garbage-collection pass.

```c
struct Node {
  int data;
  struct Node* next;
};

int getlast(struct Node* n) {
  struct Node* nxt = n->next;
  while(nxt != 0) {
    n = nxt;
    nxt = n->next;
  }
  return n->data;
}
```

Figure 7.4: Source Code for the Singly-Linked List Example

7.5 Evaluation

We answer the following research questions, through both a qualitative and a quantitative evaluation of TRex.

RQ1. On what aspects of inferred types does TRex improve upon the state of the art?
RQ2. In benchmarks commonly used in prior work, how successful is TRex at reconstructing types?

7.5.1 Qualitative Evaluation

To help answer **RQ 1**, we use a simple example of a singly-linked list. Using the Decompiler Explorer [142], we compare TRex against the outputs of popular decompilers, namely (i) Binary Ninja 3.5.4526 [141], (ii) Ghidra 11.0 [106], and (iii) Hex-Rays (IDA Pro) 8.3.0.230608 [126]. We compile the C code in Figure 7.4 to an x86-64 ELF object file, using GCC 11.4.0, strip away debugging information and pass it to all of the tools, including TRex.

Focusing on the type of the parameter, all three pre-existing decompilers successfully recognize that it is a pointer but fail to detect the `struct`. Instead, they identify it as a primitive type—`int32_t*` for Binary Ninja, `undefined4*` for Ghidra, and `unsigned int*` for Hex-Rays. A reverse-engineer *can* request further automated analysis for particular variables in Ghidra and Hex-Rays if they manually decide it could be a `struct`. For example, if they invoke Ghidra's "Auto Fill in Structure", then it updates the type to be a `struct` with 6 fields: an `undefined4`, four `undefined`s (indicating padding, distinct from `undefined1`), and an `undefined4*`. Notice that despite explicitly requesting structure analysis, Ghidra is unable to detect the recursive nature of the type.

```
// TRex's structural type for n : t31
t31 {
  UPPER_BOUND_SIZE  8
  COPY_SIZES  {8}
  POINTER_TO  t33
  INTEGER_OPS   {Add_8, Sub_8, ULt_8, SBorrow_8}
}
t33 {
  COPY_SIZES  {4}
  INTEGER_OPS   {ZeroExtendSrc_4}
  COLOCATED_STRUCT_FIELDS   8   t31
}
```

```
// TRex's C type for n : t1*
struct t1 {
  uint32_t field_0;
  t1* field_8;
};
```

Figure 7.5: TRex output on Singly-Linked List Example

In contrast, TRex produces the output in Figure 7.5. The structural type for the parameter is `t31`. For human consumption, TRex rounds this to the C type `t1*`, which successfully captures the correct `struct` shape, including the recursion and padding (`field_N` denotes the field at byte offset `N`). The only part of the type where TRex does not perfectly match the source is the first field, where the signedness differs. However, notice that the code itself does not interact with this value, and thus either signed or unsigned would be consistent with the produced code. TRex picks `uint32_t` since that is the most precise primitive available that is consistent with the only observed operation on the variable from the disassembly—a zero-extension operation.

The specific order of the `Node` fields in Figure 7.4 makes this example challenging, since recursion and colocation influence one another. Nonetheless, when we try the easier ordering (with recursive pointer being at offset 0), none of the three existing decompilers do any better at recognizing either the `struct` or the recursion (producing `int64_t*`, `undefined8*`, and `_QWORD**`, for Binary Ninja, Ghidra, and Hex-Rays, respectively). TRex produces the expected type.

To understand the impact of colocation and aggregate analysis in TRex, we can disable those particular phases—this produces the parameter type `uint32_t*` for `n` in Figure 7.4, performing similar to the other decompilers. However, in the easier case where recursion and

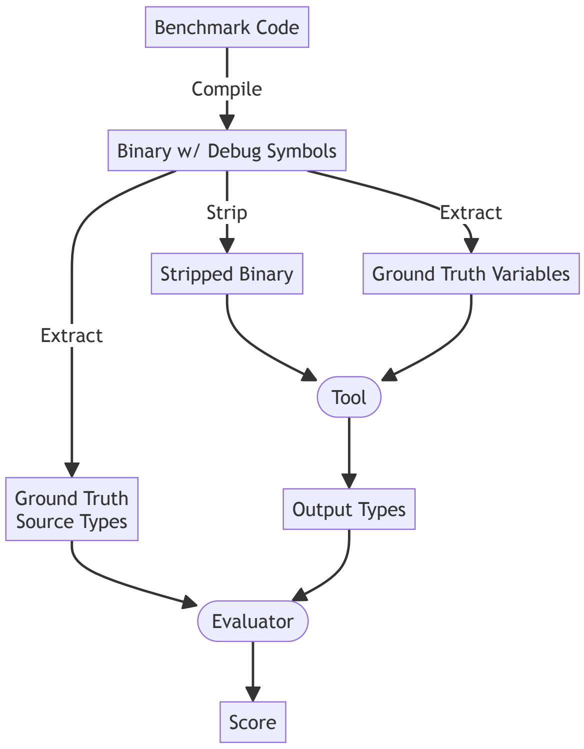


Figure 7.6: Quantitative Evaluation Steps

colocation are independent, TRex successfully detects the recursion, prints a warning about the infinitely dereferenceable pointer, and outputs the closest C-like type that captures this—`void*`.

7.5.2 Quantitative Evaluation

To explore how successful TRex is on benchmarks commonly used by prior work and help answer **RQ 2**, we perform a quantitative evaluation against 125 binaries from prior work. As discussed in Section 7.1, prior work rarely makes reproducible benchmarks available for comparison. Thus, we develop a set of reproducible benchmarks that can be used to compare against state-of-the-practice and future tools. Amongst practically available tools, we compare against Ghidra, since both IDA and Binary Ninja require commercial licenses to perform batch analysis, and our practical anecdotal evidence have neither doing particularly better than Ghidra at type inference. We additionally compare against a trivial baseline (that outputs an `undefinedN` of the correct size on each variable) to provide context on the quality of existing state-of-the-art tools.

The benchmarks themselves consist of GNU Coreutils 9.3 and SPEC CPU® 2006 [135] (hereafter referred to as Coreutils and Spec respectively). We compile each benchmark program using reproducible scripts, with debug symbols, to obtain ground truth variables and types. Then, by stripping the debug symbols, we obtain a stripped executable. The tools only have access to this stripped executable and the ground-truth variable information (location and size of each variable). They are expected to output type information, which is then scored against the ground truth types. Figure 7.6 shows these steps graphically. We use all 108 binaries from Coreutils. For Spec, of the 29, we skip all ten Fortran binaries because their DWARF debug symbols appear to be invalid, producing either empty or nonsensical sets of ground truth variables and types. In addition, we are also forced to skip two of the C++ binaries because Ghidra could not analyze them correctly.

As discussed in Section 7.2.2, it is impossible to define any single ground truth for binary type inference. Any quantitative evaluation is thus necessarily an approximation of the quality of the output types. Hence, relying on our own experience as reverse engineers (and anecdotes from others), we define a best-effort prioritized ordering of frustrations with incorrectly inferred aspects of types. Using this ordering, we have created a flowchart that defines our scoring function for measuring the agreement between the computed and the original type (Figure 7.7). For example, we find it incredibly frustrating if a pointer is not detected as such (or vice versa), since it can lead to incorrect reasoning about memory, and thus getting this wrong leads to a large score penalty. Others might have a different set of priorities, so we implement a small DSL in our evaluation tool to let reverse engineers test alternative priorities for scoring.

A purely conservative analysis would not produce useful results, as discussed in Section 7.3.2, and we find this to be the case in practice. Thus in the results below, we elide TRex's fully conservative mode. Instead, we use TRex in its default configuration—primarily conservative with common opportunistic pattern inferences enabled.

Figure 7.8a shows the mean score achieved by each tool across all variables on each of the 108 Coreutils executables (compiled on x86-64), sorted by the scores for TRex. We see that on all but three executables (namely, `od`, `sha384sum`, and `sha512sum`), TRex achieves a better average score than Ghidra, which itself achieves a better score than the trivial baseline. We note that TRex is able to outperform Ghidra *despite* TRex not yet implementing interprocedural type propagation, which Ghidra uses (in addition to external functions it knows, such as those in libc) to obtain better types. Additionally, on the aforementioned three executables, Ghidra only marginally outperforms TRex—some preliminary analysis of the root causes point towards conservatively underspecified

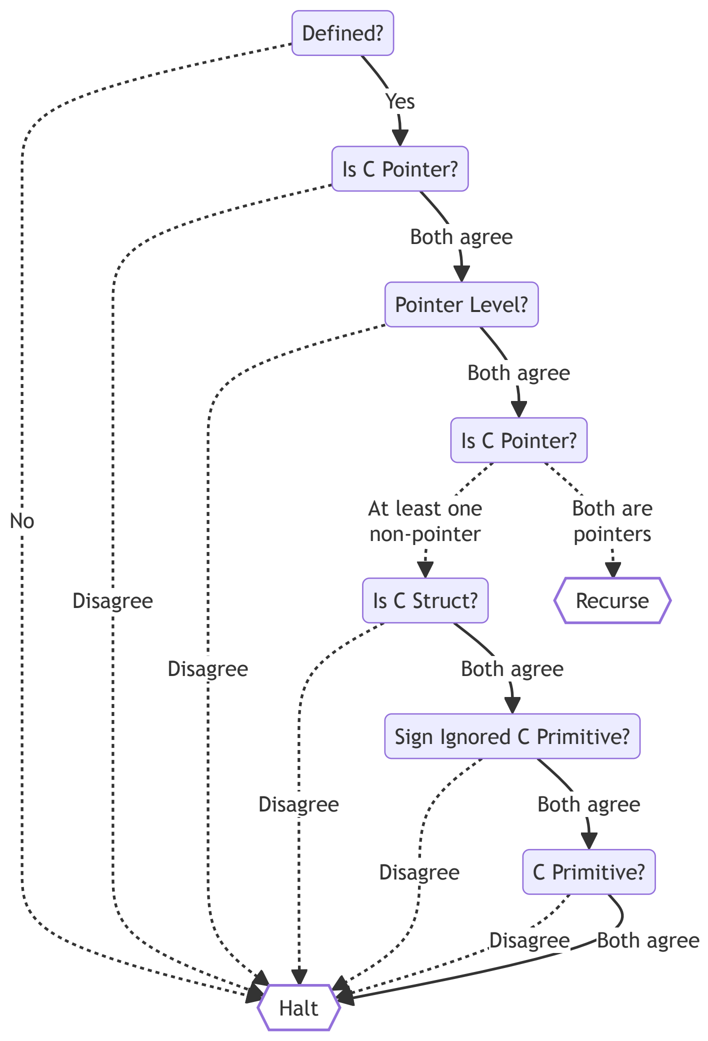
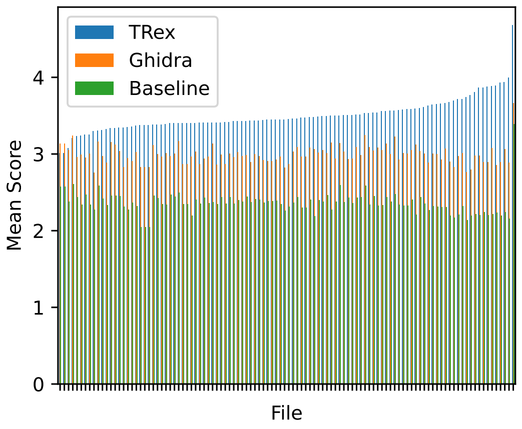


Figure 7.7: Scoring Metric for Evaluating Type Reconstruction. *Traversing solid lines increments the score by 1, while traversing dotted lines does not update the score.*

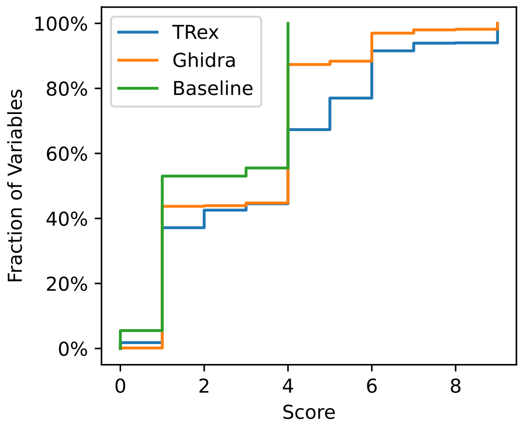
vector instructions (whose semantics one could improve with straightforward-if-non-trivial engineering effort).

To better understand the scores across the entirety of the dataset, in Figure 7.8b we also plot the cumulative distribution function (CDF) of scores across all 104,953 variables. We notice that TRex moves into higher scores sooner than Ghidra, which moves into higher scores sooner than the baseline. The mean score achieved by TRex, Ghidra, and the baseline are 3.499, 2.993, and 2.36 respectively. Note that the graph looks like a step function since the scoring function of any particular variable evaluates to an integer.

We plot similar graphs for SPEC, which covers a larger variety of executables (although



(a) Mean scores, sorted by score on TRex. Higher is better.

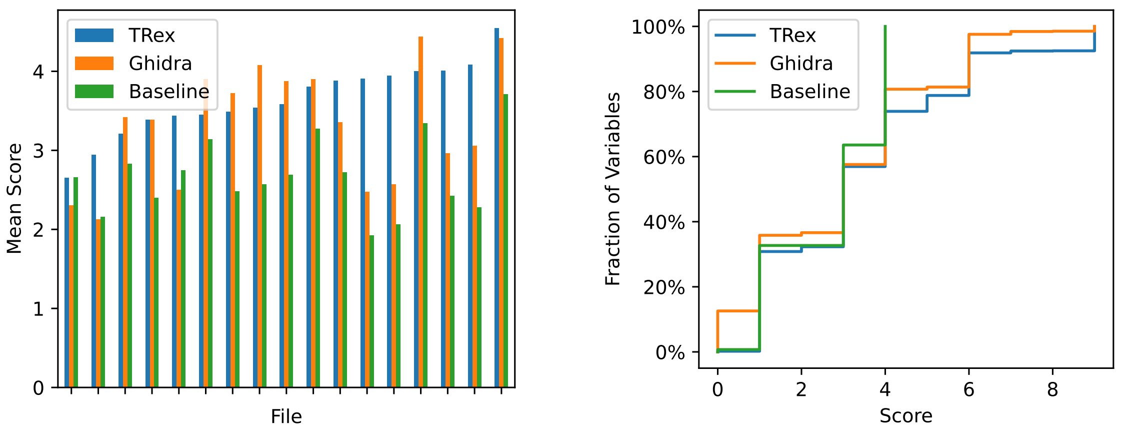


(b) CDF of scores. Lower is better.

Figure 7.8: Evaluating TRex on COREUTILS

it is compiled on x86 32-bit, which is not a case that we have optimized TRex for just yet). We notice in Figure 7.9a that the difference between the tools is not as stark as on COREUTILS. Nonetheless, TRex outperforms Ghidra on 10 of the 17 executables. The mean score achieved by the TRex, Ghidra, and the baseline are 3.638, 3.321, and 2.669 respectively. Looking at the CDF of scores across all 63,600 variables in Figure 7.9b, we see a similar behavior as on COREUTILS, in that TRex moves into higher scores sooner than Ghidra, which moves into higher scores sooner than the baseline.

Finally, we end with a brief discussion on performance. For our use cases, exact performance numbers are not particularly relevant, since we can afford to let an analysis run overnight or longer if it produces better output than something that finishes instantly. Nonetheless, we note that TRex takes only a mean of 56.32s (min=1.60s, max=1385.15s) across the entirety of COREUTILS and SPEC. We contrast this to Ghidra, whose analysis takes a mean time of 77.16s (min=7.91s, max=1606.45s). Some caveats to these numbers include the following: (i) there is no easy way to separate type-reconstruction time from total decompilation time in Ghidra, thus the numbers for Ghidra are over-estimates, and (ii) TRex spends the majority ($\sim 85\%$) of its time on SSA construction (which we have not optimized and have implemented using a naïve algorithm; we believe a faster algorithm [19] would improve performance).



(a) Mean scores, sorted by score on TRex. Higher is better.

(b) CDF of scores. Lower is better.

Figure 7.9: Evaluating TRex on Spec

7.6 Related Work

The problem of computing high-level types from binaries has a sizable literature. Caballero and Lin [15] conducted an authoritative survey of 38 prior papers in this area up to 2015. They concluded that existing tools can take widely different approaches and can vary greatly in the set of types that can be computed. Due to space, below we only discuss several more recent and/or highly relevant projects, and we refer interested readers to the survey for a more comprehensive overview.

On the deductive side, TIE (2011) by Lee et al. [71] was an early tool that popularized the use of constraint solving, which is also used by TRex. At a high level, TIE collects usage constraints about identified variables from a binary (static TIE) or a trace (dynamic TIE) and outputs a solution that satisfies all collected constraints. Compared to TIE, TRex is designed to support a more expressive type system, notably including recursive types due to their prevalence in real-world programs.

Retypd (2016) by Noonan et al. [105] is a recent prior work that also uses the constraint-solving approach. As far as we know, the type system supported by Retypd is the most expressive among the tools in the literature; indeed, it even exceeds that of TRex in one aspect. In particular, Retypd supports computing polymorphic types, e.g., $\forall\tau :$ `size_t` $\rightarrow \tau*$. For C programs, expressing such types usually involves the use of `void*` and casting, e.g., when calling `malloc`; for C++ programs, they may further express such types using the C++ template system. Adding support for polymorphic types to TRex is a natural future direction in our research.

OSPREY (2021) by Zhang et al. [160] is the latest tool among those that are based on constraint solving. Compared to prior tools, OSPREY introduced an interesting and powerful extension, namely the use of probabilistic constraints. This enables OSPREY to potentially cope with the inherent uncertainty in variables and types in binaries due to information lost during compilation. Whereas OSPREY uses probabilities to model this uncertainty and emits types that are deemed most likely in the final phase, TRex uses structural types to preserve complete information about program behavior until the final type rounding and nominal type reconstruction phases, which proceed deterministically. OSPREY also still aims for type recovery, rather than working from TRex's perspective of Type Reconstruction. Source code for OSPREY is not publicly available, so we have not been able to compare its performance with TRex's.

On the learning side, there has been a surge of new proposals in the area, leveraging some of the latest advances in machine learning and in hardware capabilities. StateFormer (2021) by Pei et al. [113] demonstrates the power of a clever use of neural networks for this domain. It starts with a fully self-supervised pre-training step where the model is taught to statistically approximate the operational semantics of instructions in both forward and backward directions using a very small number of execution states. Then, it fine-tunes the model to use the learned operational semantics to infer types. Compared to TRex, StateFormer is able to output only a fixed set of 35 types. However, we recognize a key strength of StateFormer's approach, which is its explicit learning of instruction semantics. This can be extremely valuable when new instructions are introduced by CPU vendors or when dealing with binaries for a new ISA.

Another recent learning-based approach in the area is DIRTY (2022) by Chen et al. [17]. DIRTY features a transformer neural network that was trained on a large corpus of open-source C projects mined from GitHub and is capable of recommending variable names and types as a post-processor of decompilation outputs. In total, DIRTY supports 48,888 types that were encountered in its corpus. While the set of types computable by DIRTY is much larger than that of StateFormer, it is still limited to previously-seen types when compared to TRex. Here, we note that Chen et al. suggest the use of Byte Pair Encoding [128] to extend DIRTY to support computing previously-unseen `struct` types as future work.

7.7 Concluding Remarks

Lack of high-quality source-level types has plagued decompiled code for decades despite advances in the art and science of decompilation. In this chapter, we've presented TRex, a new tool that performs automated deductive type inference for binaries using a new perspective, *Type Reconstruction*. This perspective accounts for the inherent impossibility of recovering lost source types, and guides us to perform analyses using *structural types* that capture the behavior of the program. Since human reverse engineers are more familiar with C-like output from decompilers, we then round these precise-but-verbose structural types to C-like nominal types. Overall, TRex shows a noticeable improvement in the quality of output types compared with Ghidra, an actively developed state-of-the-art decompiler used by practitioners. We also document insights derived while building TRex, such as showing that type rounding is an NP-Hard problem, but our greedy algorithm works reasonably well in practice.

More broadly, we hypothesize that the field of decompilation's focus on the impossible goal of *recovering* lost source code has contributed to the increasing complexity of tools and techniques in the academic literature. To the extent that this complexity "overfits" to a particular compilation technique or even a particular compiler, this may also explain why these techniques are not used in practical decompilers. In contrast, we show that by shifting our focus away from the impossible and instead towards what reverse engineers desire—output that captures behavior, we can design and implement simpler and more elegant decompilation tools that help reverse engineers with their actual requirements. Thus, this approach shows promise in reducing the manual effort needed by both tool developers and tool users.

Our work does lead to some natural questions for future work. Inter-procedural type propagation appears to be what powers much of Ghidra's quality of output; TRex does not implement this straightforward-but-nontrivial-effort analysis, but it would be interesting to understand how much this factors into quality of output in a Type Reconstruction setting in contrast to the older Type Recovery setting. Furthermore, TRex focuses on high-quality automated *initial* types; and it does not currently take types from users as input. It would be interesting to explore Type Reconstruction with additional input types. Similarly, TRex only takes (lifted) disassembly, but one might conceive of a Type Reconstruction analysis that instead takes pre-existing decompilation output. Along a different direction, TRex could be augmented to propose better names for the rounded C-like aggregate types, rather than the current auto-generated identifiers, possibly using recent advances in machine learning; indeed, this might be a fertile ground for learning-based Type Prediction approaches, with its strength of human readability on in-distribution binaries, to connect with Type Reconstruction, with its guarantees of reasonable output for all binaries. Finally, we speculate that our shift in perspective from Type Recovery to Type Reconstruction might more broadly be applicable to other components of the decompilation pipeline, providing more avenues for exploration.

In summary, in this chapter with the new formalism of Type Reconstruction, along with a tool implementing it, we show that automating better software comprehension for legacy code is practically feasible, without apologizing for output expressivity, or escalating complexity.

Chapter 8

Conclusion

Verily, this vichyssoise of verbiage veers most verbose, so let me simply add that . . .

V for Vendetta (2006)

Inspired by the seemingly constant tug-of-war between security and other important software qualities, in this thesis, we have explored the question of whether it is even feasible for us to have our cake and secure it too. We answer this question in the affirmative through multiple case studies, to demonstrate that not only are previously-accepted apologies possible to overcome, but also that strong security can help achieve other useful software objectives.

Using formalism and principled approaches, we first see that high-performance cryptographic software is not only verifiable with beautiful proofs, but also that the strong safety net provided by the verification allows us to safely and easily optimize well beyond the fastest unverified implementations. These elegant proofs and safe optimization use the technique of verified code transformations, which helps automatically reason about semantically equivalent code in a convenient fashion.

Our next case study looks more broadly at fearlessly running arbitrary code in our own process, through the use of WebAssembly as a narrow waist. We demonstrate two distinct techniques that achieve provably-safe sandboxing with competitive performance; the first (vWasm) uses traditional formal verification techniques, while the second (rWasm) uses a non-traditional approach of embedding to imbue code with safety. The latter additionally is uncompromising in portability and runtime extensibility, thereby providing a fertile ground for exploring additional properties in a high-assurance compiler that produces high-performance sandboxed code.

Indeed, our next case study does just that—we explore extending rWasm with an extension to WebAssembly (namely, MSWasm) that allows us to provide additional guarantees on top of sandboxing. In particular, we show that intra-module memory safety is achievable with agility in the enforcement mechanisms, such that improvements to safety enforcement mechanisms can be decoupled from the module it is applied to. This allows for environment and context-driven decisions to be made by engineers protecting users while (say) awaiting a patch for a vulnerability to be released.

We then explore two different approaches to protecting against untrustworthy serialized data. Our first approach (VMarshal) focuses on intrinsic data formats, providing a convenient user experience for users of verified serializers and parsers who do not wish to deal with the on-wire format, and instead want to focus on high-level data. The second approach (Vest) focuses on extrinsic data formats, providing a convenient user experience for users who wish to use verified serializers and parsers for externally-chosen data formats; it does so through proof-producing compilation of a DSL that helps map both low-level serialization and high-level data models in the same language.

Finally, our last case study explores a specific facet of securing source-unavailable legacy code, where prior approaches have not been deployed in practice, presumably due to escalating complexity. Our new formalism allows for elegantly inferring behavior-capturing types from binaries, reducing the manual effort needed for reverse engineering binary code, without compromising on elegance, output expressiveness, or quality.

In summary, through this dissertation, we have explored multiple case studies that each repeatedly demonstrate our core thesis—unapologetic security *is* achievable. We demonstrate this through the principled application of verification and formalism, as applicable to each case study. Not only do we eliminate prior apologies for security, but in certain cases, we also show that focusing on security unlocks other properties, such as performance or extensibility, which could not have been easily achieved otherwise.

In terms of future directions, beyond specific follow-on work to the above case studies described in their relevant chapters, it would be interesting to explore how we might combine ideas from each of them. For example, decompilation gives us an insight into the usage of memory—we may be able to use this to help carve up Wasm's linear memory in order to further improve performance. Along a different line, intra-process sandboxing helps naturally carve up process memory, yet the sandboxes need to communicate via serialized data; perhaps we can use some lightweight proofs (or a type-system) to provide more expressive communication between sandboxes safely, improving performance. Stepping back a bit and thinking more broadly, the software world is a big growing collection of incompatibilities caused by modern advancements combining with legacy software, tools, and techniques—exploring techniques that can deal with all of this as a whole is thus going to become even more pressing as more software is written. The case-studies in this dissertation are motivated by this, and thus we hope that they can provide jumping-off points towards more powerful techniques for dealing with the growing mass of both legacy and fresh software.

A closing thought: when building secure systems, we must deal with situations where the slightest mistake could be disastrous. This is exactly where rigor and formalism shines, allowing us to reason about software precisely. We have learnt that exploring beyond traditional techniques allows us to get the guarantees without the tedium or other apologies. It would be interesting to explore how far we can push towards unapologetic security. Perhaps, one day, we might be able to have provable security be the default choice; perhaps even the only reasonable choice. Well, one can dream.

Bibliography

[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 Fabian Monrose, editor, *Proceedings of the 18th USENIX Security Symposium*, pages 51–66. USENIX Association, 2009. 72, 73, 78, 81

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. 17, 33

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. *CoRR*, abs/1904.04606, 2019. 17, 33

[4] Announcing Lucet: Fastly's native WebAssembly compiler and runtime. `https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime`, March 2019. 38, 41, 57, 67

[5] Andrew W. Appel. Verified software toolchain. In *ESOP: 20th European Symposium on Programming*, 2011. 32

[6] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, April 2015. 32

[7] Arm. Armv8.5-a memory tagging extension. *White Paper*, 2019. 72, 81

[8] Arm Morello Program. `https://www.arm.com/architecture/cpu/morello`, 2022. 79

[9] 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. 7, 97, 99, 100

[10] 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. 86

[11] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*, August 2017. 17, 19, 22, 32

[12] 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. 15, 51

[13] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-safe multilingual software sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*, August 2022. 36, 77, 82

[14] Jay Bosamiya, Maverick Woo, and Bryan Parno. TRex: Practical type reconstruction for binary code. In Submission. 104

[15] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys*, 48(4):65:1–65:35, May 2016. 107, 108, 128

[16] Ligeng Chen, Zhongling He, and Bing Mao. CATI: Context-assisted type inference from stripped binaries. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98, 2020. 104, 107

[17] 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. 104, 107, 129

[18] Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`, 2020. 58

[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35. ACM Press, January 1989. 127

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

72

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 46, 90

[22] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. 11, 89

[23] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proceedings of the ACM on Programming Languages*, 3(ICFP), jul 2019. 86, 101

[24] 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. 86

[25] The pest developers. pest. The Elegant Parser. `https://pest.rs/`. 88, 99

[26] The Serde developers. Serde. `https://serde.rs/`. 100, 101

[27] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2008. ACM. 81

[28] 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. 5, 68, 71, 72, 73, 75

[29] Docker. `https://www.docker.com/`, 2021. 66

[30] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019. 17, 33

[31] 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. 36, 38, 57

[32] Ethereum WebAssembly (ewasm). `https://ewasm.readthedocs.io/en/mkdocs/`, 2021. 36

[33] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967. 10

[34] 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. 99

[35] Python Software Foundation. marshal - internal Python object serialization. `https://docs.python.org/3/library/marshal.html`, 2024. 91, 100

[36] Python Software Foundation. pickle - Python object serialization. `https://docs.python.org/3/library/pickle.html`, 2024. 91, 100

[37] 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. 16, 17, 18, 19, 29

[38] 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. 36, 68, 82

[39] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. Owl: Compositional verification of security protocols via an information-flow type system. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2023. 85

[40] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`, 2020. 67

[41] 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. 104

[42] 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. 73

[43] S. Gueron and V. Krasnov. The fragility of AES-GCM authentication algorithm. In *Proceedings of the Conference on Information Technology: New Generations*, April 2014. 28, 32

[44] Shay Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf`, September 2012. 27

[45] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. 4, 36, 58, 59, 60

[46] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, October 1988. 44

[47] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. 107

[48] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. 10

[49] How many x86 instructions are there? `https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/`, 2016. 42

[50] Hypervisor-Protected Code Integrity (HVCI). `https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/device-guard-and-credential-guard`. 66

[51] Intel®. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, April 2021. 42

[52] ISO. ISO C standard 1999. `https://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf`, 1999. Section 6.7.2.1, Item 16, Page 103. 116

[53] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association. 68, 79, 82

[54] 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. 36, 38, 67

[55] 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. 86, 101

[56] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL), December 2017. 52

[57] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. 117

[58] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1445–1461. USENIX Association, August 2020. 44

[59] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 361–372. IEEE Press, 2014. 44

[60] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), 2014. 33

[61] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. 11

[62] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018. 81

[63] 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. 36, 37, 67

[64] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2014. 44, 45

[65] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2013. 81

[66] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 165–182, New York, NY, USA, 1991. Association for Computing Machinery. 12

[67] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992. 113

[68] 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. 11, 85, 87, 90

[69] 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. 11, 87, 90

[70] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. 81

[71] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2011. 128

[72] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again:

Binary security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020. 68, 72, 75, 82

[73] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world, 2001. 88

[74] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. 11, 89

[75] 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. 73

[76] 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. 11, 24, 32, 37, 44, 45, 67

[77] 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. 72, 73, 81

[78] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the Network and Distributed System Security Symposium*, pages 1–18. The Internet Society, 2010. 104

[79] Linux Containers. `https://linuxcontainers.org/`, 2021. 66

[80] Maddie Stone and James Sadowski. A review of zero-day in-the-wild exploits in 2023. Google's Threat Analysis Group and Mandiant. `https://blog.google/technology/safety-security/a-review-of-zero-day-in-the-wild-exploits-in-2023/`. Archived: `https://archive.is/JwXhP`, March 2024. 35

[81] 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. 104, 107

[82] Henry Massalin. Superoptimizer – a look at the smallest program. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1987. 33

[83] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery. 52, 90

[84] Matthew Maurer. *Holmes: Binary Analysis Integration Through Datalog.* PhD thesis, Carnegie Mellon University, Oct 2018. 108

[85] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 62–72. ACM, 2012. 101

[86] 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. 36

[87] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7), July 1965. 23

[88] Tyler McMullen. Lucet: A compiler and runtime for high-concurrency low-latency sandboxing. Talk presented at the Principles of Secure Compilation (PriSC) workshop. `https://youtu.be/WddPAOU6v2A?t=2309s`, 2020. 57

[89] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019. 76

[90] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 1–15, New York, NY, USA, 2016. Association for Computing Machinery. 76

[91] 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. 5, 71, 73, 75, 76, 77, 78

[92] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, dec 1990. 2

[93] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. 107

[94] MITRE. Stubborn Weaknesses in the CWE Top 25. `https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html`, 2023. 86

[95] 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. In *ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pages 25–35, 1999. 41

[96] 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. 36, 38, 67

[97] Mozilla. Measurement dashboard. `https://archive.is/ku005`, January 2024. 3, 15, 16, 27

[98] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016. 23

[99] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, page 208–223, Berlin, Heidelberg, 1999. Springer-Verlag. 104, 115

[100] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM SIGPLAN Notices*, 44(6):245–258, June 2009. 72, 79

[101] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, page 31–40, New York, NY, USA, 2010. Association for Computing Machinery. 72, 73, 79

[102] 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. 36, 44, 54, 68

[103] 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)*. USENIX Association, 2020. 82

[104] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005. 72

[105] 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. 104, 108, 128

[106] National Security Agency (NSA). Ghidra. `https://www.nsa.gov/ghidra`. 7, 104, 122

[107] Department of Defense. Trusted computer system evaluation criteria. DOD 5200.28-STD, December 1985. 12

[108] Authors of VeriWasm. Private Correspondence. 38

[109] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2), jun 2018. 81

[110] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996. 2

[111] Matthew Parkinson, Kapil Vaswani, Dimitrios Vytiniotis, Manuel Costa, Pantazis Deligiannis, Aaron Blankstein, Dylan McDermott, and Jonathan Balkind. Project Snowflake: Non-blocking safe manual memory management in .NET. Technical Report MSR-TR-2017-32, Microsoft, July 2017. 81

[112] 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. 36, 68, 82

[113] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana.

StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 690–702. ACM, 8 2021. 129

[114] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, pages 151–166, 1998. 33

[115] Louis-Noel Pouchet. PolyBench-C: the Polyhedral Benchmark suite. `https://web.cs.ucla.edu/~pouchet/software/polybench/`. Accessed: January 2021, 2011. 58, 73, 79

[116] 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. 4, 11, 16, 18, 26, 28, 29

[117] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages*, 1(ICFP), aug 2017. 89

[118] Python 3. Glossary: Duck typing. `https://docs.python.org/3/glossary.html#term-duck-typing`. Archived: `https://archive.is/qvZGc`. 111

[119] 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. 6, 86, 99, 100

[120] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Layered indexed effects: Foundations and applications of effectful dependently typed programming. `https://www.fstar-lang.org/papers/layeredeffects/`, 2020. 46

[121] Frederic Recoules, Sebastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-

Laure Potet. Get Rid of Inline Assembly through Verification-Oriented Lifting. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 2019. 33

[122] The Rust Reference. Procedural macros. `https://doc.rust-lang.org/reference/procedural-macros.html`. 96

[123] John Renner, Sunjay Cauligi, and Deian Stefan. Constant-time WebAssembly. In *Principles of Secure Compilation (PriSC)*, January 2018. 68

[124] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. 11, 67

[125] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with Checked C. In *Principles of Security and Trust*. Springer, 2019. 76

[126] Hex-Rays SA. Hex-Rays Decompiler. `https://hex-rays.com/decompiler/`. 104, 122

[127] 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. 36

[128] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725. Association for Computational Linguistics, 2016. 129

[129] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association. 57

[130] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. Gwp-asan: Sampling-based detection of memory-safety bugs in production. *CoRR*, abs/2311.09394, 2023. 83

[131] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of ACM PLDI*, 2013. 33

[132] Robert W. Shirey. Internet Security Glossary, Version 2. RFC 4949, August 2007. 12

[133] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button

verification of file systems via crash refinement. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 1–16. USENIX Association, 2016. 11

[134] Pratap Singh, Joshua Gancher, Yi Cai, Jay Bosamiya, and Bryan Parno. Flaco: Automatically compiling security protocols to verified, high-assurance, performant implementations. In Submission. 85

[135] SPEC. SPEC CPU2006. `https://www.spec.org/cpu2006/`. 125

[136] 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. 11, 18, 37, 45, 89

[137] 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. 6, 86, 99, 101

[138] Coq Development Team. The Coq proof assistant. `https://coq.inria.fr/`. 11, 89

[139] The Rust programming language. `https://www.rust-lang.org/`, 2021. 52

[140] Guido van Rossum, Pablo Galindo, and Lysandros Nikolaou. PEP 617 - New PEG parser for CPython. `https://peps.python.org/pep-0617/`, 2020. 88

[141] Vector 35. Binary Ninja. `https://binary.ninja/`. 104, 122

[142] Vector 35. Decompiler explorer. `https://dogbolt.org/`. 122

[143] 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. 36, 39, 67

[144] 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. 86, 101

[145] WASI – The WebAssembly System Interface. `https://github.com/WebAssembly/WASI`, 2021. 36, 41, 57

[146] wasm2c. `https://github.com/WebAssembly/wabt`, 2021. 41, 53, 57

[147] wasm3. `https://github.com/wasm3/wasm3`. 41, 57

[148] Wasmer - The Universal WebAssembly Runtime. `https://wasmer.io/`, 2021. 41, 57

[149] Wasmtime: A small and efficient runtime for WebAssembly & WASI. `https://wasmtime.dev/`, 2021. 38, 41, 57

[150] 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. 72, 78

[151] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65, New York, NY, USA, 2018. Association for Computing Machinery. 44

[152] WAVM: WebAssembly virtual machine. `https://wavm.github.io/`, 2021. 41, 57

[153] WebAssembly Micro Runtime (WAMR). `https://github.com/bytecodealliance/wasm-micro-runtime`, 2021. 41, 57

[154] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, abs/1903.00982, 2019. 52

[155] David A. Wheeler. SLOCCount. Software distribution. `http://www.dwheeler.com/sloccount/`. 28

[156] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147. Association for Computing Machinery, 3 2020. 104

[157] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs, 2021. 52

[158] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pages 158–177. IEEE, 5 2016. 104

[159] 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 *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009. 36, 38, 40, 67

[160] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2021. 129