UNIVERSITY OF CALIFORNIA SAN DIEGO

Language Techniques for Automated Verifiction of Web Security

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

in

Computer Science

by

John Renner

2022

University of California San Diego

2022

DEDICATION

This dissertation is dedicated to my teachers without whom I would have neither the will nor the skill to have accomplished this,

To my mom, who didn't notice I learned TI-BASIC in the back of her Geometry class.

To Mr. Steuben, whose nigh-sadistic intro CS class got me hooked.

To Mr. Majeske, who taught me to be a "stuff-knower" and a troublemaker.

This dissertation is dedicated to my D&D group, whose weekly games filled these past years with creativity and friendship,

To David Coy, my once again dear friend who I'm lucky to have held onto.

To Caroline Coy, my fast friend whose chaotic energy is unmatched except by her cats.

To John Grischuk, whose name is the same as mine, and who formerly ruled Wesnoth.

To Mark Renner, my brother who housed me during some tough years.

To Jeff Renner, one of the most genuine souls I've ever met. He also plays a mean cowboy.

This dissertation is dedicated to my friends in San Diego who made this city feel like home,

To Alex Sanchez-Stern, who built a community around friendship, art, and socal chill.

To Andi Frank, my long-lost sister, my pumpkin-obsessed, ride-or-die roommate.

To Valentin Robert, who did nothing wrong and whose presence improves all situations.

To Kyle Ford, who is one of the warmest genuinely good people I've ever met.

To Mia Trautz, who is endlessly supportive, and my fellow tennis-trainee.

To Erik Moyer, whose determination in all things inspires me.

To Tristan Knoth, the gentlest giant whose immense chillness I can only aspire to.

To Ariana Mirian, who I used to be scared of but whose friendship is indispensable.

To Anish Tondwalkar, the most lovable teddy bear of a man who I somehow found again.

To Elizabeth, whose patio hangs and Death Cab roadtrips gave me life.

To Matt Kolosick, whose food, company, and vibes are unparalleled.

To Michael James, my adventuring buddy whose conversations and company I will miss.

This dissertation is dedicated to the people and places that kept me caffeinated,

To Subterranean Coffee, the former shared living room for all grad students in Hillcrest

To Art of Espresso, the site of daily pilgrimages and valuable research conversations

To Coppertop Coffee, which had neither good coffee nor good donuts, but was close

This dissertation is dedicated to my siblings legal and otherwise,

To Scott Renner, who I don't see enough of but whom I love.

To Mark Renner, who shared a room with me for years but somehow still likes me.

To Jeff Renner, whose natural joy I always enjoy and admire.

To John Grischuk, my best friend, confidant, and forever-friend.

To David Coy, [same as above]

To Chris Perdue, my outlaw and one of the most lovable, giving people I know.

To Kristen Perdue, my outlaw who cares more for her friends than anyone I've met.

To Chez Bob, my precious child, may you serve snacks with a smile for years to come

This dissertation is dedicated to my mom, and to my dad, whose immense contributions to my life defy summarization. I could never have done this without your unwavering support, attention, and love all these years.

Thank you, everyone.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

First off, I'd like to thank my advisor Deian Stefan for his advice, for his technical contributions, and for his friendship. I couldn't imagine making it through a PhD with anyone else. A huge thank you to my coauthors, specifically to Fraser Brown and Conrad Watt who, in addition to producing amazing work, have proven to be fantastic individuals all around. Thank you to the CSE department, faculty and staff, who have worked tirelessly in support of students. Lastly, I would like to thank the Student-Workers Union and the Student Researcher Union, who continue to fight the good fight for fair wages and a safe workplace for all students.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, Deian Stefan. ACM, 2019. The dissertation author was a primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, Deian Stefan. ACM, 2020. The dissertation author was a primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) Fraser Brown, John Renner, Andres Noetzli, Sorin Lerner, Hovav Schacham, Deian Stefan. ACM, 2020. The dissertation author was a primary investigator and author of this paper.

VITA

2017	Bachelor of Science, Software Engineering, *magna cum laude* Rochester Institute of Technology
2017-2022	Research Assistant, Computer Science University of California San Diego
2022	Doctor of Philosophy, Computer Science University of California San Diego

PUBLICATIONS

"Constant-time WebAssembly." John Renner, Sunjay Cauligi, Deian Stefan. *Principles of Secure Compilation (PriSC)*, January 2019.

"CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem." Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, Deian Stefan. , January 2019.

"Foundations for Parallel Information Flow Control Runtime Systems." Marco Vassena, Gary Soeller, Peter Amidon, Matthew Chan, John Renner, Deian Stefan *Principles of Security and Trust (POST)*, January 2019.

"Position Paper: Progressive Memory Safety for WebAssembly." Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, June 2019.

"FaCT: A DSL for Timing-Sensitive Computation." Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, Deian Stefan. *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2019.

"Towards a Verified Range Analysis for Javascript JITs." Fraser Brown, John Renner, Andres Noetzli, Sorin Lerner, Hovav Schacham, Deian Stefan. *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, June 2020.

"Scooter & Sidecar: A Domain-Specific Approach to Writing Secure Database Migrations." John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, Deian Stefan. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, June 2021

ABSTRACT OF THE DISSERTATION

Language Techniques for Automated Verifiction of Web Security

by

John Renner

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Deian Stefan, Chair

Web applications are often responsible for sensitive user data, but are exceedingly difficult to secure. On the backend, they lack effective tools to prevent data leakage, meanwhile bugs in the browser can compromise otherwise secure code. While developers have improved this situation by employing informal bugfinding techniques such as fuzzing, new bugs are commonly discovered and exploited in the wild. Formal verification tools promise to stem the tide of bugs, but they are difficult to use in practice; theorem provers are cumbersome while general-purpose automated verifiers fail to scale to real-world programs. Domain-specific languages (DSLs) allow us to make automated verification tractable by "baking in" a particular verification problem to the language semantics. In this dissertation we introduce three such DSLs designed to automatically provide security guarantees for web applications:

1. *CT-Wasm*, an extension to the WebAssembly type system which provides formally guaranteed protection against timing side-channel attacks.

2. *Scooter*, a domain-specific language which prevents server-side data leakage in database-backed applications.

3. *VeRA*, a subset of C++ which verifies the correctness of range analysis in Firefox.

Introduction

Today's internet users are caught in the middle of an ever-escalating arms race between hackers trying to their steal data, and the application developers trying to protect them. Modern browsers rely immense test suites and extensive fuzzing to find and prevent bugs before they can be exploited[238, 182], yet actively exploited vulnerabilities are found with shocking regularity[284]. Meanwhile, web application developers lack the tools and infrastructure to prevent even simple data leakage[231, 142]. It is clear that despite the great success of the tools available today, users are still at significant risk. If we want to ensure the security of users on the internet, we need stronger guarantees than those provided by randomized testing—we need guarantees rooted in formal methods. Unfortunately, today's formal verification techniques are limited in two key ways. Tools like interactive theorem provers are too costly—it takes even proof engineering experts years to verify serious systems [288, 106, 154]. And, automated verifications tools struggle with large codebases [76]. We need techniques that can be adopted by developers building web browsers and web application.

Our thesis is that domain-specific languages, whose semantics are crafted specifically to aid an automated verifier, can bridge this gap. While this approach is unsuitable for verification in general, it can make verification of specific security properties tractable. The contributions of this dissertation are the identification of three such security properties, and for each, a DSL which allows that property to be reliably verified:

Chapter 1: Constant-time cryptography with CT-Wasm If the execution time of a cryptographic routine varies with respect to a secret input, such as a private key, then it is vulnerable to *timing attacks* which can extract that secret [156] (even over the network[77] or between VMs [220]). A program lacking this vulnerability is referred to as *constant-time*. While an automated verifier for this property exists [33], it does not scale to client-side web apps—it operates on LLVM bytecode and its performance would prohibitively impact page loads. This chapter introduces Constant-time WebAssembly (CT-Wasm) which extends WebAssembly with a set of types and operations for handling secret data. These extensions constitute an embedded DSL which allows for verification of the constant-time property as a side-effect of WebAssembly's existing, linear-time typechecking algorithm.

Chapter 2: Backend leakage prevention with Scooter Applications that handle user data have *policies* that describe which users can access what data (e.g. "Only my friends can see my birthday"). This means that they are inherently tied to an underlying data schema. As applications evolve, the schema often changes, requiring corresponding policy updates. Subtle mistakes in policy updates, can result in unintentionally granting broader access to data[158]. This chapter prevents *Scooter*, a DSL for expressing schemas, policies, and updates to both, along with *Sidecar*, a verifier which ensures that no update unintentionally expands access to data. Scooter makes it possible for developers to build server-side web apps that are not only secure by construction but evolve securely. We evaluate Scooter and Sidecar against several existing applications and show that the policy preservation property can be verified in sub-second time. Unlike CT-Wasm, Scooter compromises decidability in favor of enhanced expressiveness (see Section 2.6.1); despite this, verification times below 100 milliseconds for all inputs.

Chapter 3: Correct range analysis with VeRA JavaScript is a memory safe language, and thus requires bounds checks on all array accesses. These checks can slow down execution, so modern JavaScript engines attempt to elide them whenever possible. This elision is powered by an information-gathering pass called *range analysis* which tracks the range of possible values for all numbers; if the range of values falls entirely within bounds, the bounds check can be eliminated. Mistakes in range analysis can lead out of bounds accesses, a very useful vulnerability for attackers to exploit. This chapter introduces VeRA, a tool that verifies the correctness of range analysis in Firefox, by lowering a subset of C++ to SMT. VeRA is able to verify the correctness of several range analysis routines, and successfully found a bug in Firefox, but times out when verifying several critical routines.

Because VeRA needed to operate on existing code, the DSL (a subset of C++) is very expressive and does little to aid the underlying verifier. In response to this realization, we reimplement VeRA using a generic automated verification language backed by Corral [163]. We call this reimplementation "PrimaVera" and find that we can verify even more routines, with the help of a more powerful backend.

Chapter 1

CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem

A significant amount of both client and server-side cryptography is implemented in JavaScript. Despite widespread concerns about its security, no other language has been able to match the convenience that comes from its ubiquitous support on the "web ecosystem"—the wide variety of technologies that collectively underpins the modern World Wide Web. With the introduction of the new WebAssembly bytecode language (Wasm) into the web ecosystem, we have a unique opportunity to advance a principled alternative to existing JavaScript cryptography use cases which does not compromise this convenience.

We present Constant-Time WebAssembly (CT-Wasm), a type-driven, strict extension to WebAssembly which facilitates the verifiably secure implementation of cryptographic algorithms. CT-Wasm's type system ensures that code written in CT-Wasm is both information flow secure and resistant to timing side channel attacks; like base Wasm, these guarantees are verifiable in linear time. Building on an existing Wasm mechanization, we mechanize the full CT-Wasm specification, prove soundness of the extended type system, implement a verified type checker, and give several proofs of the language's security properties.

We provide two implementations of CT-Wasm: an OCaml reference interpreter and a native implementation for Node.js and Chromium that extends Google's V8 engine. We also implement a CT-Wasm to Wasm rewrite tool that allows developers to reap the benefits of CT-Wasm's type system today, while developing cryptographic algorithms for base Wasm environments. We evaluate the language, our implementations, and supporting tools by porting several cryptographic primitives—Salsa20, SHA-256, and TEA—and the full TweetNaCl library. We find that CT-Wasm is fast, expressive, and generates code that we experimentally measure to be constant-time.

1.1 Introduction

When implementing a cryptographic algorithm, functional correctness alone is not sufficient. It is also important to ensure properties about information flow that take into account the existence of *side channels*—ways in which information can be leaked as side-effects of the computation process. For example, the duration of the computation itself can be a side channel, since an attacker could compare different executions to infer which program paths were exercised, and work backwards to determine information about secret keys and messages.

Writing code that does not leak information via side channels is daunting even with complete control over the execution environment, but in recent years an even more challenging environment has emerged—that of *in-browser cryptography*—the implementation of cryptographic algorithms in a user's browser using JavaScript. Modern JavaScript runtimes are extremely complex software systems, incorporating just-in-time (JIT) compilation and garbage collection (GC) techniques that almost inherently expose timing side-channels [199, 197, 259]. Even worse, much of the JavaScript cryptography used in the wild is implemented by "unskilled cryptographers" [237] who do not account for even the most basic timing side channels. It is dangerous enough that unsecure, in-browser cryptography has become commonplace on the web, but the overwhelming popularity of JavaScript as a development language across all platforms [95] has driven adoption of JavaScript cryptography on the server-side as well. With multiple JavaScript crypto libraries served by the NPM package manager alone having multiple-millions of weekly downloads [249, 84, 146, 92]), many of the issues noted above are also exposed server-side.

To fundamentally address the state of crypto in the web ecosystem, a solution must simultaneously compete with the apparent convenience of JavaScript crypto for developers while having better security characteristics. Modifying complex, ever-evolving JavaScript engines to protect JavaScript code from leakage via timing channels would be a labyrinthine task. Luckily, this is not necessary: all major browsers recently added support for WebAssembly (Wasm) [271, 133].

Wasm is a low-level bytecode language. This alone provides a firmer foundation for cryptography than JavaScript: Wasm's close-to-the-metal instructions give us more confidence in its timing characteristics than JavaScript's unpredictable optimizations. WebAssembly also distinguishes itself through its strong, static type system, and principled design. Specifically, Wasm has a formal small-step semantics [133]; well-typed Wasm programs enjoy standard *progress* and *preservation* properties [275], which have even been mechanically verified [267]. These formal foundations are a crucial first step towards developing in-browser crypto with guarantees of security.

In this chapter, we go further, extending Wasm to become a verifiably secure cryptographic language. We augment Wasm's type system and semantics with cryptographically meaningful types to produce Constant-Time WebAssembly (CT-Wasm). At the type level, CT-Wasm allows developers to distinguish secret data (e.g., keys and messages) from public data. This allows us to impose *secure information flow* [226] and *constant-time* programming disciplines [50, 208] on code that handles secret data and ensure that well-typed CT-Wasm code cannot leak such data, even via timing side channels.

CT-Wasm brings together the convenience of in-browser JavaScript crypto with the security of a low-level, formally specified language. CT-Wasm allows application developers to incorporate third-party cryptographic libraries of their choosing, much as they do today with JavaScript. But, unlike JavaScript, CT-Wasm ensures that these libraries cannot leak secrets by construction—a property we guarantee via a fully mechanized proof.

CT-Wasm's type system draws from previous assembly language type systems that enforce constant-time [50]. Our system, however, is explicitly designed for the in-browser crypto use case and is thus distinguished in two key ways. First, like Wasm, we ensure that type checking is blisteringly fast, executing as a single linear pass. Second, our type system makes trust relationships explicit: CT-Wasm only allows code explicitly marked as "trusted" to declassify data, bypassing the security restrictions on secret data otherwise imposed by our type system.

Contributions. In sum, this chapter presents several contributions:

- CT-Wasm: a new low-level bytecode language that extends Wasm with cryptographically meaningful types to enable secure, in-browser crypto.
- A fully mechanized formal model of the type system and operational semantics of CT-Wasm, together with a full proof of soundness, a verified type checker, and proofs of several security properties, not least the *constant-time* property (see Section 1.2.1).
- Two implementations of CT-Wasm: we extend the W3C specification reference implementation and the real-world implementation of Wasm in V8.
- Implementations, in CT-Wasm, of several important cryptographic algorithms, including the TweetNaCl crypto library [63]. We experimentally evaluate our implementation work with respect to correctness, performance, and security.
- Support tools that allow developers to **(1)** leverage the CT-Wasm verifier to implement secure crypto code that will run on existing base Wasm implementations, in the style of the

TypeScript compiler [180], and **(2)** semi-automatically infer CT-Wasm annotations for base Wasm implementations.

Open Source. All source and data are available under an open source license at [268].

Paper Organization. We first review WebAssembly and the constant-time programming paradigm (Section 1.2) and give a brief overview of CT-Wasm (Section 3.2). In Section 1.4 we describe the CT-Wasm language and its semantics. Our mechanized model and formal security guarantees are detailed in Section 1.5. We describe our implementations, supporting tools, and evaluation in Sections 1.6. We review related work in Section 3.8. Finally we discuss future work in Section 1.8 and conclude.

1.2 Background

In this section we give a brief overview of the constant-time programming paradigm and the WebAssembly bytecode language. We then proceed to an overview of Constant-Time WebAssembly.

1.2.1 Constant-time Programming Paradigm

Naive implementations of cryptographic algorithms often leak information—the very information they are designed to protect—via timing side channels. Kocher [156], for example, shows how a textbook implementation of RSA can be abused by an attacker to leak secret key bits. Similar key-recovery attacks were later demonstrated on real implementations (e.g., RSA [77] and AES [56, 198]). As a result, crypto-engineering best practices have shifted to mitigate such timing vulnerabilities. Many modern cryptographic algorithms are even designed with such concerns from the start [61, 57, 59].

The prevailing approach for protecting crypto implementations against timing attacks is to ensure that the code runs in "constant time". An implementation is said to be *constant-time* if its execution time is not dependent on sensitive data, referred to as *secret* values (e.g., secret keys or messages). Constant-time implementations ensure that an attacker observing their execution behaviors cannot deduce any secret values. Though the precise capabilities of attackers vary—e.g., an attacker co-located with a victim has more capabilities than a remote attacker—most secure crypto implementations follow a conservative *constant-time programming paradigm* that altogether avoids variable-time operations, control flow, and memory access patterns that depend on secrets [94, 208].

Verifying the constant-time property (or detecting lack thereof) for a given implementation is considered one of the most important verification problems in cryptography [33, 32, 69, 31, 288, 73, 81, 106]. To facilitate formal reasoning, these works typically represent this constant-time property using a *leakage model* [74] over a small-step semantics for a given language. A leakage model is a map from program state/action to an *observation*, an abstract representation of an attacker's knowledge. For each construct of the language, the leakage model encodes what information is revealed (to an attacker) by its execution. For example, the leakage model for branching operations such as `if` or `while` leaks all values associated with the branch condition, to represent that an attacker may use timing knowledge to reason about which branch was taken [33]. Proving that a given program enjoys the constant-time property can then be abstracted as a proof that the leakage accumulated over the course of the program's execution is invariant with respect to the values of secret inputs.

In general, the leakage model of a system must encompass the behavior of hardware and compiler optimizations across all different platforms. For example, for C, operators such as division and modulus, on some architectures, are compiled to instruction sequences that have value-dependent timing. A conservative leakage model must accordingly encode these operators as leaking the values of their operands [33]. While there is unavoidably a disconnect between the abstraction of a leakage model and the actions of real-world compilers and architectures, implementations that have such formal models have proven useful in practice. For example, the

HACL* library [288] has been adopted by Firefox [64], while Fiat [106] has been adopted by Chrome.

Unfortunately, much of this work does not translate well to the web platform. Defining a leakage model for JavaScript is extremely difficult. JavaScript has many complex language features that contribute to this difficulty—from prototypes, proxies, to setters and getters [104]. Even if we restrict ourselves to well-behaving subsets of JavaScript (e.g., asm.js [138] or defensive JavaScript [65]), the leakage model must capture the behavior of JavaScript runtimes—and their multiple just-in-time compilers and garbage collectors.

Despite these theoretical shortcomings, JavaScript crypto libraries remain overwhelmingly popular [196, 155, 84, 92, 146, 249], even in the presence of native libraries which were intended to curb their use in the web ecosystem [135]. Unfortunately, these competing solutions proved inadequate. Native crypto libraries differ wildly across platforms. For example the Web Crypto [135] and the Node.js crypto [194] APIs (available to browser and server-side JavaScript respectively) barely overlap, undercutting a major motivation for using JavaScript in the first place—its cross-platform nature. They are also unnecessarily complex (e.g., the Web Crypto API, like OpenSSL, is "*the space shuttle of crypto libraries*" [121]) when compared to state-of-the-art libraries like NaCl [62]. And, worst of all, none of these native libraries implement modern cryptographic algorithms such as the Poly1305 Message Authentication Code [57], a default in modern crypto libraries [62]). As we argue in this chapter, WebAssembly can address these shortcomings, in addition to those of JavaScript. Because of its low-level nature, we can sensibly relate existing work on assembly language leakage models to Wasm and provide a formal, principled approach to reasoning about constant-time crypto code. This gives us an excellent foundation on which to build CT-Wasm. Moreover, we will show that Poly1305, among many other cryptographic algorithms, can be securely implemented in CT-Wasm. We next give an overview of Wasm and describe our extensions to the language.

1.2.2 WebAssembly

WebAssembly is a low-level bytecode language newly implemented by all major browsers. The stack-machine language is designed to allow developers to efficiently and safely execute native code in the browser, without having to resort to browser-specific solutions (e.g., Native Client [280]) or subsets of JavaScript (e.g., asm.js [138]). Hence, while Wasm shares some similarities with low-level, assembly languages, many Wasm design choices diverge from tradition. We review three key design features relevant to writing secure crypto code: Wasm's module system, type system, and structured programming paradigm. We refer the reader to [133] for an excellent, in-depth overview of Wasm.

Module system. WebAssembly code is organized into *modules*. Each module contains a set of definitions: *functions*, *global variables*, a *linear memory*, and a *table* of functions. Modules are *instantiated* by the embedding environment—namely JavaScript—which can invoke Wasm functions exported by the module, manipulate the module's memory, etc. At the same time, the embedding environment must also provide definitions (e.g., from other Wasm modules) for functions the module declared as imports.

In a similar way to Safe Haskell [257], we extend Wasm's module system to further allow developers to specify if a particular import is trusted or untrusted. In combination with our other type system extensions, this allows developers to safely delineate the boundary between their own code and third-party, untrusted code.

Strong type system. WebAssembly has a strong, static type system and an unambiguous formal small-step semantics [133]. Together, these ensure that well-typed WebAssembly programs are "safe", i.e., they satisfy progress and preservation [275, 267]. This is especially important when executing Wasm code in the browser—bytecode can be downloaded from arbitrary, potentially untrustworthy parties. Hence, before instantiating a module, Wasm engines validate (type check) the module to ensure safety. We extend the type system to enable developers to explicitly annotate secret data and extend the type checker to ensure that secrets are not leaked (directly or indirectly).

Structured programming paradigm. WebAssembly further differs from traditional assembly languages in providing structured control flow constructs instead of simple (direct/indirect) jump instructions. Specifically, Wasm provides high-level control flow constructs for branching (e.g., **if-else** blocks) and looping (e.g., **loop** construct with the **br_if** conditional branch). The structured control flow approach has many benefits. For example, it ensures that Wasm code can be validated and compiled in a single pass [133]. This provides a strong foundation for our extension: we can enforce a constant-time leakage model via type checking, in a single pass, instead of a more complex static analysis [50].

These design features position WebAssembly as an especially good language to extend with a light-weight information flow type system that can automatically impose the constant-time discipline on crypto code [261, 226, 50]. In the next section, we give an overview of our extension: CT-Wasm.

1.3 Constant-Time WebAssembly, an overview

We extend Wasm to enable developers to implement cryptographic algorithms that are verifiably constant-time. Our extension, Constant-Time WebAssembly, is rooted in three main design principles. First, CT-Wasm should allow developers to explicitly specify the sensitivity of data and automatically ensure that code handling secret data cannot leak the data. To this end, we extend the Wasm language with new secret values (e.g., secret 32-bit integers typed s32) and secret memories. We also extend the type system of Wasm to ensure that such secret data cannot be leaked either directly (e.g., by writing secret values to public memory) or indirectly (e.g., via control flow and memory access patterns), by imposing secure information flow and constant-time disciplines on code that handles secrets (see Section 1.4). We are careful to design CT-Wasm as a strict syntactic and semantic superset of Wasm—all existing Wasm code is valid CT-Wasm—although no security benefits are guaranteed without additional secrecy annotations.

Second, since Wasm and most crypto algorithms are designed with performance in mind, CT-Wasm must not incur significant overhead, either from validation or execution. This is especially true for the web use case. Overall page load time is considered one of—if not *the*—key website performance metric [243, 188], so requiring the web client to conduct expensive analyses of loaded code before execution would be infeasible. Our type-driven approach addresses this design goal—imposing almost no runtime overhead. CT-Wasm only inserts dynamic checks for indirect function calls via **call_indirect**; much like Wasm itself, this ensures that types are preserved even for code that relies on dynamic dispatch. In contrast to previous type checking algorithms for constant-time low-level code [50, 33], CT-Wasm leverages Wasm's structured control flow and strongly-typed design to implement an efficient type checking algorithm—in a single pass, we can verify if a piece of crypto code is information flow secure and constant-time (see Section 1.5.2). We implement this type checker in the V8 engine and as a standalone verified tool. Our standalone type checker can be used by crypto engineers during development to ensure their code is constant-time, even when "compiled" to legacy Wasm engines without our security annotations (see Section 1.6.3).

Third, CT-Wasm should be flexible enough to implement real-world crypto algorithms. To enforce constant-time programming, our type system is more restrictive than more traditional information flow control type systems (e.g., JIF's [186, 184] or FlowCaml's [209]). For example, we do not allow allow branching (e.g., via **br_if** or **loop**) on secret data or secret-depended memory instructions (**load**s and **store**s). These restrictions, however, are no more onerous than what developers already impose upon themselves [94, 208]: our type checker effectively ensures that untrusted code respects these (previously) self-imposed limitations. CT-Wasm does, however, provide an escape hatch that allows developers to bypass our strict requirements: explicit declassification with a new **declassify** instruction, which can be used to downgrade the sensitivity of data from secret to public. This is especially useful when developers encrypt data and thus no longer need the type system to protect it or, as Figure 1.1 shows, to reveal (and,

```
(func $nacl_secretbox_open
    ...
    (call $crypto_onetimeauth_verify)
    (i32.declassify)
    (i32.const 0)
    (if (i32.eq) (then
        ...
        (call $crypto_stream_xor)
        (return (i32.const 0))))
    (return (i32.const -1)))
```

Figure 1.1: Verified decryption in TweetNaCl relies on a declassification to terminate early (if verification fails).

by choice, explicitly leak) the success or failure of a verification algorithm. CT-Wasm allows these use cases, but makes them explicit in the code. To ensure declassification is not abused, our type system restricts the use of **declassify** to functions marked as trusted. This notion of trust is transitively enforced across function and module boundaries: functions that call trusted functions must themselves be marked trusted. This ensures that developers cannot accidentally leak secret data without explicitly opting to trust such functions (e.g., when declaring module imports). Equally important, by marking a function untrusted, developers give a swiftly verifiable contract that its execution cannot leak secret data directly, indirectly via timing channels, or via declassification.

Trust model. Our attacker model is largely standard. We assume an attacker that can **(1)** supply and execute arbitrary untrusted CT-Wasm functions on secret data and **(2)** observe the runtime behavior of this code, according to the leakage model we define in Section 1.5. Since CT-Wasm does not run in isolation, we assume that the JavaScript embedding environment and all trusted CT-Wasm functions are correct and cannot be abused by the attacker to leak sensitive data. Under this model, CT-Wasm guarantees that the attacker will not learn any secrets. In practice, these guarantees allow application developers to execute untrusted, third-party crypto libraries (e.g., from content distribution networks or package managers such as NPM) without fear that leakage

$$
\begin{array}{lrl}
\text{(immediates)} & \mathit{imm} & ::= \mathit{nat} \\
\text{(secrecy types)} & \mathit{sec} & ::= \mathsf{secret} \mid \mathsf{public} \\
\text{(trust types)} & \mathit{tr} & ::= \mathsf{trusted} \mid \mathsf{untrusted} \\
\text{(packed types)} & \mathit{pt} & ::= \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32} \\
\text{(value types)} & t & ::= \mathsf{i32}'\ \mathit{sec} \mid \mathsf{i64}'\ \mathit{sec} \mid \mathsf{f32} \mid \mathsf{f64} \\
\text{(function types)} & \mathit{ft} & ::= t^* \to t^* \\
\text{(global types)} & \mathit{gt} & ::= \mathsf{mut}^?\ t
\end{array}
$$

$$
\begin{array}{rl}
\mathit{unop}_{\mathsf{i}N} & ::= \textbf{clz} \mid \textbf{ctz} \mid \textbf{popcnt} \\
\mathit{unop}_{\mathsf{f}N} & ::= \textbf{neg} \mid \textbf{abs} \mid \textbf{ceil} \mid \textbf{floor} \mid \\
 & \quad \textbf{trunc} \mid \textbf{nearest} \mid \textbf{sqrt} \\
\mathit{binop}_{\mathsf{i}N} & ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div\_}\mathit{sx} \mid \\
 & \quad \textbf{rem\_}\mathit{sx} \mid \textbf{and} \mid \textbf{or} \mid \textbf{xor} \mid \\
 & \quad \textbf{shl} \mid \textbf{shr\_}\mathit{sx} \mid \textbf{rotl} \mid \textbf{rotr} \\
\mathit{binop}_{\mathsf{f}N} & ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div} \mid \\
 & \quad \textbf{min} \mid \textbf{max} \mid \textbf{copysign} \\
\mathit{testop}_{\mathsf{i}N} & ::= \textbf{eqz} \\
\mathit{relop}_{\mathsf{i}N} & ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt\_}\mathit{sx} \mid \textbf{gt\_}\mathit{sx} \mid \\
 & \quad \textbf{le\_}\mathit{sx} \mid \textbf{ge\_}\mathit{sx} \\
\mathit{relop}_{\mathsf{f}N} & ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt} \mid \textbf{gt} \mid \textbf{le} \mid \textbf{ge} \\
\mathit{cvtop} & ::= \textbf{convert} \mid \textbf{reinterpret} \mid \\
 & \quad \textbf{classify} \mid \textbf{declassify} \\
\mathit{sx} & ::= \textbf{s} \mid \textbf{u}
\end{array}
$$

$$
\begin{array}{lrl}
\text{(constants)} & k & ::= \dots \\
\text{(instructions)} & e & ::= \textbf{unreachable} \mid \textbf{nop} \mid \textbf{drop} \mid \textbf{select}\ \mathit{sec} \mid \\
 & & \quad \textbf{block}\ \mathit{ft}\ e^*\ \textbf{end} \mid \textbf{loop}\ \mathit{ft}\ e^*\ \textbf{end} \mid \\
 & & \quad \textbf{if}\ \mathit{ft}\ e^*\ \textbf{else}\ e^*\ \textbf{end} \mid t\textbf{.const}\ k \mid \\
 & & \quad \textbf{br}\ \mathit{imm} \mid \textbf{br\_if}\ \mathit{imm} \mid \textbf{br\_table}\ \mathit{imm}^+ \mid \\
 & & \quad \textbf{return} \mid \textbf{call}\ \mathit{imm} \mid \textbf{call\_indirect}\ (\mathit{tr},\mathit{ft}) \mid \\
 & & \quad \textbf{get\_local}\ \mathit{imm} \mid \textbf{set\_local}\ \mathit{imm} \mid \\
 & & \quad \textbf{tee\_local}\ \mathit{imm} \mid \textbf{get\_global}\ \mathit{imm} \mid \\
 & & \quad \textbf{set\_global}\ \mathit{imm} \mid \\
 & & \quad t\textbf{.load}\ (\mathit{pt\_sx})^?\ a\ o \mid t\textbf{.store}\ \mathit{pt}^?\ a\ o \mid \\
 & & \quad \textbf{memory.size} \mid \textbf{memory.grow} \mid \\
 & & \quad t.\mathit{unop}_t \mid t.\mathit{binop}_t \mid t.\mathit{testop}_t \mid \\
 & & \quad t.\mathit{relop}_t \mid t.\mathit{cvtop}\ t\_\mathit{sx}^?
\end{array}
$$

$$
\begin{array}{lrl}
\text{(functions)} & \mathit{func} & ::= \mathit{ex}^*\ \textbf{func}\ (\mathit{tr},\mathit{ft})\ \textbf{local}\ t^*\ e^* \mid \\
 & & \quad \mathit{ex}^*\ \textbf{func}\ (\mathit{tr},\mathit{ft})\ \mathit{imp} \\
\text{(globals)} & \mathit{glob} & ::= \mathit{ex}^*\ \textbf{global}\ \mathit{gt}\ e^* \mid \mathit{ex}^*\ \textbf{global}\ \mathit{gt}\ \mathit{imp} \\
\text{(tables)} & \mathit{tab} & ::= \mathit{ex}^*\ \textbf{table}\ n\ \mathit{imm}^* \mid \mathit{ex}^*\ \textbf{table}\ n\ \mathit{imp} \\
\text{(memories)} & \mathit{mem} & ::= \mathit{ex}^*\ \textbf{memory}\ n\ \mathit{sec} \mid \\
 & & \quad \mathit{ex}^*\ \textbf{memory}\ n\ \mathit{sec}\ \mathit{imp} \\
\text{(imports)} & \mathit{imp} & ::= \textbf{import}\ \text{“}\mathit{name}\text{” “}\mathit{name}\text{”} \\
\text{(exports)} & \mathit{ex} & ::= \textbf{export}\ \text{“}\mathit{name}\text{”} \\
\text{(modules)} & \mathit{mod} & ::= \textbf{module}\ \mathit{func}^*\ \mathit{glob}^*\ \mathit{tab}^?\ \mathit{mem}^?
\end{array}
$$

$$
\begin{array}{lll@{\qquad}lll}
\mathsf{sec}\ (\mathsf{i}N'\ \mathit{sec}) & \triangleq & \mathit{sec} & \mathsf{i}N & ::= & \mathsf{i}N'\ \mathsf{public} \\
\mathsf{sec}\ \mathsf{f}N & \triangleq & \mathsf{public} & \mathsf{s}N & ::= & \mathsf{i}N'\ \mathsf{secret}
\end{array}
$$

Figure 1.2: CT-Wasm abstract syntax as an extension of the grammar given by [133].

will occur.

1.4 CT-Wasm Semantics

We specify CT-Wasm primarily as an extension of WebAssembly's syntax and type system, with only minor extensions to its dynamic semantics. Our new secrecy and trust annotations are designed to track the flow of secret values through the program and restrict their usage to ensure both secure information flow and constant-time security. We give the CT-Wasm extended syntax in Figure 2.3, the core type system in Figure 1.3, and an illustrative selection of the runtime

$$(\text{contexts})\ \ C ::= \left\{ \begin{array}{l} \textsf{trust}\ tr,\ \textsf{func}\ (tr, ft)^*,\ \textsf{global}\ gt^*,\ \textsf{table}\ n^?, \\ \textsf{memory}\ (n, sec)^?,\ \textsf{local}\ t^*,\ \textsf{label}\ (t^*)^*,\ \textsf{return}\ (t^*)^? \end{array} \right\}$$

$$tr \succ_{\mathbf{tr}} tr' \triangleq (tr = tr') \lor (tr = \texttt{trusted} \land tr' = \texttt{untrusted})$$

$$\frac{}{C \vdash t.\textbf{const}\ c : \varepsilon \to t} \qquad \frac{}{C \vdash t.\mathit{unop} : t \to t} \qquad \frac{}{C \vdash t.\mathit{binop} : t\ t \to t}$$

$$\frac{\textsf{sec}\ t = sec}{C \vdash t.\mathit{testop} : t \to (\textsf{i32}'\ sec)} \qquad \frac{\textsf{sec}\ t = sec}{C \vdash t.\mathit{relop} : t\ t \to (\textsf{i32}'\ sec)}$$

$$\frac{t_1 \neq t_2 \qquad sx^? = \varepsilon \Leftrightarrow (t_1 = \mathbf{i}n_1'\ sec \land t_2 = \mathbf{i}n_2'\ sec \land |t_1| < |t_2|) \lor (t_1 = \mathbf{f}n \land t_2 = \mathbf{f}n')}{C \vdash t_1.\textbf{convert}\ t_2\_sx^? : t_2 \to t_1}$$

$$\frac{t_1 \neq t_2 \qquad |t_1| = |t_2| \qquad \textsf{sec}\ t_1 = \textsf{sec}\ t_2}{C \vdash t_1.\textbf{reinterpret}\ t_2 : t_2 \to t_1} \qquad \frac{(t_1 = \mathbf{i}n'\ \textsf{secret} \land t_2 = \mathbf{i}n'\ \textsf{public})}{C \vdash t_1.\textbf{classify}\ t_2 : t_2 \to t_1}$$

$$\frac{C_{\textsf{trust}} = \textsf{trusted} \qquad (t_1 = \mathbf{i}n'\ \textsf{public} \land t_2 = \mathbf{i}n'\ \textsf{secret})}{C \vdash t_1.\textbf{declassify}\ t_2 : t_2 \to t_1}$$

$$\frac{}{C \vdash \textbf{unreachable} : t_1^* \to t_2^*} \qquad \frac{}{C \vdash \textbf{nop} : \varepsilon \to \varepsilon} \qquad \frac{}{C \vdash \textbf{drop} : t \to \varepsilon} \qquad \frac{sec = \textsf{secret} \longrightarrow\ \textsf{sec}\ t = \textsf{secret}}{C \vdash \textbf{select}\ sec : t\ t\ (\textsf{i32}'\ sec) \to t}$$

$$\frac{ft = t_1^n \to t_2^m \qquad C, \textsf{label}\,(t_2^m) \vdash e^* : ft}{C \vdash \textbf{block}\ ft\ e^*\ \textbf{end} : ft} \qquad \frac{ft = t_1^n \to t_2^m \qquad C, \textsf{label}\,(t_1^n) \vdash e^* : ft}{C \vdash \textbf{loop}\ ft\ e^*\ \textbf{end} : ft}$$

$$\frac{ft = t_1^n \to t_2^m \qquad C, \textsf{label}\,(t_2^m) \vdash e_1^* : ft \qquad C, \textsf{label}\,(t_2^m) \vdash e_2^* : ft}{C \vdash \textbf{if}\ ft\ e_1^*\ \textbf{else}\ e_2^*\ \textbf{end} : t_1^n\ \textsf{i32} \to t_2^m} \qquad \frac{C_{\textsf{return}} = t^*}{C \vdash \textbf{return} : t_1^*\ t^* \to t_2^*}$$

Figure 1.3: CT-Wasm typing rules as an extension of the typing rules given by [133].

$$\frac{C_{\mathsf{label}}(i) = t^*}{C \vdash \textbf{br}\ i : t_1^*\ t^* \to t_2^*} \qquad \frac{C_{\mathsf{label}}(i) = t^*}{C \vdash \textbf{br\_if}\ i : t^*\ \mathsf{i32} \to t^*} \qquad \frac{(C_{\mathsf{label}}(i) = t^*)^+}{C \vdash \textbf{br\_table}\ i^+ : t_1^*\ t^*\ \mathsf{i32} \to t_2^*}$$

$$\frac{C_{\mathsf{trust}} = tr \quad C_{\mathsf{func}}(i) = (tr', ft) \quad tr \succ_{\mathbf{tr}} tr'}{C \vdash \textbf{call}\ i : ft} \qquad \frac{ft = t_1^* \to t_2^* \qquad C_{\mathsf{trust}} = tr \qquad tr \succ_{\mathbf{tr}} tr' \qquad C_{\mathsf{table}} = n}{C \vdash \textbf{call\_indirect}\ (tr', ft) : t_1^*\ \mathsf{i32} \to t_2^*}$$

$$\frac{C_{\mathsf{local}}(i) = t}{C \vdash \textbf{get\_local}\ i : \varepsilon \to t} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \textbf{set\_local}\ i : t \to \varepsilon} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \textbf{tee\_local}\ i : t \to t}$$

$$\frac{C_{\mathsf{global}}(i) = \mathsf{mut}^?\ t}{C \vdash \textbf{get\_global}\ i : \varepsilon \to t} \qquad \frac{C_{\mathsf{global}}(i) = \mathsf{mut}\ t}{C \vdash \textbf{set\_global}\ i : t \to \varepsilon}$$

$$\frac{C_{\mathsf{memory}} = (n, sec) \qquad \mathsf{sec}\ t = sec \qquad 2^a \leq (|tp| <)^? |t| \qquad (tp\_sz)^? = \varepsilon \lor t = \mathbf{i}m'\ sec}{C \vdash t\textbf{.load}\ (tp\_sz)^?\ a\ o : \mathsf{i32} \to t}$$

$$\frac{C_{\mathsf{memory}} = (n, sec) \qquad \mathsf{sec}\ t = sec \qquad 2^a \leq (|tp| <)^? |t| \qquad tp^? = \varepsilon \lor t = \mathbf{i}m'\ sec}{C \vdash t\textbf{.store}\ tp^?\ a\ o : \mathsf{i32}\ t \to \varepsilon}$$

$$\frac{C_{\mathsf{memory}} = (n, sec)}{C \vdash \textbf{memory.size} : \varepsilon \to \mathsf{i32}} \qquad \frac{C_{\mathsf{memory}} = (n, sec)}{C \vdash \textbf{memory.grow} : \mathsf{i32} \to \mathsf{i32}}$$

$$\frac{}{C \vdash \varepsilon : \varepsilon \to \varepsilon} \qquad \frac{C \vdash e_1^* : t_1^* \to t_2^* \qquad C \vdash e_2 : t_2^* \to t_3^*}{C \vdash e_1^*\ e_2 : t_1^* \to t_3^*} \qquad \frac{C \vdash e^* : t_1^* \to t_2^*}{C \vdash e^* : t^*\ t_1^* \to t^*\ t_2^*}$$

Figure 1.3: CT-Wasm typing rules (cont.)

(values)	v	$::=$	$t.\textbf{const}\ k$
(store index)	a	$::=$	imm
(module instances)	$inst$	$::=$	$\{\mathsf{func\_i}\ a^*,\ \mathsf{global\_i}\ a^*,\ \mathsf{table\_i}\ a^?,\ \mathsf{mem\_i}\ a^?\}$
(function closures)	cl	$::=$	$\{\mathsf{instance\_ind}\ a,\ \mathsf{type}\ (tr,ft),\ \mathsf{code}\ func\} \mid \{\mathsf{type}\ (tr,ft), \mathsf{host}\ ...\}$
(memory instances)	mi	$::=$	$byte^*$
(store)	s	$::=$	$\{\mathsf{inst}\ inst^*,\ \mathsf{func}\ cl^*,\ \mathsf{global}\ (\mathsf{mut}^?\ v)^*,\ \mathsf{table}\ cl^*,\ \mathsf{mem}\ (sec,mi)^*\}$
(administrative instructions)	e	$::=$	$\dots \mid \textbf{trap} \mid \textbf{callcl}\ cl \mid \textbf{label}_n\{e^*\}\ e^*\ \textbf{end} \mid \textbf{local}_n\{i;v^*\}\ e^*\ \textbf{end}$
(configurations)	c	$::=$	$s;v^*;e^*$

$$\begin{array}{rcll}
s;vs;\ (\mathsf{s}N.\textbf{const}\ k)\ t_2.\textbf{declassify}\ t_1 & \rightsquigarrow_i & s;vs;\ (\mathsf{i}N.\textbf{const}\ k) & \\
s;vs;\ (\mathsf{i}N.\textbf{const}\ k)\ t_2.\textbf{classify}\ t_1 & \rightsquigarrow_i & s;vs;\ (\mathsf{s}N.\textbf{const}\ k) & \\
s;vs;\ v_1\ v_2\ ((\textbf{i32}'\ sec).\textbf{const}\ 0)\ \textbf{select}\ sec' & \rightsquigarrow_i & s;vs;v_2 & \\
s;vs;\ v_1\ v_2\ ((\textbf{i32}'\ sec).\textbf{const}\ k+1)\ \textbf{select}\ sec' & \rightsquigarrow_i & s;vs;v_1 & \\
s;vs;\ (\textbf{i32.const}\ k)\ \textbf{call\_indirect}\ (tr,ft) & \rightsquigarrow_i & s;vs;\textbf{callcl}\ cl & \begin{array}{l}\text{if } \mathsf{table\_i}\ ((\mathsf{inst}\ s)!i) = a \\ \text{and } ((\mathsf{table}\ \mathrm{i})!a)!k = cl \\ \text{and type}\ \ cl = (tr,ft)\end{array} \quad (*) \\
s;vs;\ (\textbf{i32.const}\ k)\ \textbf{call\_indirect}\ (tr,ft) & \rightsquigarrow_i & s;vs;\textbf{trap} & \text{otherwise}
\end{array}$$

$(*)$ **callcl** *cl* represents a function closure about to be entered as a local context. It is used to define a unifying dynamic semantics for the various forms of function call in Wasm, and its semantics is unchanged from [133].

Figure 1.4: Selected CT-Wasm semantic definitions, extended from [133]. Since the vast majority of the reduction rules are unchanged, we give only a few examples here.

reduction rules in Figure 1.4.

We now consider aspects of the base WebAssembly specification, and describe how they are extended to form Constant-Time WebAssembly.

1.4.1 Instances

WebAssembly's typing and runtime execution are defined with respect to a module *instance*. An instance is a representation of the global state accessible to a WebAssembly configuration (program) from link-time onwards. In Figure 1.3, the typing context C abstracts the *current instance*. In Figure 1.4, the small-step runtime reduction relation is indexed by the current instance i.

Instances are effectively a collection of indexes into the *store*, which keeps track of all global state potentially shared between different configurations.[1] If an element of the WebAssembly store (e.g., another module's memory or function) is not indexed by the current instance, the executing WebAssembly code is, by construction, prevented from accessing it.

1.4.2 Typing and Value Types

Wasm is a stack-based language. Its primitive operations produce and consume a fixed number of *value types*. Wasm's type system assigns each operation a type of the form $t^* \to t'^*$, describing (intuitively) that the operation requires a stack of values of type t^* to execute, and will produce a stack of values of type t'^* upon completion. The type of a Wasm code section (a list of operations) is the composition of these types, with a given operation potentially consuming the results of previous operations.

Base Wasm has four value types: i32, i64, f32, and f64, representing 32 and 64 bit integer

[1]In [133], all instances are held as a list in the store, with evaluation rules parameterized by an index into this list. The "live" specification recently changed this so that evaluation rules are directly parameterized by an instance [269]. We give our semantics as an extension of the original paper definition, although the transformation is ultimately trivial, so we will often refer to the current instance index as the "current instance".

and floating point values, respectively. To allow developers to distinguish between public and secret data, we introduce new value types which denote *secret values*. Formally, we first define secrecy annotations, *sec*, which can take two possible values: secret or public. We then extend the integer value types so that they are parameterized by this annotation. For syntactic convenience, we define the existing i32 and i64 WebAssembly type annotations as denoting *public (integer) values*, with new annotations s32 and s64 representing *secret (integer) values*. Floating point types are always considered public, since most floating point operations are variable-time and vulnerable to timing attacks [35, 157, 36].

As shown in Figure 1.3, all CT-Wasm instructions (except **declassify**) preserve the secrecy of data. We do not introduce any subtyping or polymorphism of secrecy for existing Wasm operations over integer values; pure WebAssembly seeks to avoid polymorphism in its type system wherever practical, a paradigm we continue to emulate. Instead, we make any necessary conversions explicit in the syntax of CT-Wasm. For example, the existing i32.**add** instruction of Wasm is interpreted as operating over purely public integers, while a new s32.**add** instruction is added for secret integers. We introduce an explicit **classify** operation which relabels a public integer value as a secret. This allows us to use public values wherever secret values are required; this is safe, and makes such a use explicit in the representation of the program.

Together with the control flow and memory access restrictions described below, our type system guarantees an information flow property: ensuring that, except through **declassify**, public computations can never depend on secret values. We give a mechanized proof of this in Section 1.5.2.

1.4.3 Structured Control Flow

Our type system enforces a constant-time discipline on secret values. This means that we do not allow secret values to be used as conditionals in control flow instructions, such as **call_indirect**, **br_if**, **br_table**, or **if**; only public values can be used as conditionals. This is an onerous restriction, but it is one that cryptography implementers habitually inflict on themselves in pursuit of security. Indeed, it is described as best-practice in cryptography implementation style guides [94], and as discussed throughout this chapter, many theoretical works on constant-time model such operations as unavoidably leaking the value of the conditional to the attacker.

Our type system does, however, allow for a limited form of secret conditionals with the **select** instruction. This instruction takes three operands and returns the first or second depending on the third, condition operand. Since secrecy of the conditional can be checked statically by the type system, secrecy annotations have no effect on the dynamic semantics of Figure 1.4. Importantly, **select** can do this without branching: conditional move instructions allow **select** to be implemented using a single, constant-time hardware instruction [147] and, for processors without such instructions a multi-instruction arithmetic solution exists [94]. In either case, to preserve the constant-time property if the conditional is secret, both arguments to **select** must be fully evaluated. This is the case in the Wasm abstract machine, but real engines must ensure that they respect this when implementing optimizations. We extend the **select** instruction with a secrecy annotation; a **select** secret instruction preserves constant-time (permitting secret conditionals), but may permit fewer optimizations.

1.4.4 Memory

Though secret value types allow us to track the secrecy of stack values, this is not enough. Wasm also features linear memories, which can also be used to store and load values. We thus annotate each linear memory with *sec*. Our type system ensures that public (resp. secret) values can only be stored in memories annotated public (resp. secret). Dually, it ensures that loads from memory annotated *sec* can only produce *sec* values. To ensure that accessing memory does not leak any information [198, 77], our type system also require that all memory indices to **load** and **store** be public. These restrictions preserve our information flow requirements in the presence of arbitrary memory operations.

Our coarse-grained approach to annotating memory is not without trade offs. Since Wasm only allows one memory per module, to store both public and secret data in memory, a developer must create a second module and import accessor functions for that module's memory. A simple micro benchmark implementing this pattern reveals a 30% slowdown for the memory operations. In practice, this is not a huge concern. Once base Wasm gains support for multiple memories [270], a module in CT-Wasm could have both public and secret memories; we choose not to implement our own solution now so as to maintain forwards compatibility with the proposed Wasm extension. Moreover, as we find in our evaluation (Section 1.6.4), many crypto algorithms don't require both secret and public memory in practice.

A yet more sophisticated and fine-grained design would annotate individual memory cells. We eschew this design largely because it would demand a more complex (and thus slower) type-checking algorithm (e.g., to ensure that a memory access at a dynamic offset is indeed of the correct sensitivity).

1.4.5 Trust and Declassification

As previously mentioned, it is sometimes necessary for CT-Wasm to allow developers to bypass the above restrictions and cast secret values to public. For example, when implementing an encryption algorithm, there is a point where we transfer trust away from information flow security to the computational hardness of cryptography. At this point, the secret data can be *declassified* to public (e.g., to be "leaked" to the outside world).

As a dual to **classify** we provide the **declassify** instruction, which transfers a secret value to its equivalent public one. Both **classify** and **declassify** exist purely to make explicit any changes in security status; as Figure 1.4 shows, these instructions do not imply any runtime cost. These security casting operations (and our annotations, in general) do, however, slightly increase the size of the bytecode when dealing with secret values (purely public computations are unaffected), but the simplicity and explicit nature of the security annotations are a worthwhile trade-off. We give experimental bytecode results for our CT-Wasm cryptographic implementations in Section 1.6.4.

To restrict the use of **declassify**, as Figure 1.3 shows, we extend function types with a *trust* annotation that specifies whether or not the function is trusted or untrusted. In turn, CT-Wasm ensures that only trusted functions may use **declassify** and escape the restrictions (and guarantees) of the CT-Wasm type system. For untrusted functions, any occurrence of **declassify** is an error. Moreover, trust is transitive: an untrusted function is not permitted to call a trusted function.

We enforce these restrictions in the typing rules for **call** and **call_indirect**. But, per the original WebAssembly specification, the **call_indirect** instruction must be additionally guarded by a runtime type check to ensure type safety. Thus we extend this runtime type check to additionally check that security annotations are respected. This is the only place in the semantics where our security annotations have any effect on runtime behavior.

Put together, our security restrictions allow CT-Wasm to communicate strong guarantees through its types. In an untrusted function, where **declassify** is disallowed, it is impossible for a secret value to be directly or indirectly used in a way that can reveal its value. Thus, sensitive information such as private keys can be passed into unknown, web-delivered functions, and so long as the function can be validated as untrusted, CT-Wasm guarantees that it will not be leaked by the action of the function. We next describe our mechanization effort, which includes a proof of this property (see Section 1.5.8).

1.5 Formal Model

We provide a fully mechanized model of the CT-Wasm language, together with several mechanized proofs of important properties, including a full proof of soundness of the extended type system, together with proofs of several strong security properties relating to information flow and constant-time. We build on top of a previous Isabelle model of WebAssembly [267], extending it with typing rules incorporating our secret types, annotations for trusted and untrusted functions, and the semantics of classification and declassification. At a rough count, we inherit ˜8,600 lines of non-comment, non-whitespace Isabelle code from the existing mechanization, with our extensions to the semantics and soundness proofs representing ˜1,700 lines of alterations and insertions. Our new security proofs come to ˜4,100 lines.

1.5.1 Soundness

We extend the original mechanized soundness proof of the model to our enhanced type system. For the most part, this amounted to a fairly mechanical transformation of the existing proof script. While we re-prove both the standard *preservation* and *progress* soundness properties, we will not illustrate the progress property in detail here, since its proof remains almost unchanged from the existing work, while the preservation property is relevant to our subsequent security proofs, and required non-trivial changes for the cases relating to function calls. Both proofs proceed by induction over the definition of the typing relation.

WebAssembly's top level type soundness properties are expressed using an extended typing rule given over configurations together with an instance, as a representation of the WebAssembly runtime state. Broadly, a configuration $c = s; vs; es$ is given a *result type* of the form ts if its operation stack, es, can be given a *stack type* of the form $[] \rightarrow ts$ under a typing context C which abstracts the instance, the store s, and local variables vs. This judgement is written as $\vdash_i c : ts$.

We further extend this so that configurations, formerly typed by ts, the *result type* of their stack, are additionally typed according to the level of trust required for their execution; configuration types now take the form $(tr,\ ts)$. For example, a configuration containing the privileged **declassify** operation will have "trusted" as the trust component of its type. The preservation property now certifies that trust is preserved by reduction along with the type of the configuration's stack. As a consequence, a configuration that is initially typed as untrusted is proven to remain typeable as untrusted across its entire execution, and will never introduce a privileged instruction at any intermediate stage of reduction.

Theorem 1.5.1.1 (preservation)**.**
Given a configuration c, if $\vdash_i c : (tr, ts)$ *and* $c \overset{a}{\rightsquigarrow}_i c'$, *then* $\vdash_i c' : (tr, ts)$.

1.5.2 Security Properties

We provide fully mechanized proofs, in Isabelle, that our type system guarantees several related language-level security properties for all untrusted code. These proofs, as well as the full definition of the leakage model, are available in [268]. We show that CT-Wasm's type system guarantees several security properties, including non-interference and constant-time. We conclude by showing that a well-typed untrusted CT-Wasm program is guaranteed to satisfy our constant-time property, the property which was the motivation for the type system's design.

Provided definitions, lemmas, and theorems are directly named according to their appearances in the mechanization, for easy reference.

1.5.3 Public Indistinguishability

We define an indistinguishability relation between WebAssembly configurations, given by $\sim_{\mathbf{c}}$. Intuitively, *(public) indistinguishability* holds between two configurations if they differ only in the values of their secret state. That is, the values and types of their public state must be equal, as must the types of their secret state. Formally, we define $\sim_{\mathbf{c}}$ over configurations in terms of indistinguishability relations for each of their components. These definitions can be found in Figure 1.5. This relation is required for the expression of the constant-time property, and mirrors the equivalence relation used for the same purpose by [50] between program states. We prove that typeability of a WebAssembly configuration is invariant with respect to $\sim_{\mathbf{c}}$.

$$t_1\textsf{\textbf{.const}}\ k_1 \sim_{\mathbf{v}} t_2\textsf{\textbf{.const}}\ k_2 \triangleq t_1 = t_2 \land (k_1 = k_2 \lor \textsf{sec}\ t_1 = \textsf{sec}\ t_2 = \textsf{secret})$$

$$e_1 \sim_{\mathbf{e}} e_2 \triangleq \begin{cases} (e_a \sim_{\mathbf{v}} e_b) & \text{if} \quad \begin{array}{l} e_1 = t_1\textsf{\textbf{.const}}\ k_1 \\ e_2 = t_2\textsf{\textbf{.const}}\ k_2 \end{array} \\ (e_a \sim_{\mathbf{e}} e_b)^n & \text{if} \quad \begin{array}{l} e_1 = \textsf{\textbf{block}}\ \mathit{ft}\ {e_a}^n\ \textsf{\textbf{end}} \\ e_2 = \textsf{\textbf{block}}\ \mathit{ft}\ {e_b}^n\ \textsf{\textbf{end}} \end{array} \quad \text{or} \quad \begin{array}{l} e_1 = \textsf{\textbf{loop}}\ \mathit{ft}\ {e_a}^n\ \textsf{\textbf{end}} \\ e_2 = \textsf{\textbf{loop}}\ \mathit{ft}\ {e_b}^n\ \textsf{\textbf{end}} \end{array} \\ \begin{array}{l} (e_a \sim_{\mathbf{e}} e_b)^n \\ \land\ (e_c \sim_{\mathbf{e}} e_d)^m \end{array} & \text{if} \quad \begin{array}{l} e_1 = \textsf{\textbf{if}}\ \mathit{ft}\ {e_a}^n\ \textsf{\textbf{else}}\ {e_c}^m\ \textsf{\textbf{end}} \\ e_2 = \textsf{\textbf{if}}\ \mathit{ft}\ {e_b}^n\ \textsf{\textbf{else}}\ {e_d}^m\ \textsf{\textbf{end}} \end{array} \quad \text{or} \quad \begin{array}{l} e_1 = \textsf{\textbf{label}}_n\{{e_a}^n\}\ {e_c}^m\ \textsf{\textbf{end}} \\ e_2 = \textsf{\textbf{label}}_n\{{e_b}^n\}\ {e_d}^m\ \textsf{\textbf{end}} \end{array} \\ \begin{array}{l} (v_a \sim_{\mathbf{v}} v_b)^n \\ \land\ (e_a \sim_{\mathbf{e}} e_b)^m \end{array} & \text{if} \quad \begin{array}{l} e_1 = \textsf{\textbf{local}}_n\{i; {v_a}^n\}\ {e_a}^m\ \textsf{\textbf{end}} \\ e_2 = \textsf{\textbf{local}}_n\{i; {v_b}^n\}\ {e_b}^m\ \textsf{\textbf{end}} \end{array} \\ e_1 = e_2 & \text{otherwise} \end{cases}$$

$$\left\{\begin{array}{l} \mathit{inst_1}^*, \\ \mathit{func_1}^*, \\ (\mathit{mut_1}, \mathit{glob_1})^*, \\ \mathit{table_1}^?, \\ (\mathit{sec_1}, \mathit{mem_1})^? \end{array}\right\} \sim_{\mathbf{s}} \left\{\begin{array}{l} \mathit{inst_2}^*, \\ \mathit{func_2}^*, \\ (\mathit{mut_2}, \mathit{glob_2})^*, \\ \mathit{table_2}^?, \\ (\mathit{sec_2}, \mathit{mem_2})^? \end{array}\right\} \triangleq \begin{array}{l} (\mathit{inst_1} = \mathit{inst_2})^* \\ \land\ (\mathit{func_1} = \mathit{func_2})^* \\ \land\ (\mathit{mut_1} = \mathit{mut_2} \land \mathit{glob_1} \sim_{\mathbf{v}} \mathit{glob_2})^* \\ \land\ (\mathit{table_1} = \mathit{table_2})^? \\ \land \left(\begin{array}{l} (\textsf{size}\ \mathit{mem_1} = \textsf{size}\ \mathit{mem_2} \land \mathit{sec_1} = \mathit{sec_2} = \textsf{secret}) \\ \lor\ (\mathit{mem_1} = \mathit{mem_2} \land \mathit{sec_1} = \mathit{sec_2} = \textsf{public}) \end{array} \right)^? \end{array}$$

$$s_1; {v_1}^*; {e_1}^* \ \sim_{\mathbf{c}}\ s_2; {v_2}^*; {e_2}^* \triangleq (s_1 \sim_{\mathbf{s}} s_2) \land (v_1 \sim_{\mathbf{v}} v_2)^* \land (e_1 \sim_{\mathbf{e}} e_2)^*$$

Figure 1.5: Definition of $\sim_{\mathbf{c}}$.

$$s; vs;\ (\mathsf{s32.}\textbf{const}\ k_1)\ (\mathsf{s32.}\textbf{const}\ k_2)\ \mathsf{s32.}\mathit{binop} \quad \overset{a}{\leadsto}_i \quad s; vs;\ (\mathsf{s32.}\textbf{const}\ (k_1\ \mathit{binop}\ k_2))$$

$$\text{with } a \quad \triangleq \quad \mathsf{binop\_action}(\mathit{binop}, (\mathsf{s32.}\textbf{const}\ k_1), (\mathsf{s32.}\textbf{const}\ k_2))$$

$$s; vs;\ v_1^n\ \textbf{callcl}\ cl \quad \overset{a}{\leadsto}_i \quad s'; vs';\ v_2^m \quad (*)$$

$$\begin{aligned} \text{with } cl \quad &\triangleq \quad \{\mathsf{type}\ (tr, t_1^n \to t_2^m), \mathsf{host}\ ...\} \\ a \quad &\triangleq \quad \mathsf{host\_action}(s, v_1^n, s', v_1^m, cl) \end{aligned}$$

$$a_1 \sim_{\mathbf{a}} a_2 \triangleq \begin{cases} op_1 = op_2 & \text{if} \quad \begin{array}{l} a_1 = \mathsf{binop\_action}(op_1, v_1, v_2) \\ a_2 = \mathsf{binop\_action}(op_2, v_1', v_2') \end{array} \quad \text{and} \quad \text{is\_safe\_binop}(op_1) \\ s_1 \sim_{\mathbf{s}} s_2 \\ \wedge\ (v_1 \sim_{\mathbf{v}} v_2)^n \\ \wedge\ s_1' \sim_{\mathbf{s}} s_2' & \text{if} \quad \begin{array}{l} a_1 = \mathsf{host\_action}(s_1, v_1^n, s', v_{1'}^m, cl_1) \\ a_2 = \mathsf{host\_action}(s_2, v_2^n, s_2', v_{2'}^m, cl_2) \end{array} \quad \text{and} \quad \mathsf{trust}(cl_1) = \mathsf{untrusted} \\ \wedge\ (v_{1'} \sim_{\mathbf{v}} v_{2'})^m \\ \wedge\ cl_1 = cl_2 \\ \ldots \\ a_1 = a_2 & \text{otherwise} \end{cases}$$

$(*)$ The full axiomatic description of host function behavior is not reproduced here. Full details can be found in [267], or in our mechanization.

Figure 1.6: Example of CT-Wasm action annotations and equivalence.

Lemma 1.5.3.1 (equivp_config_indistinguishable)**.**

$\sim_{\mathbf{c}}$ is an equivalence relation.

Lemma 1.5.3.2 (config_indistinguishable_imp_config_typing)**.**

If $\vdash_i c : (tr, ts)$, then for all c' such that $c \sim_{\mathbf{c}} c'$, $\vdash_i c' : (tr, ts)$.

1.5.4 Action Indistinguishability

The constant-time property is most naturally expressed as an equivalence of *observations*, which are abstractions of an attacker's knowledge defined with respect to the *leakage model* of the system. We adopt a leakage model which extends the leakiest model depicted by [33], accounting for leakage of branch conditions, memory access patterns, and the operand sizes of unsafe binary operations, namely division and modulus. In addition, we must express our trust in the host environment that WebAssembly is embedded within. A host function marked as untrusted will leak all public state it has access to when called, but never any secret state. In reality, the host environment is the web browser's JavaScript engine, and user-defined JavaScript is treated as trusted, so this corresponds to trusting that the engine's provided built-in functions are not malicious or compromised, and obey the properties guaranteed by the untrusted annotation.

We augment the WebAssembly reduction relation with state-parameterized actions, as $c \overset{a}{\leadsto}_i c'$, effectively defining a labelled transition system. Traditionally, a constant-time proof in the style of [50] would define its leakage model as a function from either action or state to a set of observations. However, it is instead convenient for us to adopt a novel representation of the leakage model as an equivalence relation, given by $\sim_{\mathbf{a}}$, between actions, denoting *action indistinguishability*. Intuitively, if two actions are defined as being equivalent by $\sim_{\mathbf{a}}$, this implies that they are indistinguishable to an attacker. This definition is inspired by the *low view* equivalence relations seen in formal treatments of information flow [226], which are used to embody an attacker's view of a system. An illustration of our definitions can be found in Figure 1.6.

This approach is helpful because the behavior of the CT-Wasm host environment, as inherited from Wasm, is specified entirely axiomatically, and may leak a wide variety of differently-typed state, making a set-based definition of leakage unwieldy. For completeness, we sketch a more traditional leakage model as a supplement in the mechanization, although this leakage model does not capture the full range of observations induced by the leakage of the host environment, because, as mentioned, such a definition would be overly complicated when we have a simpler alternative.

Having chosen our equivalence-based representation of the leakage model, *observations* become instances of a quotient type formed with respect to $\sim_{\mathbf{a}}$. This notion will be made precise in Section 1.5.9.

Lemma 1.5.4.1 (equivp_action_indistinguishable)**.**
$\sim_{\mathbf{a}}$ is an equivalence relation.

This representation allows us to define a configuration being constant-time as a property of trace equivalence with respect to $\sim_{\mathbf{a}}$. However, one final issue must be ironed out. Taking informal definitions of "trace" and "observation" for illustrative purposes, the standard statement of the constant-time property for a WebAssembly configuration could naïvely read as follows:

Definition (sketch) 1.5.4.2 (naïve constant-time)**.**
A configuration-instance pair (c,i) is constant-time iff for all c' such that $c \sim_{\mathbf{c}} c'$, the trace of (c,i) and the trace of (c',i) induce the same observations.

Unfortunately, WebAssembly is not a completely deterministic language, and so this standard definition does not apply, as a configuration cannot be uniquely associated with a trace. There are two ways we can address this. First, we can alter the semantics of WebAssembly to make it deterministic. But, despite WebAssembly's non-determinism being highly trivial in most respects, one of our goals is for CT-Wasm to be a strict extension to WebAssembly's existing semantics. Instead, we choose to generalize the standard definition of constant-time so that it can be applied to non-deterministic programs, in an analogous way to known possibilistic generalizations of security properties such as non-interference [111, 174]. A formal statement and proofs related to this generalized definition will follow in Section 1.5.8.

Definition (sketch) 1.5.4.3 (non-deterministic constant-time)**.**
A configuration-instance pair (c,i) is constant-time iff, for all c' such that $c \sim_{\mathbf{c}} c'$, the set of traces of (c,i) and the set of traces of (c',i) induce the same observations.

This generalization implicitly introduces the assumption that, where more than one choice of reduction is available, the probability of a particular single step being chosen is not dependent on any secret state. For WebAssembly, we have very good reason to expect that this is the case, because, as previously mentioned, WebAssembly's non-determinism is highly trivial—also as a deliberate design decision. The only relevant non-determinism which exists in the model is the non-determinism of the **grow_memory** instruction, non-determinism of exception (**trap**) propagation, and non-determinism of the host environment. For **grow_memory**, our type system forces all inputs and outputs of the operation to be public, and our leakage model specifies that the length of the memory is leaked by the operation. For exception propagation, WebAssembly's non-determinism in this aspect is purely an artifact of the formal specification's nature as a small-step semantics, and the definition of its evaluation contexts. In a real implementation, when an exception occurs, execution halts immediately. For the host, we simply trust that the user's web browser is correctly implemented and, when making non-deterministic choices, respects secret and untrusted annotations, with respect to our leakage model.

1.5.5 Self-isomorphism

We initially prove a security property for arbitrary untrusted sections of code which is a single-step analogy to the *self-isomorphism* property [207], which, stepwise comparing the executions of all program configurations with observably equivalent state, forbids observable differences not just in the state, but in the program counter. This single-step property is very strong, and is the key to proving all of the future properties given in this section. The proof proceeds by induction over the definition of the reduction relation.

Lemma 1.5.5.1 (config_indistinguishable_imp_reduce)**.**
If $\vdash_i c : (\mathsf{untrusted}, ts)$ *for some ts, then for all* c' *such that* $c \sim_{\mathbf{c}} c'$*, if* $c \overset{a}{\rightsquigarrow}_i c_a$ *then there exists* c'_a *and* a' *such that* $c' \overset{a'}{\rightsquigarrow}_i c'_a$ *and* $c_a \sim_{\mathbf{c}} c'_a$ *and* $a \sim_{\mathbf{a}} a'$*.*

From the definition of $\sim_{\mathbf{c}}$, we know that c_a and c'_a contain the same instructions, modulo the values of secretly typed constants.

1.5.6 Bisimilarity

We now define our notion of bisimilarity. We prove that programs that vary only in their secret inputs are bisimilar to each-other while performing $\sim_{\mathbf{a}}$-equivalent actions in lockstep. This property is sometimes known as the *strong security property* [227]. Configurations in WebAssembly always reduce with respect to an instance, so we define bisimulation in terms of configurations together with their instances.

Definition 1.5.6.1 (config_bisimulation)**.**

config_bisimulation R $\triangleq$

$$\forall ((c,i),(c',i')) \in \mathrm{R}.$$
$$(\forall c_a,\ a.\ c \overset{a}{\leadsto}_i c_a \longrightarrow \exists c'_a,\ a'.\ c' \overset{a'}{\leadsto}_{i'} c'_a \wedge a \sim_{\mathbf{a}} a' \wedge (c_a,i),(c'_a,i') \in \mathrm{R}) \wedge$$
$$(\forall c'_a,\ a'.\ c' \overset{a'}{\leadsto}_{i'} c'_a \longrightarrow \exists c_a,\ a.\ c \overset{a}{\leadsto}_i c_a \wedge a \sim_{\mathbf{a}} a' \wedge (c_a,i),(c'_a,i') \in \mathrm{R})$$

Definition 1.5.6.2 (config_bisimilar)**.**

config_bisimilar $\triangleq \bigcup$ { R | config_bisimulation R }

We prove that the set of pairs of well-typed, publicly indistinguishable configurations forms a bisimulation. From this and our definition of bisimilarity, we immediately have our version of the strong security property.

Definition 1.5.6.3 (typed_indistinguishable_pairs)**.**

typed_indistinguishable_pairs $\triangleq \{\ ((c,i),(c',i))\ |\ \vdash_i c : (\mathsf{untrusted},\ ts) \wedge c \sim_{\mathbf{c}} c'\ \}$

Lemma 1.5.6.4 (config_bisimulation_typed_indistinguishable_pairs)**.**

config_bisimulation typed_indistinguishable_pairs

Theorem 1.5.6.5 (config_indistinguishable_imp_config_bisimilar)**.**

If $\vdash_i c : (\mathsf{untrusted},\ ts)$ *for some ts, then for all* c' *such that* $c \sim_{\mathbf{c}} c'$, $((c,i),(c',i)) \in$ config_bisimilar..

1.5.7 Non-interference

We define a reflexive, transitive version of our reduction relation, given as $c \overset{as}{\rightsquigarrow}_i^* c_{as}$, annotated by an ordered list of actions. We can then prove the following property as a transitive generalization of our initial lemma, capturing the classic non-interference property. This is a strict information flow input-output property which encodes that publicly indistinguishable programs must have publicly indistinguishable outputs.

Lemma 1.5.7.1 (config_indistinguishable_trace_noninterference)**.**
If $\vdash_i c : (\mathsf{untrusted}, ts)$ *for some ts, then for all* c' *such that* $c \sim_{\mathbf{c}} c'$*, if* $c \overset{as}{\rightsquigarrow}_i^* c_{as}$ *then there exists* c'_{as} *and* as' *such that* $c' \overset{as'}{\rightsquigarrow}_i^* c'_{as}$ *and* $c_{as} \sim_{\mathbf{c}} c'_{as}$ *and as pairwise* $\sim_{\mathbf{a}}$ *with* as'*.*

1.5.8 Constant-time

We now formally discuss the constant-time property we originally sketched (Section 1.5.4.3). We define, coinductively, the set of possible traces for a configuration with respect to an instance. In Isabelle, the trace is represented by the type *action llist*, the codatatype for a potentially infinite list of actions. Equivalence between traces is then given as corecursive pairwise comparison by $\sim_{\mathbf{a}}$, written as llist_all2 $\sim_{\mathbf{a}}$ in Isabelle. We lift equivalence between traces to equivalence between sets of traces in the standard way. This is already defined as a specialization of Isabelle's built-in rel_set predicate.

Definition 1.5.8.1 (config_is_trace)**.**

$$\nexists c_a.\ c \overset{a}{\rightsquigarrow}_i c_a \longrightarrow \text{config\_is\_trace}\ (c,i)\ []$$

$c \overset{a}{\rightsquigarrow}_i c_a \wedge \text{config\_is\_trace}\ (c_a,i)\ as \longrightarrow \text{config\_is\_trace}\ (c,i)\ (a :: as)$

Definition 1.5.8.2 (config_trace_set)**.**

$\text{config\_trace\_set}\ (c,i) \triangleq \{\ as \mid \text{config\_is\_trace}\ (c,i)\ as\ \}$

Definition 1.5.8.3 (rel_set)**.**

$\text{rel\_set}\ R\ A\ B \triangleq (\forall x \in A.\ \exists y \in B.\ R\ x\ y) \wedge (\forall y \in B.\ \exists x \in A.\ R\ x\ y)$

Definition 1.5.8.4 (trace_set_equiv)**.**

trace_set_equiv $\triangleq$ rel_set (llist_all2 $\sim_{\mathbf{a}}$)

From the above, we can now formally define our constant-time property. We establish CT-Wasm's titular theorem: all typed untrusted configurations are constant-time.

Definition 1.5.8.5 (constant_time_traces)**.**

constant_time_traces $(c, i) \triangleq$

$\forall c'.\ c \sim_{\mathbf{c}} c' \longrightarrow$ trace_set_equiv (config_trace_set (c, i)) (config_trace_set (c', i))

Theorem 1.5.8.6 (config_untrusted_constant_time_traces)**.**

If $\vdash_i c$: (untrusted, *ts*) *for some ts, then (c,i) is constant-time.*

1.5.9 Observations as Quotient Types

The definition above gives the constant-time property in terms of an equivalence between trace sets, where the abstract observations of existing literature on the constant-time property are left implicit in the definition of $\sim_{\mathbf{a}}$. We now discuss how observations can be re-introduced as objects into our formalism, allowing us to adopt the standard definition of the constant-time property as equality between sets of observations.

We observe that, as $\sim_{\mathbf{a}}$ is an equivalence relation, we may use it to define a quotient type [140]. Quotient types are the type-theoretic analogy to quotient sets, where elements are partitioned into equivalence classes. Isabelle allows us to define and reason about quotient types, and to verify that particular functions over the underlying type may be *lifted* to the quotient type and remain well-defined [143]. This amounts to a proof that the function has the same value for each member of an equivalence class abstracted by the quotient type.

We can define the type of *observations* as the quotient type formed from the underlying type *action llist* with the equivalence relation being llist_all2 $\sim_{\mathbf{a}}$. Since $\sim_{\mathbf{a}}$ defines our leakage model, this *observation* type precisely characterizes the information that the model allows an attacker to observe during execution. We can then (trivially) lift the previous configuration trace set definition to observations, and give our alternative definition of the constant-time property.

Definition 1.5.9.1 (observation)**.**
$\textit{observation} \triangleq \textit{action llist} \,/\, (\text{llist\_all2} \ \sim_{\mathbf{a}})$

Lemma 1.5.9.2 (config_obs_set)**.**
The lifting of the function config_trace_set from the type ((config $\times$ inst) $\rightarrow$ (action llist) set) to the type ((config $\times$ inst) $\rightarrow$ observation set) is well-defined.

Definition 1.5.9.3 (constant_time)**.**
$\text{constant\_time}\ (c,i) \triangleq \forall c'.\ c \sim_{\mathbf{c}} c' \longrightarrow \text{config\_obs\_set}\ (c,i) = \text{config\_obs\_set}\ (c',i)$

We additionally give weaker versions of all of the results in 1.5.8 and 1.5.9 using a stronger definition of config_is_trace that is inductive rather than coinductive in [268].

1.6 Implementation

In this section we describe our CT-Wasm implementations and supporting tools. We also describe our evaluation of the CT-Wasm language design and implementation, using several cryptographic algorithms as case studies. All materials referenced here are available in [268].

1.6.1 CT-Wasm Implementations

We provide two CT-Wasm implementations: a reference implementation and a native implementation for V8 as used in both Node.js and the Chromium browser. We describe these below.

Reference implementation

We extend the Wasm reference interpreter [272] to implement the full CT-Wasm semantics. Beyond providing an easily understandable implementation of the spec, the reference interpreter serves two roles. First, it provides an easy to understand implementation of the CT-Wasm specification in a high-level language (OCaml), when compared to, say, the optimized V8 implementation. Moreover, the interpreter (unlike V8) operates on both bytecode and text-format CT-Wasm code. We found this especially useful for testing handwritten CT-Wasm crypto implementations and our V8 implementation of CT-Wasm. Second, the reference Wasm implementation also serves as the basis for a series of tools. In particular, we reuse the parsers, typed data structures, and testing infrastructure (among other parts) to build and test our supporting tools and verified type checker.

V8 implementation

WebAssembly in both Node.js and Chromium is implemented in the V8 JavaScript engine. V8 parses Wasm bytecode, validates it, and directly compiles the bytecode to a low-level "Sea of Nodes" [89] representation (also used by the JavaScript just-in-time compiler), which is then compiled to native code. We extend V8 (version 6.5.254.40) to add support for CT-Wasm. We modify the Wasm front-end to parse our extended bytecode and validate our new types. We modify the back-end to generate code for our new instructions. While the parser modifications are straightforward, our validator fundamentally changes the representation of types. V8 assumes a one-to-one correspondence between the (Sea of Nodes) machine representation of types and Wasm types. This allows V8 to simply use type aliases instead of tracking Wasm types separately. Since s32 and s64 have the same machine representation as i32 and i64, respectively, our implementation cannot do this. Our CT-Wasm implementation, instead, tracks types explicitly and converts CT-Wasm types to their machine representation when generating code; since our approach is largely type-driven, the code generation for CT-Wasm is otherwise identical to that of Wasm. By inspecting the generated assembly code, we observed that V8 does not compile the **select** instruction to constant-time assembly. We therefore implement a separate instruction selection for secret **select** so that the generated code is in constant-time.

CT-Wasm represents each instruction over secrets as a two-byte sequence—the first byte indicates if the operation is over a secret, the second indicates the actual instruction. We take this approach because the existing, single-byte instruction space is not large enough to account for all (public and secret) CT-Wasm instructions; introducing polymorphism is overly intrusive to the specification and V8 implementation. Importantly, this representation is backwards compatible: all public operations are encoded in a single byte, as per the Wasm spec. Indeed, all our modifications to the V8 engine preserve backwards compatibility—CT-Wasm is a strict superset of Wasm, and thus our changes do not affect the parsing, validation or code generation of legacy Wasm code.

1.6.2 Verified Type Checker

We provide a formally verified type checker for CT-Wasm stacks, and integrate it with our extension of the OCaml reference implementation. This type checker does not provide the informative error messages of its unverified equivalent, so we include it as an optional command-line switch which toggles its use during the validation phase of CT-Wasm execution. We validate this type checker against our conformance tests, and our crypto implementations.

The type checker is extended from the original given by [267], however major modifications needed to be made to the original constraint system and proofs. The original type checker introduced an enhanced type system with polymorphic symbols; the type of an element of the stack during type checking could either be entirely unconstrained (polymorphic), or an exact value type. We must add an additional case to the constraint system in order to produce a sound and complete algorithm; it is possible for an element of the stack to have a type that is unconstrained in its representation, but must be guaranteed to be secret. This means that in addition to the original TAny and TSome constraint types, we must introduce the additional TSecret type, and extend all previous lemmas, and the soundness and completeness proofs, for this case.

1.6.3 CT-Wasm Developer Tools

We provide two tools that make it easier for developers to use CT-Wasm: ct2wasm, allows developers to use CT-Wasm as a development language that compiles to existing, legacy Wasm runtimes; wasm2ct, on the other hand, helps developers rewrite existing Wasm code to CT-Wasm. We describe these below.

ct2wasm

Constant-Time WebAssembly is carefully designed to not only enforce security guarantees purely by the static restrictions of the type system, but also be a strict syntactic and semantic superset of WebAssembly. These facts together mean that CT-Wasm can be used as a principled development language for cryptographic algorithms, with the final implementation distributed as base WebAssembly with the crypto-specific annotations removed. In this use-case, CT-Wasm functions as a security-oriented analogy to TypeScript [180]. TypeScript is a form of statically typed JavaScript, designed to facilitate a work-flow where a developer can complete their implementation work while enjoying the benefits of the type system, before *transpiling* the annotated code to base JavaScript for distribution to end users. Similarly, CT-Wasm facilitates a work-flow where cryptography implementers can locally implement their algorithms in Constant-Time WebAssembly in order to take advantage of the information flow checks and guarantees built into our type system, before distributing the final module as base WebAssembly.

We implement a tool, ct2wasm, analogous to the TypeScript compiler, for transpiling CT-Wasm code to bare Wasm. This tool first runs the CT-Wasm type checking algorithm, then strips security annotations from the code and removes the explicit coercions between secret and public values. Moreover, all secret **select** operations are rewritten to an equivalent constant-time sequence of bitwise operations, since, as previously mentioned in Section 1.6.1, the **select** instruction is not always compiled as constant-time. Like the TypeScript compiler [179], ct2wasm does *not* guarantee the total preservation of all CT-Wasm semantics and language properties after translation, especially in the presence of other bare Wasm code not originally generated and type checked by our tool. However, we can offer some qualified guarantees even after translation.

With the exception of the **call_indirect** instruction, the runtime behaviors of CT-Wasm instructions are not affected by their security annotations, as these are used only by the type system. The **call_indirect** instruction exists to facilitate a dynamic function dispatch system emulating the behavior of higher order code, a pattern which is, to the best of our knowledge, non-existent in serious cryptographic implementations. Aside from this, bare WebAssembly interfacing with the generated code may violate some assumptions of the Constant-Time WebAssembly type system. For example, if the original code imports an untrusted function, it assumes that any secret parameters to that function will not be leaked. However at link-time, the type-erased code could have its import satisfied by a bare WebAssembly function which does not respect the untrusted contract.

ct2wasm detects these situations, and warns the developer wherever the translation may not be entirely semantics-preserving. We aim for a sound overapproximation, so that a lack of warnings can give confidence to the implementer that the translation was robust, but nevertheless one that is realistic enough that many CT-Wasm cryptographic implementations can be transpiled without warnings (see Section 1.6.4).

By default, ct2wasm assumes that the host itself is a trusted environment. This matches the assumptions made throughout the paper. The tool offers an additional *paranoid* mode, which warns the developer about every way the module falls short of total encapsulation. These conditions are likely to be too strict for many real-world cryptographic implementations designed to be used in a JavaScript environment—the conditions imply that the host is not allowed direct access to the buffer where the encrypted message is stored. But, as Wasm becomes more ubiquitous, this mode could provide additional guarantees to self-contained Wasm applications (e.g., the Nebulet micro-kernel [189]) that do not rely on a JavaScript host to execute.

wasm2ct

CT-Wasm is a useful low-level language for implementing cryptographic algorithms from the start, much like qhasm [60] and Jasmin [31]. But, unlike qhasm and Jasmin, WebAssembly is not a domain-specific language and developers may already have crypto Wasm implementations. To make it easier for developers to port such Wasm implementations to CT-Wasm, we provide a prototype tool, wasm2ct, that semi-automatically rewrites Wasm code to CT-Wasm.

At its core, wasm2ct implements an inference algorithm that determines the security labels of local variables, functions, and globals.[2] Our inference algorithm is conservative and initially assumes that every value is secret. It then iteratively traverses functions and, when assumptions are invalidated, relabels values (and the operations on those values) to public as necessary. For example, when encountering a **br_if** instruction, wasm2ct relabels the operand to public and traverses the function AST backwards to similarly relabel any of the values the operands it depends on. For safety, our tool does not automatically insert any declassification instructions. Instead, the developer must insert such instructions explicitly when the label of a value cannot be unified.

Beyond manually inserting **declassify** instructions, wasm2ct also requires developers to manually resolve the sensitivity of certain memory operations. wasm2ct does not (yet) reason about memories that have mixed sensitivity data: statically determining whether a memory load at a dynamic index is public in the presence of secret memory writes is difficult. Hence, wasm2ct assumes that all memory is secret—it does not create a separate module to automatically partition the public and secret parts. In such cases, the developer must resolve the type errors manually—a task we found to be relatively easy given domain knowledge of the algorithm. We leave the

[2]wasm2ct operates at semantic level to allow non-local, cross-function inference, but also supports a syntactic mode which rewrites Wasm text format's S-expressions. We found both to be useful: the former in porting TEA and Salsa20 without manual intervention, the latter in semi-automatically porting the TweetNaCl library.

development of a more sophisticated tool (e.g., based on symbolic execution [42]) that can precisely reason about memory—at least for crypto implementations—to future work.

1.6.4 Evaluation

We evaluate the design and implementation of CT-Wasm by answering the following questions:

1. Can CT-Wasm be used to express real-world crypto algorithms securely?
2. What is the overhead of CT-Wasm?
3. Does CT-Wasm (and ct2wasm) produce code that runs in constant-time?

To answer these questions, we manually implement three cryptographic algorithms in CT-Wasm: the Salsa20 stream cipher [61], the SHA-256 hash function [192], and the TEA block cipher [273].[3] Following [50], we chose these three algorithms because they are designed to be constant-time and *should* be directly expressible in CT-Wasm. Our implementations are straightforward ports of their corresponding C reference implementations [58, 90, 273]. For both Salsa20 and TEA, we label keys and messages as secret; for SHA-256, like [50], we treat the input message as secret.

Beyond these manual implementations, we also port an existing Wasm implementation of the TweetNaCl library [63, 244]. This library implements the full NaCl API [62], which exposes 32 functions. Internally, these functions are implemented using the XSalsa20, SHA-512, Poly1305, and X25519 cryptographic primitives. For this library, we use the wasm2ct tool to semi-automatically label values; most inputs are secret, represented as (public) "pointers" into the secret memory.

[3]TEA and several variants of the block cipher are vulnerable and should not be used in practice [153, 141, 139]. We only implement TEA to evaluate our language as a measure of comparison with [50].

We also use ct2wasm to strip labels and produce fully unannotated versions all our CT-Wasm algorithms. We run ct2wasm with *paranoid* mode off, since this corresponds to our current security model. ct2wasm reported no warnings for any of the ports, i.e., no parts of the translations were flagged as endangering the preservation of semantics, given our previously stated assumptions.

To ensure that our ports are correct, we test our implementations against JavaScript counterparts. For Salsa20 and SHA-256, we test our ports against existing JavaScript libraries, handling 4KB and 8KB inputs [149, 78]. For TEA, we implement the algorithm in JavaScript and test both our CT-Wasm and JavaScript implementations against the C reference implementation, handling 8 byte inputs [273]. Finally, for TweetNaCl, we use the Wasm library's test suite [244].

Experimental setup. We run all our tests and benchmarks on a 24-core, 2.1GHz Intel Xeon 8160 machine with 1TB of RAM, running Arch Linux (kernel 4.16.13). We use Node.js version 9.4.0 and Chromium version 65.0.3325.125, both using V8 version 6.5.254.40, for all measurements. Unless otherwise noted, our reported measurements are for Node.js. For each manually-ported crypto primitive we run the benchmark for 10,000 iterations, Salsa20 and SHA-256 processing 4KB and 8KB input messages, TEA processing 8B blocks. For TweetNaCl, we use the library's existing benchmarking infrastructure to run each function for 100 iterations, since they process huge input messages (approximately 23MB). We report the median of these benchmarks.

Expressiveness

With **declassify**, CT-Wasm can trivially express any cryptographic algorithm, even if inputs are annotated as secret, at the cost of security. We thus evaluate the expressiveness of the untrusted subset of CT-Wasm that does not rely on **declassify**. In particular, we are interested in understanding to what degree real-world crypto algorithms can be implemented as untrusted code and, when this is not possible, if the use of **declassify** is sparse and easy to audit.

We find that all crypto primitives—TEA, Salsa20, XSalsa20, SHA-256, SHA-512, Poly1305, and X25519—can be implemented as untrusted code. This is not very surprising since the algorithms are designed to be implemented in constant-time. Our port of Poly1305 did, however, require some refactoring to be fully untrusted. Specifically, we refactor an internal function (`poly1305_blocks`) to take a public value as a function argument instead of a reference to a public memory cell (since our memory is secret).

The TweetNaCl library requires a single **declassify** instruction, in the crypto_secretbox API—the API that implements secret-key authenticated encryption. As shown in Figure 1.1, we use **declassify** in the decryption function (crypto_secretbox_open) to return early if the ciphertext fails verification; this leak is benign, as the attacker already knows that any modifications to the ciphertext will fail verification [62]. A naïve port of TweetNaCl (e.g., as automatically generated by wasm2ct) would also require declassification in the crypto_sign_open API. This function operates on public data—it performs public-key verification, but relies on helper functions that are used by other APIs that compute on secrets. Since CT-Wasm does not support polymorphism over secrets, the results of these functions would need to be declassified. Trading-off bytecode size for security, we instead refactor this API to a separate untrusted module and copy these helper functions to compute on public data.

Overhead

Using our TweetNaCl and our manually ported cryptographic algorithms, we measure the overhead of CT-Wasm on three dimensions—bytecode size, validation time, and execution time. Our extended instruction set imposes a modest overhead in bytecode size due to our new annotations but imposes no overhead on non-cryptographic Wasm code. We also find that CT-Wasm does not meaningfully affect validation or runtime performance.

Bytecode size. Since CT-Wasm represents instructions over secrets as a two-byte sequence, an annotated CT-Wasm program will be as large or larger than its unannotated counterpart. For the TweetNaCl library, the unannotated, original Wasm compiles to 21,662 bytes; the bytecode size

	CT-Wasm Node			Vanilla Node	
	CT-Wasm	Wasm	ct2wasm	Wasm	ct2wasm
Salsa20	0.013	0.011	0.019	0.011	0.010
SHA-256	0.014	0.012	0.013	0.012	0.012
TEA	0.004	0.003	0.004	0.003	0.004
TweetNaCl	0.272	0.141	0.237	0.133	0.222

Figure 1.7: Median validation time (ms) of our ported crypto primitives and TweetNaCl library. We report the performance of our CT-Wasm port, the original Wasm implementation, and the ct2wasm stripped version for our modified Node.js runtime and an unmodified, vanilla Node.

of our semi-automatically annotated CT-Wasm version—including functions in the signing API that must be duplicated with public and secret versions—is 40,050 bytes, an overhead of roughly 85%. For our hand-annotated implementations of Salsa20, SHA256, and TEA, we measure the mean overhead to be 15%. The additional overhead for TweetNaCl is directly from the code duplication—the overhead of an earlier implementation that used **declassify** was roughly 18%—an overhead that can be reduced with techniques such as label polymorphism [186].

Validation. We measure the performance of the CT-Wasm type checker when validating both annotated and unannotated (via ct2wasm) code and compare its performance with an unmodified validator. Figure 1.7 summarizes our measurements. We find that our baseline validator is 14% slower than an unmodified validator, a slowdown we attribute to our representation of CT-Wasm types (see Section 1.6.1). Moving from unannotated code to annotated code incurs a cost of 20%. This is directly from the larger binary—validation in V8 is implemented as a linear walk over the bytecode. Note that though these relative slowdowns seem high, the absolute slowdowns are sub-millisecond, only occur once in the lifetime of the program, and thus have no meaningful impact on applications.

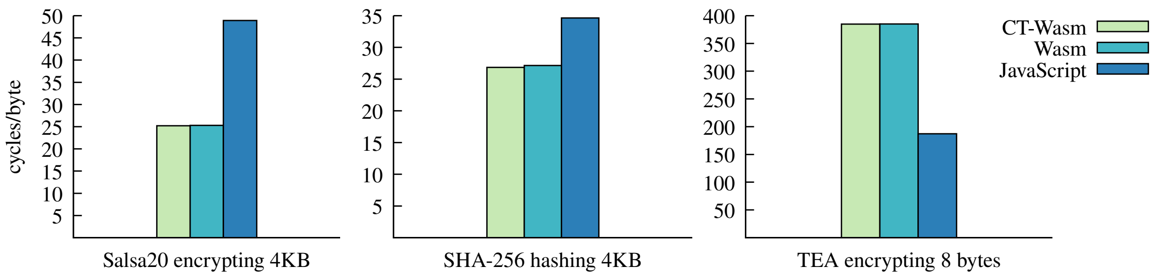


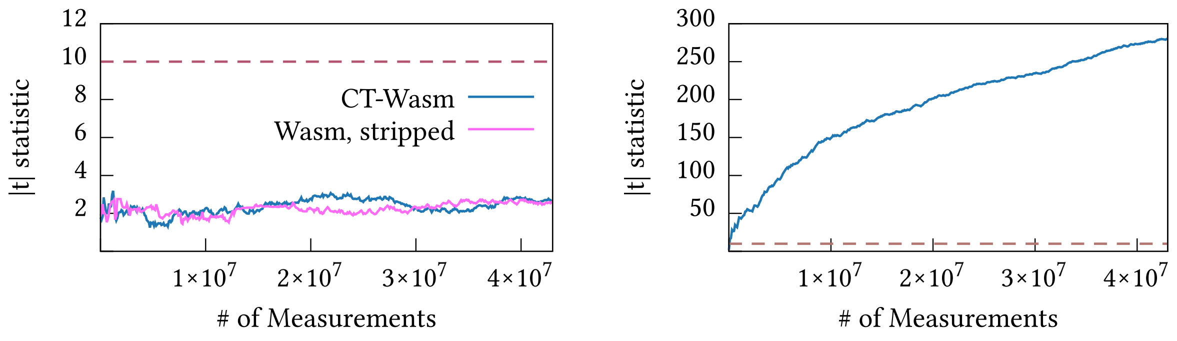
Figure 1.8: Runtime performance of handwritten crypto primitives.

Runtime. As with validation, we measure the impact of both our modified engine and of CT-Wasm annotations on runtime performance. We compare our results with reference JavaScript implementations in the case of Salsa20, SHA-256, and TEA; we compare our results with the reference TweetNaCl Wasm implementation. For both Node.js and Chromium we find that our modifications to the runtimes do not impact performance and that CT-Wasm generated code is on par with Wasm code—the mean performance overhead is less than 1%. We find our TweetNaCl implementation to be as fast as the original Wasm implementation for all the NaCl functions. Figure 1.8 compares our manual ports with JavaScript implementations: Wasm and CT-Wasm are comparable and faster than JavaScript for both Salsa20 and SHA-256. The TEA JavaScript implementation is faster than the Wasm counterparts; we believe this is because the JavaScript implementation can be more easily optimized than Wasm code that requires crossing the JavaScript-Wasm boundary.

Security

To empirically evaluate the security of our implementations, we run a modified version of dudect [219]. The dudect tool runs a target program multiple times on different input classes, collects timing information and performs statistical analysis on the timing data to distinguish the distributions of the input classes. We modify dudect to more easily use it within our existing JavaScript infrastructure. Specifically, we modify the tool to read timing information from a file and not measure program execution times itself. This allows us to record time stamps before and after running a crypto algorithm, and ignore the effects of JavaScript engine boot up time, Wasm validation, etc.

We run our modified version of dudect on the CT-Wasm and ct2wasm versions of Salsa20, SHA-256, TEA, and TweetNaCl's secretbox API. Following the methodology in [219], we measure the timing of an all-zero key versus randomly generated keys (or messages, in the case of SHA-256). All other inputs are zeroed out. We take 45 million measurements, each measurement running 10 iterations of the respective algorithm. For algorithms that can take an arbitrary message size, we use messages of 64 bytes in length. dudect compares the timing distributions of the two input classes using the Welch's t-test, with a cutoff of $|t| > 10$ to disprove the null hypothesis that the two distributions are the same. As seen in Figure 1.9a, our CT-Wasm and ct2wasm implementations for the TweetNaCl secretbox code have $|t|$ values well below the threshold of 10; this is the case for all our other algorithms as well.



(a) TweetNaCl secretbox implementation, CT-Wasm and ct2wasm

(b) Salsa20 CT-Wasm implementation, broken JavaScript harness

Figure 1.9: dudect measurements for various cryptographic algorithms.

Beyond ensuring that our CT-Wasm implementations are constant-time, running dudect revealed the subtlety of using JavaScript for crypto. In an early implementation of the Salsa20 JavaScript harness, we stored keys as arrays of 32-bit integers instead of typed byte arrays before invoking the CT-Wasm algorithm. As seen in Figure 1.9b, this version of the harness was decidedly not constant-time. We believe that time variability is due to JavaScript transparently boxing/unboxing larger integer values (e.g., those of the randomly generated keys), but leaving smaller integer values alone (e.g., those of the all-zero key).

We also discovered a second interesting case while measuring the SHA-256 JavaScript implementation: calling the hash update function once per iteration, instead of 10, caused the timing distributions to diverge wildly, with $|t|$-statistics well over 300. Placing the function call inside a loop, even for just a single iteration, caused the distributions to become aligned again, with $|t|$-statistics back under the threshold of 10. We did not observe this behavior for the CT-Wasm SHA-256 implementation and hypothesize that this time variability was due to JavaScript function inlining. We leave investigation of JavaScript timing variabilities and their impact to future work.

1.7 Related work

An initial high-level design for CT-Wasm, which this work entirely supersedes, has been previously described [216].

Low-level crypto DSLs. Bernstein's qhasm [60] is an assembly-level language used to implement many cryptographic routines, including the core algorithms of the NaCl library. However, the burden is still on the developer to write constant-time code, as qhasm has no notion of non-interference. CAO [46] and Cryptol [113] are higher-level DSLs for crypto implementations, but do not have verified non-interference guarantees.

Vale [73] and Jasmin [31] are structured assembly languages targeting high-performance cryptography, and have verification systems to prove freedom from side-channels in addition to functional correctness. Vale and Jasmin both target native machine assembly, and rely upon the Dafny verification system [164]. Vale uses a flow-sensitive type system to enforce non-interference, while Jasmin makes assertions over a constructed product program with each compilation. This work does not consider functional correctness in CT-Wasm, and uses a very simple type system to enforce non-interference. This approach scales better in the context of a user's browser quickly verifying a downloaded script for use in a web application.

High-level crypto DSLs. The HACL* [288] cryptographic library is written in constrained subsets of the F* verification language that can be compiled to C. Like CT-Wasm, HACL* provides strong non-interference guarantees. Unlike CT-Wasm, though, the proof burden is on the developer and does not come for free, i.e., it is not enforced by the type system directly. Though it currently compiles to C, the HACL* authors are also targeting Wasm as a compilation target [210]. FaCT [81] is a high-level language that compiles to LLVM which it then verifies with ct-verif [33]. CAO [46, 45] and Cryptol [113] are high-level DSLs for crypto implementations, but do not have verified non-interference guarantees. All these efforts are complementary to our low-level approach.

Leakage models. Our leakage model derives much of its legitimacy from existing work on the side-channel characteristics of low-level languages, both practical [219, 94] and theoretical [33, 50, 69]. We aim to express our top level security information flow and constant-time properties in a way that is familiar to readers of these works. Where our work differs from the above works on constant-time is in our representation of *observations*. We draw inspiration from the equivalence relation-based formalizations described by [226] for timing sensitive non-interference, which treats the number of semantic steps as its observation of a program execution. This is fundamentally related, and sometimes even given as synonymous, to the constant-time condition [50].

Our type system—which facilitates the non-interference result—can be characterized as a specialization of the Volpano-Irvine-Smith security type system [261]. Our equivalence relation-based observations are similar to the abstractions used by [51, 52]. To the best of our knowledge, our proof work is the first to use quotient types to connect such a low view equivalence representation of an attacker's observational power [226] to a proof of a leakage model-based constant-time property.

The literature on non-interference above is split as to whether traces and their associated properties are expressed inductively or coinductively. We give both interpretations, with the coinductive definition additionally capturing an observation equivalence guarantee between publicly indistinguishable non-terminating programs, encoding that even if a program does not terminate, timing side-channels from visible intermediate side-effects will not leak secret values. [207] give a coinductive treatment of the non-interference property, but for an idealized language, and do not connect it to the constant-time property.

1.8 Future Work

We have described two approaches to using CT-Wasm, either as a native implementation or a "development language" for base Wasm. As an intermediate between these two, CT-Wasm can be "implemented" in existing engines by poly-filling the `WebAssembly` API to validate CT-Wasm code and rewrite it to Wasm. Doing this efficiently is, unfortunately, not as simple as compiling ct2wasm to JavaScript or WebAssembly—to avoid pauses due to validation, ct2wasm must be implemented efficiently (e.g., at the very least as a streaming validator).

CT-Wasm takes a conservative approach to trust and secrecy polymorphism in order to ensure design consistency with Wasm. Even given this direction, there is possible space for relaxation, especially regarding **call_indirect** and higher-order code.

While we have experimentally validated that our cryptography implementations do not show input-dependent timing characteristics, the V8 WebAssembly implementation is still relatively new. Future implementations may implement aggressive optimizations that could interfere with our guarantees. A principled investigation of the possible implications of heuristically triggered JIT optimizations on the timing characteristics of WebAssembly would allow us to maintain our guarantees in the presence of more aggressive compiler behaviours.

We foresee CT-Wasm to be useful not only as a development language but also as target language for higher-level crypto languages. Since some of these language (e.g., HACL* [288] and FaCT [81]) are already starting to target WebAssembly, it would be fruitful extending these projects to target CT-Wasm as a secure target language instead. At the same time, extending wasm2ct to (fully) automatically infer security annotations from base Wasm would potentially prove yet more useful—this would allow developers to compile C/C++ libraries such as libsodium [102] to Wasm (e.g., with Emscripten [282]) and use wasm2ct to ensure they are secure.

1.9 Conclusion

We have presented the design and implementation of Constant-Time WebAssembly, a low-level bytecode language that extends WebAssembly to allow developers to implement verifiably secure crypto algorithms. CT-Wasm is fast, flexible enough to implement real-world crypto libraries, and both mechanically verified and experimentally measured to produce constant-time code. Inspired by TypeScript, CT-Wasm is designed to be usable today, as a development language for existing, base Wasm environments. Both as a native and development language, CT-Wasm provides a principled direction for improving the quality and auditability of web platform cryptography libraries while maintaining the convenience that has made JavaScript successful.

Acknowledgments

We thank the anonymous POPL and POPL AEC reviewers for their suggestions and insightful comments. We thank Andreas Rossberg and Peter Sewell for their support during this work. We thank Dan Gohman for insightful discussions. Conrad Watt is supported by an EPSRC Doctoral Training award, the Semantic Foundations for Interactive Programs EPSRC program grant (EP/N02706X/1), and the REMS: Rigorous Engineering for Mainstream Systems EPSRC program grant (EP/K008528/1). This work was supported in part by a gift from Cisco and by the

CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, Deian Stefan. ACM, 2019. The dissertation author was a primary investigator and author of this paper.

Chapter 2

Scooter & Sidecar: A Domain-Specific Approach to Writing Secure Database Migrations

Web applications often handle large amounts of sensitive user data. Modern secure web frameworks protect this data by (1) using declarative languages to specify security policies alongside database schemas and (2) automatically enforcing these policies at runtime. Unfortunately, these frameworks do not handle the very common situation in which the schemas or the policies need to evolve over time—and updates to schemas and policies need to be performed in a carefully coordinated way. Mistakes during schema or policy migrations can unintentionally leak sensitive data or introduce privilege escalation bugs. In this work, we present a domain-specific language (Scooter) for expressing schema and policy migrations, and an associated SMT-based verifier (Sidecar) which ensures that migrations are secure as the application evolves. We describe the design of Scooter and Sidecar and show that our framework can be used to express realistic schemas, policies, and migrations, without giving up on runtime or verification performance.

2.1 Introduction

Protecting user data in modern database-driven Web applications is hard: web developers must ensure that their application code correctly safeguards user data with every request. Unfortunately, popular Web frameworks don't help developers get this right because they don't account for security policy. They rely on developers to implement ad-hoc security mechanisms—to properly sprinkle just the right `if` statements throughout their code. When developers inevitably fail, their applications expose sensitive user data—from credentials [145] to COVID-19 test results [231] to user data from previously hacked databases [142].

Model-policy-view-controller (MPVC) frameworks (e.g., Lifty [205], Jacqueline [276], LWeb [203], and Hails [119, 118]) promise to address this problem by making data-access policies first class. MPVC frameworks allow developers to specify access-control policies on data alongside their schemas—the code describing the data model and interfaces used to access data—often using a declarative domain-specific language (DSL). For example, when defining the schema for user profiles, developers might specify a policy like "only the user can modify their profile and only they and their friends can see their email address." MPVC frameworks enforce such policies automatically, for example, when controllers—the code handling user requests—access the data model. This reduces the amount of code developers need to get right [118, 203]: instead of correctly implementing thousands of checks, they simply need to write correct, declarative policy code.

Reality, though, is more complicated than traditional MPVC frameworks suggest. Both models and policies constantly evolve through *migrations*. And unsafe schema or policy migrations can have devastating consequences—from leaking sensitive data to privilege escalation attacks.

Schema migrations modify and extend data models. When performing schema migrations, developers must (1) correctly reconcile their changes to the schema with changes to the application code (e.g., the controllers that interface with the new data modes and the view code that renders the data to, say, HTML or JSON) and (2) ensure that their changes are secure and do not violate the policy used to safeguard existing data. This is hard because migrations are typically written in low-level, error-prone database interfaces—most often, directly in SQL—rather than the high-level object relational mappers (ORMs) used in application code [265, 266]. And even if developers manage to write correct migrations (e.g., using synthesis to automatically update the application code according to the new schema [266]), they still need to ensure that their migrations abide by the policy. In practice this is manual and error prone [100, 150]: migration tools are not aware of policies, so developers must manually ensure their code abides by the policy, with no room for error—unsafe migrations (e.g., copying sensitive data to public locations) often do not break application functionality and thus go unnoticed until it's too late (e.g., the sensitive data is leaked).

Policy migrations can also introduce security vulnerabilities when extending or modifying data-access policies. Existing frameworks make no guarantees about a new policy's relationship to the old policy, which can result in users gaining access to sensitive or critical data that would otherwise have stayed safe. Such leaks were common enough in Hails (e.g., in the Task management app [232]) that the authors modified their policy DSL with a new keyword that (tried to) make it clear when a field policy was updated to be publicly accessible [118]. This isn't unique to Hails or MPVC frameworks, though: it happens in traditional MVC frameworks, too. For example, the authors of the Ghost blogging platform unintentionally allowed contributors to edit blog posts [116], and they introduced this bug in a patch *itself* designed to fix a bug in their policy code [117]. Likewise, a refactor of HotCRP's policy code [159] inadvertently granted unauthenticated users administrator rights [158].

We address unsafe migrations via three contributions.

1. The Scooter Domain-Specific Language Our first contribution is a new domain-specific language, Scooter, that allows developers to (1) declaratively specify data models and security policies on these models; and (2) write imperative schema and policy migrations, which update the data models and policies. Today, developers use wildly different languages for these tasks (e.g., ORM for describing the data model, custom DSL for policy specification, and SQL for schema migrations) and, as discussed above, ensuring that changes to models and policies are safe is a manual, error-prone task. By unifying specification and migration, Scooter makes it easier for developers to write safe migrations.

2. The Sidecar Verifier Our second contribution is a static tool, Sidecar, which verifies the safety of migrations written in Scooter. At its core, Sidecar relies on an automated procedure that determines whether one policy is as strict or stricter than another. To verify a data migration, we use a static abstract analysis to track the flow of information across the migration and use this procedure to verify that the policy used to safeguard migrated data is at least as restricting as the policies on the sources of that data. Similarly, to verify a policy migration, we use the automated procedure to compare the new policy against the old. We designed Scooter and Sidecar together to ensure that verification is fully automatic and fast. Moreover, when verification fails, we designed Sidecar to generate a counterexample to help developers understand and debug policy violations.

3. Implementation and Evaluation Our third contribution is an implementation and evaluation of the Scooter DSL and the Sidecar verifier. We implemented Scooter and Sidecar in Rust. For migrations that pass the verifier, the Scooter compiler generates (1) a *migration interpreter* that performs the verified-safe migration, (2) an authoritative specification containing the declarative model and policy, and (3) a typed Rust ORM implementation for each model. The generated ORM enforces policies automatically at run time and forces developers to update their application code to account for schema changes “for free”, i.e., by generating the ORM we ensure that schema changes manifest as type errors. We evaluate Scooter and Sidecar on seven case studies from the

MPVC literature, including a port of LWeb's Build it Break it Fix it, as well as a Ruby-on-Rails application used at UC San Diego for PhD Visit Day. We find that: (1) Scooter can express almost all policies and migrations from these previous efforts; (2) Scooter's ORM policy enforcement imposes under 11% overhead, which is comparable to previous work [203, 118]; and (3) Sidecar verifies most safe migrations in under a second, and for unsafe ones (e.g., the HotCRP migration from [158]) it generates useful counterexamples.

Open Source All source code is available under an open source license at [217].

2.2 Motivation and Overview

In this section, we give a brief overview of how migrations—both traditional database migrations and updates to policy code—can introduce security vulnerabilities and how Scooter eliminates these vulnerabilities.

The Chitter MPVC Application We use a simple social media web application, Chitter, as an example. Chitter allows users to post 42-character messages—peeps—on a public bulletin board. Though peeps are public, the app also handles sensitive information about users, e.g., follower relationships, private messages, pronouns, email addresses, and passwords. The Chitter developers are serious about protecting this data and use an MPVC framework to (1) separate the model and policy code from the rest of the application (the views and controllers) and (2) enforce the policy code automatically, at runtime. Figure 2.1 gives part of Chitter's data model for user profiles and its policy. The policy states that a user's email address is only visible to that user and Chitter administrators (who are, themselves, users); that the user's pronouns are visible to the user and the users they follow; and that the user and any admin can modify all but the `isAdmin` field, which can only be modified by admins.

```
User
  name: String
   read: public
   write: u -> [u] + User::Find({isAdmin: true})
  email: String
   read: u -> [u] + User::Find({isAdmin: true})
   write: u -> [u] + User::Find({isAdmin: true})
  pronouns: String
   read: u -> [u] + u.followers
   write: u -> [u] + User::Find({isAdmin: true})
  isAdmin: Bool
   read: u -> [u] + User::Find({isAdmin: true})
   write: u -> User::Find({isAdmin: true})
  followers: Set(User)
   read: u -> [u] + u.followers
   write u -> [u] + User::Find({isAdmin: true})
  ...
```

Figure 2.1: Chitter users model and policy in (simplified) Scooter.

2.2.1 Unsafe Migrations

Though using an MPVC framework helps the Chitter developers safeguard user data at runtime, modifying and extending the model or policy code could undermine this effort.

Unsafe Schema Migrations. Extending model schemas can inadvertently leak data and allow users to bypass data access policies. Consider, for example, changing the Chitter application by extending the user model with a new public `bio` field. To do so, Chitter developers modify their model:

```
User
...
+ bio: String
+  read: public
+  write: u -> [u] + User::Find({isAdmin: true})
...
```

They also write a migration—in SQL—that extends the underlying database to populate the new `bio` field, in this case with the user's name and pronouns:

```
ALTER TABLE user ADD bio STRING;
UPDATE user
SET bio = CONCAT("Hi! I'm ", user.name,
                 "(", user.pronouns, ").")
```

Since this migration isn't constrained by a policy, it *accidentally leaks sensitive data*—the pronouns—to all users.

Direct leaks like this are not the only concern, though. Migrations can modify data used by policy code—and unintentionally introduce leaks or privilege escalation bugs, i.e., grant users access to data they otherwise would not be able to access. For example, setting the `isAdmin` field of a user allows that user to read and write other users' profiles.

The problem is that *schema migrations are decoupled from policy code*—so developers must implicitly and informally enforce their data access policy on each migration. Doing so for Chitter is easy, but real-world migrations and policies are far more complicated.

Unsafe Policy Migrations. Changes to policy code can also introduce leaks and privilege escalation bugs. Consider further extending the Chitter application with a new hierarchy of administration: admins and moderators. Moderators, unlike admins, should only be allowed to read (and edit) free-form data like names and bios, which could contain potentially inappropriate content. To add support for moderators, the Chitter developers replace the boolean `isAdmin` field with an integer, `adminLevel`—where 0 is used for normal (unprivileged) users, 1 for moderators, and 2 for admins. They then update the model and policy in several steps.

They start by extending the user model with `adminLevel`:

```
User
...
```

```
+ adminLevel: Int
+  read:  u -> [u] + User::Find({adminLevel: 2})
+  write: u -> User::Find({adminLevel: 2})
...
```

Then, they perform a schema migration to add the new field, setting the admin level according to the old `isAdmin` field:

```
  ALTER TABLE user ADD adminLevel INT;
  UPDATE user
  SET adminLevel = CASE WHEN isAdmin
    THEN 2
    ELSE 0
  END;
```

Next, they update the policy code to use `adminLevel` instead of `isAdmin` and only then remove `isAdmin` from the model and underlying database (via another migration).

The new policy is introduced as an edit to the old:

```
User
...
  email: String
-  read: u -> [u] + User::Find({isAdmin: true})
+  read: u -> [u] + User::Find({adminLevel: 2})
-  write: u -> [u] + User::Find({isAdmin: true})
+  write: u -> [u] + User::Find({adminLevel: 2})
...
  bio: String
-  write: u -> [u] + User::Find({isAdmin: true})
```

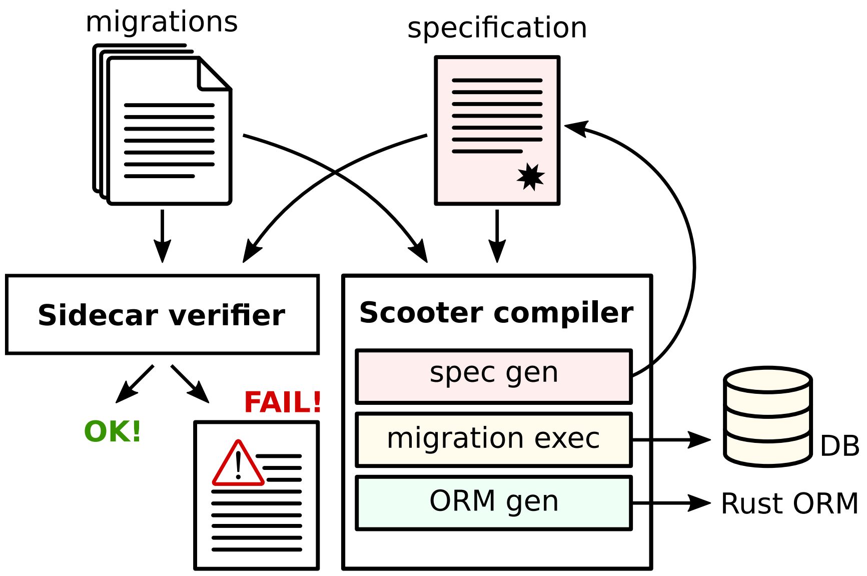


Figure 2.2: Given a migration and data model and policy specification, Sidecar verifies the safety of the migration. If the migration is unsafe, Sidecar produces a counterexample. Otherwise, the Scooter compiler (1) updates the specification, (2) executes that migration against the database, and (3) generates a type-safe policy-enforced Rust ORM, which appliations use to access persistent data.

```
+  write: u -> [u] + User::Find({adminLevel >= 0})
...
```

Alas, this policy is overly permissive: instead of restricting the bio field writers to the user, moderators, and admins, the new policy accidentally allows *any user* to write.

The problem is that *policy migrations are decoupled from policy enforcement*. Therefore, the burden is on developers to ensure that migration code doesn't sidestep the declared data access policies. Unsurprisingly, developers get this wrong—and when they do, the error is silent. Changes that inadvertently weaken policies persist until someone is lucky enough to notice them [158] or loud enough exploiting them [250].

2.2.2 Safe Migrations with Scooter and Sidecar

With Scooter, developers don't directly modify models and policies, nor do they write schema migrations using low-level interfaces like SQL. Instead, they use Scooter for both tasks

(Figure 2.2). To start, developers implement a migration that generates the initial model and policy specification; Figure 2.1 gives an example of a (simplified) specification generated by Scooter.[1] Then, all policy and schema migrations are relative to and update this specification (and its underlying database representation). Let's consider how the Chitter migrations would be implemented using Scooter.

Preventing Unsafe Schema Migrations With Scooter, the Chitter developers extend the original user model with bios by writing a migration script:

```
1 User::AddField(
2  bio : String {
3   read: public,
4   write: u -> [u] + User::Find({isAdmin:true})
5  }, u -> "I'm "+u.name+"("+u.pronouns+")");
```

This script extends the `User` model with a new string field `bio` and populates the bio according to the anonymous function on line 5. Our Sidecar verifier, however, catches the leak before it's too late: Sidecar automatically infers that `u.pronouns` data ends up in `u.bio` and that the `bio` policy is less restrictive than the `pronouns` policy. Scooter will only execute the migration after the developers modify the update function to not use `u.pronouns`.

Preventing Unsafe Policy Migrations To extend Chitter with moderators, the Chitter developers, again, write a single script (instead of multiple scripts and file edits):

```
User::AddField(
 adminLevel : I64 {
  read: u -> [u] + User::Find({adminLevel:2}),
```

[1]As we describe in Section 2.3, Scooter is slightly more verbose. For example, developers need to specify who is allowed to create and delete objects, and not just who is allowed to read and write fields.

```
 4   write: u -> User::Find({adminLevel: 2})
 5  }, u -> if u.isAdmin then 2 else 0);
 6
 7 User::UpdateFieldPolicy(email, {
 8  read: u -> [u] + User::Find({adminLevel: 2}),
 9  write: u -> [u] + User::Find({adminLevel: 2})
10 });
11 User::UpdateFieldWritePolicy(bio,
12  u -> [u] + User::Find({adminLevel >= 0}));
13
14 User::RemoveField(isAdmin);
```

This migration would add the new `adminLevel` field, update the email and bio policies accordingly, and remove the old `isAdmin` field. Sidecar, however, catches the unsafe policy update on lines 11–12, stops Scooter from executing the migration, and generates a counterexample showing the policy violation:

```
Principal: User(0)
# CAN NOW ACCESS:
User { id: User(1),
       isAdmin: false,
       adminLevel: 0,
       bio: "",
       ...  }
# OTHER RECORDS:
User { id: User(0),
       isAdmin: false,
       adminLevel: 0,
            ...  }
```

This, in effect, forces the Chitter developers to fix the unsafe policy. They could do this by rewriting the policy to:

```
        u -> [u] + User::Find({adminLevel: 2}),
```

This policy is safe—it is equivalent to the old one—but it's not the intended policy. The intended policy *is* more permissive: it should allow the user, admins, *and moderators* to edit `bios`.

Scooter allows developers to weaken policies but requires them to be explicit about the change being more permissive:

```
      User::WeakenFieldWritePolicy(bio,
        u -> [u] + User::Find({adminLevel > 0}),
        "Reason: allow moderators to update bios.");
```

Being explicit does not prevent developers from getting such migrations wrong (e.g., using `>=` instead of `>`). It does, however, make it easier to audit migrations and narrow the focus of security reviews.

Safe Migrations Workflow As Figure 2.2 shows, once Sidecar verifies the safety of a migration, Scooter performs the actual database migration and updates the model specification to reflect any model or policy changes. The model (and policy) specification is then used to generate a Rust ORM library, which allows application code to retrieve and modify persisted objects using a standard high-level typed interface (e.g., `User::Find`) that automatically enforces data-access policies. Like previous work [203], generating ORM code from the DSL specification also forces developers—via normal compiler type errors—to update the view and controller code to account for schema changes. We describe Scooter in more detail next.

(variable)	*var* ::=	$x_0, x_1, .., x_n$
(datetime)	*datetime* ::=	now | *d*¡month¿-¡day¿-¡year¿-¡hour¿:¡minute¿:¡second¿
(constant)	*const* ::=	*string*, *integer*, *float*, *datetime*, true, false, public
(binary ops)	*binop* ::=	*gencmp* | *op* | *numcmp*
	op ::=	+ | -
	numcmp ::=	< | <= | > | >=
	gencmp ::=	== | !=
(find operators)	*fop* ::=	: | $\ni$ | *numcmp*
(set literal)	*set* ::=	$[e_0, .., e_n]$
(functions)	*func* ::=	*var* -> *e*
(expressions)	*e* ::=	*const* | *set* | *var* | !*e* | (*e* *binop* *e*)
		| (if *e* then *e* else *e*) | (match *e* as *var* in *e* else *e*) | None | Some(*e*)
		| *e*.map(*func*) | *e*.flat_map(*func*) | *e*.*field*
		| *Model*::ById(*e*)
		| *Model*::Find($\{field_1\ fop_1\ e_1, \ldots, field_n\ fop_n\ e_n\}$)

Figure 2.3: The syntax of value expressions shared between policies and migrations in Scooter.

2.3 Design

We built the Scooter languages according to three main design goals:

1. **Unification**. Right now, developers typically use different languages to manage their databases (e.g., SQL) define their policies (e.g., Hails), and query the database (e.g., ORMs). Scooter aims to provide a unified semantics for migrations and policy specification that mirrors popular ORM patterns.

2. **Expressiveness**. A unified language is only effective if it can express the union (or more!) of what separate languages are able to. For example, the language must be able to express both real-world policies and real application data models.

3. **Verifiability**. The verifier must catch safety violations statically, with informative and actionable error messages.

Scooter maintains a single *policy file*, written in Scooter$_p$, that contains the current schema and all policies (unification). Users *do not* manually update this file or write Scooter$_p$ directly.

Instead, Scooter automatically updates the file when users write and run Scooter_m migration scripts—the scripts that make schema changes, manipulate data, and update new policies. Before executing a migration, Scooter verifies that the migration's changes are safe with respect to the current policy (verifiability). This process is illustrated in Figure 2.2.

In this section, we elaborate on how Scooter fulfills our three design goals. First, we explain how schemas and policies are expressed in Scooter_p (§2.3.1). Then, we explain how users express changes to the schema and policy through migrations in Scooter_m—and how, as a result, Scooter is able to statically prevent migration errors (§2.3.2). Finally, we describe how users write applications using Scooter's ORM (§2.3.3).

2.3.1 Declaring Policies

Scooter expresses both schemas and policies because they are inherently coupled; policies guard access to the data defined in the schema and are themselves defined in terms of queries against that same schema.

Schemas Scooter uses a standard ORM data model, defining schemas in terms of *models* composed of typed *fields*. Figure 2.4 shows a policy file containing a simple `User` model with a single `name` field of type `String`.

To express relational data, Scooter generates an implicit `id` field that acts as a unique identifier for each instance—i.e., database row—in the model. This allows one model instance to hold a reference to another (e.g., the `bestFriend` field in Figure 2.4 refers to the user `id`). The `id` field is strongly typed—`User::id` is of type `Id(User)`, allowing type-safe object lookups.

While `id`s are powerful enough to express any relational construct—one-to-one, one-to-many, many-to-many—they cannot express all *policies* on their own (§2.6.3). Scooter's `Set` type allows users to express otherwise inexpressible policies and, moreover, makes it easy to express simple one-to-many relations. For example, a `User` may have many emails (`Set(String)`).

```
@static-principal
Unauthenticated

@principal
User {
  create: _ -> [Unauthenticated],
  delete: none,

  name: String {
    read: public,
    write: u -> [u.id]},
  bestFriend: Id(User) {
    read: u -> [u.id, u.bestFriend],
    write: u -> [u.id]},
  adminLevel: I64 {
    read: public,
    write: u -> User::Find({adminLevel: 2})
                 .map(u -> u.id)}}
```

Figure 2.4: Simple user profile and principal declaration in Scooter.

Principals and Policies In Scooter, applications operate on data through basic create, read, update, and delete (CRUD) *operations*, where each operation is performed on behalf of a *principal*. *Policies* define, for each operation, the set of principals allowed to perform that operation. For example, Figure 2.4 states that all principals can read a `User`'s name.

What is a principal? In Scooter, many principals are simply database object `id`s. To specify that a model's `id` is a valid principal, Scooter annotates that model with `@principal`. For example, the `User` in Figure 2.4 is a principal and is annotated as such. We call this kind of principal a *dynamic principal* because its existence is tied to the state of the database.

Sometimes, though, it's important to express policies in terms of application infrastructure (as opposed to database objects). Scooter uses *static principals* for this purpose. For example, Figure 2.4 declares an `Unauthenticated` static principal, which the application can use for operations not made on behalf of a logged-in user. The policy also states that the `Unauthenticated` principal is the only one allowed to create `Users`; in other words, only users who are not logged in are able to create new users. While the set of static principals varies by application, we find two to be very common: an `Unauthenticated` principal and a `Login` principal that has read access to all password data but is used sparingly.

Policy Functions Scooter expresses the relationship between operations and principals (e.g., `Users` and their ability to change their usernames) as a *policy function* from the target instance of the operation (e.g., `Users`) to a set of principals allowed to perform that operation (e.g., change usernames). For example, Figure 2.4 states that the policy for writing to `User::name` is `u -> [u.id]`. Scooter uses square brackets to denote sets, so this function says: "for any user *u*, the set of principals allowed to change its name contains only *u.id*". The language contains two convenience terms for common functions: `public`, which is a function that returns all principals, and `none`, which is the same as `_ -> []`.

For any operation on a model *m*, the type of a policy function must be $m \rightarrow$ `Set(Principal)`. Within the function, the policy is free to traverse instances and query the models to construct its output set: Scooter policies use conditionals, mathematical operations, comparisons, and comparison-based querying.

Policy functions are strongly typed expressions. This ensures that policies cannot crash at runtime—they will always produce a set of principals—and simplifies the lowering of policy expressions to SMT (for verification). The full type system for policies can be found in the appendix of the original Scooter paper[218].

2.3.2 Migrations

Users update their policies and schema by writing migration scripts in $\mathsf{Scooter}_m$. Migration scripts consist of a series of commands that modify the schema (e.g., add a field) and the policy. Crucially, $\mathsf{Scooter}_p$ and $\mathsf{Scooter}_m$ share an underlying semantics, unlike traditional MPVC frameworks where policies are expressed with models and migrations are expressed in raw SQL.

Differing semantics make migration safety verification hard, while unified semantics—as with Scooter$_p$ and Scooter$_m$—make verification easy.

Schema Changes. In Scooter, users can change schemas by creating and deleting models or by creating or deleting fields of those models. Whenever users create a model or field, they must include all `read`, `write`, `create`, and `delete` policies. Consider the following migration, which extends the policy in Figure 2.4 with public posts called Peeps:

```
 1 CreateModel(Peep {
 2   create: public,
 3   delete: p -> [p.author],
 4 
 5   author: Id(User) {
 6     read: public,
 7     write: none,
 8   },
 9 });
10 Peep::AddField(body: String {
11     read: public,
12     write: p -> [p.author],},
13 p -> "Peep by " + User::ById(p.author).name);
```

This migration first creates a `Peep` model containing an author (lines 1–9), then adds a peep body (lines 11–14). Line 14 specifies that all peeps receive a default body that states the author's name. This is *required* in Scooter: when developers add a field to a model, they must provide a function that populates that field with an initial value.

Migrations can also remove fields and models as long as other policy functions do not depend on them. For example, the following would fail, because the `body` policy above refers to author:

```
  Peep::RemoveField(author);
```

On the other hand, the following would work, because no policies (other than those within Peep) depend on Peep:

```
  DeleteModel(Peep);
```

Policy Changes. In addition to schema updates, Scooter migrations can express policy updates. For example, a developer may want to update the create policy on Peep:

```
  Peep::UpdatePolicy(create, p -> [p.author]);
```

This migration replaces the previous policy (public) with a new policy function that only allows users to create a peep when they are the author; previously, anyone could create a peep with any author. The UpdatePolicy command indicates the developer's intent is to provide a policy that is at least as strict as the old policy; the verifier will prove that this is true before Scooter executes the migration. If developers need to weaken a policy, they can use WeakenPolicy, and the verifier won't check for strictness preservation. Finally, developers can also strengthen and weaken field policies with UpdateFieldPolicy and WeakenFieldPolicy.

Principal Changes. Scooter migrations can also change the set of principals using AddPrincipal, RemovePrincipal, AddStaticPrincipal, and RemoveStaticPrincipal, which have no effect on the underlying schema. The verifier will stop developers from removing any principal that is used in policy functions.

Verifying Migrations Scooter verifies the safety of an entire migration before it executes any part of it, which obviates rolling back migrations partway through because of errors. The main challenge for verification is that the correctness of one migration command depends on its predecessors. For example, consider the following migration, which creates a `User` model and then adds a `bestFriend` field and a `secret` field that is shared between a user and their best friend.

```
 1
 2 CreateModel(User {
 3   create: public,
 4   delete: u -> [u.id],
 5 });
 6 User::AddField(bestFriend: Id(User) {
 7   read: public,
 8   write: u -> [u.id],
 9 }, u -> u.id);
10 User::AddField(secret: String {
11   read: u -> [u.id, u.bestFriend],
12   write: u -> [u.id],
13 }, _ -> "my_secret");
14
```

The `User::AddField` commands are only valid after the `User` model has been created (using `CreateModel` on line 1). If lines 1–4 were omitted, Scooter would reject the migration because of a missing `User` model. Likewise, the `read` policy for `secret` does not typecheck unless the `bestFriend` field has already been added to `User`.

To address the fact that each migration command's safety depends on prior commands, Scooter maintains an in-memory representation of models models that it uses to typecheck and verify each command. Once Scooter has verified a command, it records the command's effect on the set of models and continues on to verify the next command until the migration is complete or it hits an error. Scooter does not actually manipulate data during this process, so in case of an error, Scooter doesn't need to rollback database state. When Scooter has verified the migration completely, it executes the migration against the database and writes the in-memory policy to the Scooter_p file.

2.3.3 The Scooter ORM

Following a successful migration, the Scooter compiler generates an ORM implementation in Rust for each model. The ORM enforces policies dynamically before performing database queries. Like most ORMs, our ORM is agnostic to the underlying database system and relies on a *driver* to communicate with an actual database; we implement and evaluate a MongoDB driver. Since our ORM is largely standard, we only describe the Scooter-specific details.

Acting on Behalf of Principals Before querying an ORM model, developers must declare a principal with which to perform the query. For example:

```
// set up the db connection
let db_conn = // ...
// declare the principal
let princ = db_conn.as_princ(Unauthenticated);
// query the database
let u = User::find_by_id(princ, some_user_id);
```

Web applications rarely require manual principal management, though: typically, middleware automatically selects a principal based on the signed-in user (e.g., instead of the `Unauthenticated` principal on line 5).

Handling Overly Sensitive Fields Queries to the database return *partial objects*; the ORM removes fields that the principal does not have read access to. In turn, developers must handle fields whose values are missing due to policy enforcement. For example, in this code snippet, they must account for a missing `email` field:

```
match u.email {
  // principal has read permissions:
  Some(email) => println!("Success");
  // principal does not have read permissions:
  None => println!("Failure");
}
```

This forces developers to explicitly consider permissions. In practice they need to do this already, for example, when implementing views.

Handling Policy Failures When writing to the database, the ORM checks the relevant `create` or `update` policies and returns an error if necessary. For example, in this snippet, the ORM code accounts for an attempted edit that could fail because `princ` does not have the proper edit permissions:

```
match edited_user.save(princ) {
  Ok(_) => println!("Save successful")
  Err(_) => println!("Save failed"),
}
```

These errors force developers to respond to both policy and database failures. In development mode they can respond with the exact access violation; in production mode, they can simply return an HTTP 403 Forbidden response.

2.4 Verifying Policy Updates in SMT

The core of our Sidecar verifier is centered around proving that one policy is at least as strict as another, a property we call *strictness*. This *strictness* property not only allows Sidecar to verify `Update` commands, but also allows it to prevent data leaks. In this this section, we first formalize the strictness safety property and describe how Sidecar translates this property into SMT formulas, allowing it to verify `Update` commands using an off-the-shelf SMT solver—specifically, Z3 [99]. Finally, we show how Sidecar uses strictness to detect leaks.

Strictness Property Recall that a Scooter policy (for a given operation) is a function p that takes an instance, i.e., an object, and returns the set of principals who are allowed to perform that operation. Because policies can query the database, p must also take the database as a parameter. Formally, it is safe to strengthen policy p_1 to policy p_2 iff the following strictness property holds:

$$\forall db, \forall i \, . \, p_1(db,i) \supseteq p_2(db,i) \tag{2.1}$$

That is, for all databases and for all instances (objects) in those databases, p_2 must produce a subset of the principals returned by p_1.

For each migration of a policy from p_1 to p_2, Sidecar checks the migration's safety by translating this formula into an SMT query. Unfortunately, a direct translation of this formula to SMT leads to many different problems, some related to performance and some related to counterexample generation (which requires the solver to generate a full database). To sidestep these issues, Sidecar translates policies into set-free SMT queries. We describe our translation to

SMT next.

Leakage Formula SMT solvers verify a property by proving that its negation is unsatisfiable. When we negate the strictness property we get the core of our SMT query:

$$\exists db,\ i,\ u.\ u \in p_2(db,i) \wedge \neg(u \in p_1(db,i)) \tag{2.2}$$

That is, there exists some database *db*, instance *i*, and principal *u*, such that *u* is permitted by p_2 and not by p_1. While we can express this in SMT directly (using the theory of arrays [176]), our translation eliminates sets and set comparison. First, it translates set fields to an equivalent join-table representation. Next, Sidecar distributes the $\in$ operator across all expressions, to eliminate all remaining set-typed expressions and variables. We describe these next.

Translating Set Fields The Scooter language allows users to define fields that contain sets. For example, a single user on a social media site may have many followers, which programmers express in Scooter as follows:

```
User { ... friends: Set(Id(User)) { ... } }
```

Standard ORM practice translates the above into a join table. We adopt a similar approach—we encode the user-friend relation explicitly by adding the following model of the relation:

```
UserFriends { from: Id(User) {...},
              to:   Id(User) {...} }
```

Using this encoding, `friends` field access can be translated into an appropriate query on the `UserFriends` table. The Sidecar verifier performs this translation at the language level, before translating to SMT. For example, it translates the expression `user.friends` into:

```
UserFriends::Find({from: u}).map(uf -> uf.to)
```

Translating Set Expressions Once set fields are removed, Sidecar rewrites the leakage condition (2.2) into an equivalent formula without sets. Sidecar does this by distributing the $\in$ operator across Scooter expressions. In most cases this is straightforward. For example, $u \in (e_1 + e_2) \rightsquigarrow (u \in e_1) \vee (u \in e_2)$. The two exceptions are `map`, `flat_map`, and `Find`.

When Sidecar distributes $\in$ across `map` it introduces an existential:

$$u \in e_1\,\texttt{.map}(x \to e_2) \rightsquigarrow \exists v.\ v \in e_1 \wedge u = e_2[v/x]$$

Similarly for `flat_map`:

$$u \in e_1\,\texttt{.flat\_map}(x \to e_2) \rightsquigarrow \exists v.\ v \in e_1 \wedge u \in e_2[v/x]$$

Because all instances used in policies are in the database, when translating $u \in M\texttt{::Find(\{...\})}$, we can simply check if u meets the criteria of the `Find` query:

$$u \in M\texttt{::Find(\{}\ldots\ f_i\ op_i e_i\ \ldots\texttt{\})} \rightsquigarrow \bigwedge_i (u.f_i\ op_i\ e_i)$$

This translation eliminates all remaining set expressions and variables from the leakage formula. We give the complete definition of $\rightsquigarrow$, as well as proofs of correctness and set elimination, in the appendix of the original Scooter paper.

Translating Instances and IDs Sidecar translates instances to SMT by encoding each field as a function, much like Nijjar et. al [191]. For example, Sidecar translates the declaration `email: String` inside `User` into a function `email` : `User` $\to$ `String`. So, `u.email` in Scooter is translated to `(email u)` in SMT. Instances like `u` in our SMT encoding are uninterpreted values (which can only be used as parameters to field functions like the `email` function above). This encoding also allows us to easily encode id-uniqueness. In particular, instead of asserting uniqueness as $\forall o_1, o_2 \,.\, o_1.id = o_2.id \Rightarrow o_1 = o_2$, we define an `id` function (which represents the `id` field) to return the instance itself, i.e., we define `(id i)` to return `i`. This avoids expensive quantifiers and reduces $o_1.id = o_2.id$ to $o_1 = o_2$.

Translating Primitives In addition to `Bool` which is trivially represented in SMT, Scooter supports integers (`I64`), doubles (`F64`), `Option` types, and `DateTime`. `I64` and `F64` are naturally represented as bitvectors whose operations are encodable in first-order logic and are built-in to Z3. We encode `Option`s using SMT-LIB's datatype declarations. `DateTime` requires special handling. We encode `DateTime`s as UNIX timestamps (bitvectors in SMT). This allows Scooter to use integer bitvector comparison to implement `DateTime` comparison. Sidecar models the `now()` constructor as an unconstrained bitvector. When comparing two policies, Scooter assumes they are invoked at the same time and thus uses the same unconstrained value for all occurrences of `now()`.

Detecting Data Leaks Sidecar uses the policy strictness check, combined with dataflow analysis, to detect data leaks. We say data leaks when, during migration, data flows from a more restrictive field to a more permissive field. For example, this (leaky) migration moves data from the private email field to a public bio:

```
CreateModel(User {
  create: public,
  delete: u -> [u.id],

  email: String {
    read: u -> [u.id],
    write: u -> [u.id],
  }
});
AddField(bio: String {
  read: public,
  write: u -> [u.id],
}, u -> u.email);
```

Before the migration, a user's email was only visible to the user; afterwards, everyone can read the email since it is used to initialize the bio field.

We detect leaks in two steps. First, we use a simple static *dataflow* analysis on the Scooter language to detect which fields flow to which other fields during the migration. Second, for each field f_1 that flows to a field f_2 we check that the policy for f_2 is at least as strict as the policy for f_1.

Using Prior Definitions Sometimes the correctness of a policy migration relies on the schema migrations that preceded it. To reiterate the example from Section 2.2.1:

```
User::AddField(
  adminLevel : I64 {
    read: _ -> User::Find({adminLevel: 2})
    write: _ -> User::Find({adminLevel: 2})
  }, u -> if u.isAdmin then 2 else 0);
User::UpdateFieldWritePolicy(bio,
 u -> [u] + User::Find({adminLevel: 2}));
```

While Sidecar normally encodes fields as uninterpreted functions in SMT, in this example Sidecar proves this migration safe by using a *prior definition*. Specifically, the initialization function used for `adminLevel` defines the relationship between `adminLevel` and `isAdmin`—so Sidecar knows that a user has an `adminLevel` of 2 if and only if `isAdmin` is true. Using prior definitions is necessary for verifying certain migrations, but can also have surprising semantics (§2.6.4).

2.5 Evaluation

We evaluate our Scooter language, our Sidecar verifier, and our policy-enforcing Rust ORM by answering the following questions:

1. Can the Scooter language express common policies and migrations (§2.5.1)?
2. Can Sidecar detect unsafe migrations? (§2.5.2)
3. Is the Sidecar verifier performant enough to use regularly (§2.5.3)?
4. Is the overhead of the generated ORM in line with existing policy enforcement techniques (§2.5.4)?

To answer these questions we port case studies from existing policy frameworks to Scooter: one from LWeb [203], three from Hails [119], one from Lifty [205], and one from UrFlow [86]. We use MPVC case studies because they contain explicit policies. None of these case studies provide migrations, however, so we reconstruct them, when possible, from git histories. In addition, we port a production Ruby on Rails application—and its migrations—to Scooter. Rails does not provide a policy language, so we reverse engineer policies from the behavior of the application.

Results Summary We find that Scooter is capable of expressing the vast majority of policies and migrations showcased by existing frameworks—confirming that Scooter's verification-oriented design decisions don't unduly limit the expressiveness of the language. We find that Sidecar can detect unsafe migrations (and generate counterexamples) from real applications. Furthermore, we find that verification with Sidecar takes less than a second to complete and that our ORM imposes a runtime overhead (11%) comparable to the LWeb [203] and Hails [118] MPVC frameworks.

Project	Framework	# Models	# Fields	# Migr	Migr LOC	Unique Policies	Migration Actions
BIBIFI	LWeb	46	215	11	183	4	37/37
Visit Days	Rails	4	19	10	139	7	21/21
GitStar	Hails	3	8	1	11	7	6
LambdaChair	Hails	4	8	1	38	5	2/2
Learn-by-Hacking	Hails	3	13	5	63	7	22/23
Ur-Calendar	UrFlow	2	8	1	52	6	1/1
Lifty Conference	Lifty	6	26	1	175	10	1/1

Figure 2.5: A list of case studies, along with metrics. # Models is the number of models in the final policy; # Fields is the number of fields on all models in the final policy; # Migr is the number of migrations considered; Migr LOC is the total lines of code of migrations expressed in Scooter; Unique Policies indicates the count of unique policy functions that were ported to Scooter; Migration Actions indicates the ratio of migration actions that were expressible in Scooter.

Experimental Setup We conduct all performance measurements on an Arch Linux (kernel 5.11.9) desktop with an Intel i7 6700K processor, 4 cores (8 hyperthreads) at 4GHz, and 16 GB of RAM.

2.5.1 Scooter Language Expressiveness

To answer this question we port case studies from existing policy frameworks to Scooter and discuss the capabilities and limitations of the language. For each case study, we report the number of migrations, lines of migration code, the number of policies successfully ported, and the number of migration actions used. Our results—reported in Figure 2.5—show that Scooter is able to represent all policies and all but one migration. In the remainder of this section, we discuss each case study and the process of porting them to Scooter.

Build it Break it Fix it We port the LWeb BIBIFI production web application designed to manage and coordinate security programming contests [252]. The application allows administrators to create contests and posts related to those contest and manage the teams and scores of contests, while regular users can log in and see the current contests and their team's details. LWeb developers write policies on fields of a record as disjunctions of static principals and (other) fields of the record. BIBIFI uses automatic schema migrations to (1) remove fields or (2) add fields with default values. Scooter is able to express all the BIBIFI data models, policies, and migrations.

Visit Day We port a production Ruby on Rails app designed to schedule meetings between visiting prospective PhD students and faculty [215]. The application allows users to create privileged accounts for scheduling meetings, as well as unprivileged accounts so that students and faculty can view their schedules; users can reset their passwords and invite other users. We port both the application and its ActiveRecord [112] migration scripts with no issues. There are ten migrations, totaling 139 lines of code. The original hand-written policy is 25 lines of code, but after all the migrations, it becomes an automatically generated 103-line policy file.

GitStar We port the GitStar [253] benchmark—a lightweight GitHub-like application—from the Hails project [119]. We make one modification. Hails repositories have a `reader` field that can be a set of user ids or a special `public` value; since Scooter does not include arbitrary sum types, we instead encode this as two fields, a boolean `is_public` and a set of user `readers`.

LambdaChair LambdaChair [254], a lightweight conference review system, is another Hails benchmark. It features Program Committee (PC) users, as well as non-PC users, both of whom can be paper authors. It also has a `root` principal that can edit any paper. The authors of LambdaChair evolved the LambdaChair application over time: first, they created authors and PC members, and then, through a migration, they added papers and permissions on those papers. However, they did so in ad-hoc way, by changing the policy and the model by hand. Scooter is able to express the LambdaChair migrations in thirty-eight lines of code, and Sidecar can quickly verify those migrations for safety.

Learn-by-Hacking We port another Hails benchmark, Learn-by-Hacking [255], that lets users incorporate code into tutorials, blog posts, and more. The original authors evolved Learn-by-Hacking through five migrations (e.g., one adds tags (short categories) to associate with posts). Using our DSL alone, we are unable to express one migration using Scooter—that adds a database of existing tags—since this migration relies on querying and then dynamically creating objects. As we further discuss in Section 2.6.2, this migration can be implemented using the Scooter ORM.

UrFlow Calendar We port UrFlow Calendar [85], an application that allows users to manage a calendar, from the UrFlow project [86] without any issues. Ur encodes policies very differently from Scooter—as a SQL-based eDSL—but Scooter is still able to express this benchmark's policies despite the difference.

Lifty Conference We port Lifty Conference [256], another conference review system, from the Lifty project [205]. This benchmark is different from the others because Lifty is not an ORM; the benchmark operates on in-language values and types. The Lifty policies rely on a singleton which we translated into a database object. Scooter is able to express all policies from the Lifty benchmark.

2.5.2 Detecting Unsafe Migrations

Since none of the above case studies had unsafe schema or policy changes, we ensure that Sidecar can detect unsafe changes by implementing several unsafe schema and policy changes. First, our test suite contains multiple negative tests, including the unsafe Chitter application migrations described in Section 3.2. Second, we model two unsafe migrations from two applications: (1) a refactor of HotCRP's policy code that inadvertently granted unauthenticated users administrator rights [159, 158]; and (2) a policy change in the Hails Task management app that inadvertenly made projects readable to all users [232]. In all cases, Scooter sucesfully detects the unsafe migrations and generates counterexamples.

2.5.3 Sidecar Verification Speed

We evaluate the performance of Sidecar by timing the migrations from all of the case studies from Figure 2.5. The most expensive migration takes 88.8ms to verify; the fastest takes 10.3ms. Performing the safety checks on a given command, say `AddField`, takes 7.1–12.7ms.

2.5.4 ORM Performance Overhead

We measure the performance overhead of our ORM on two benchmarks—an end-to-end macro-benchmark and a micro-benchmark.

Macro-Benchmark To understand the overhead our ORM imposes on real web applications, we port two BIBI controller benchmarks from LWeb [203] and measure the policy enforcement overhead on latency for each endpoint. Specifically, we port the `/announcements` route, which fetches announcements and schedules, and the `/profile` route, which fetches the logged-in user's profile. We use Scooter with the Rocket web framework (version 0.4) and Handlebar template system (version 1.1). To measure latency, we use the Apache benchmarking tool `ab`; we configure `ab` to make 10,000 requests with 16 concurrent connections. We find that the overhead on mean and tail latency, which we measure to be 4ms for both end points, is in the noise (< 0.1ms). This is not surprising: BIBIFI policies only allow field accesses and static principals—and thus checking whether a field can be accessed is answered by an equality check on already available data. Unlike Scooter, the overhead of enforcement in LWeb for these endpoints is 2.41–19.01%; this is likely because LWeb uses IFC, whereas Scooter only performs access-control checks at the database boundary.

Task	Create Post	View Friend Posts
Unchecked	0.313 ms	13.8 ms
Hand checked	0.334 ms	14.9 ms
Scooter checked	0.331 ms	15.2 ms

Figure 2.6: The time taken for two application level tasks, in the three configurations: unchecked, when manually checked, and when checked with our Rust ORM.

Micro-Benchmark Because BIBIFI's policies are not representative of the more complex policies supported by Scooter—policies that require database queries—we implement a micro-benchmark around the Chitter policy from Section 3.2. Specifically, we measure the ORM performance overhead by timing two different actions: (a) creating a Chitter post and (b) viewing a list of friends' posts. We do so in three configurations: (1) Unchecked: native database bindings with no policy checks; (2) Hand checked: manually written policy checks; and (3) Scooter checked: Scooter's ORM enforcement. We measure each action 10,000 times in each configuration and report the mean. As shown in Figure 2.6, our ORM is only slightly slower than manually inserted checks. In real web applications, these overheads are hidden by network latency and other application components.

2.6 Discussion and Limitations

Like all languages and verifiers both Scooter and Sidecar have limitations. We discuss some of these next.

2.6.1 Expressiveness

Scooter tries to balance the need to express complex behavior with the need to be a reliable tool. Though we originally designed Scooter strictness checking to be decidable, we added language features—namely, set subtraction and cyclical models—which can be used to write policies that the SMT solver may not be able to solve. These features are useful for expressing certain data models and policies. For example set subtraction is necessary for expressing deny lists (in an open-world system where we cannot enumerate all principals). Though these features could cause Sidecar to time out (and thus require the developer to perform an unchecked migration), our verifier did not time out on any of the case studies or our tests.

There are many opportunities for additions to Scooter that increase expressivity without affecting decidability. For example, Scooter currently only support data types that can be used in policies. We envision extending Scooter with common datatypes (e.g., binary blobs, JSON objects, and geolocations) that cannot be referenced in polices—and thus Sidecar does not have to reason about—but are useful for many applications. Similarly, we envision extending Scooter with anonymous records that would, for example, allow developers to express projections of database objects.

2.6.2 Data Migrations

Scooter does not support manipulating data in the database apart from the populating function in `AddField`. For example, developers can't create or edit objects in a migration script. Real-world migrations sometimes need to do this, though—and indeed we encountered this when porting Learn-by-Hacking to Scooter.

Verifying the safety of these operations is difficult because they can cause *indirect leakage*. For example, in a social media site where only a user's friends can see their email, a migration that creates a new friendship between two users also leaks their emails. In contrast to the kind of leakage Sidecar prevents, this leakage does not require sensitive data to flow to a permissive output. Instead, it modifies a policy-relevant field such that the permissions represented by an existing policy function are expanded.

Developers can work around this limitation by using the ORM to perform migrations at the application level, as a series of database queries. This ensures that all database accesses are protected by policies. In the rare case where developers need to elide policy enforcement, our

ORM, in debug mode, also allows developers to temporarily turn-off enforcement.

2.6.3 Transactions

Scooter has no transactional semantics; any sequence of read, write, create, and delete actions must be valid at each step. This can create problems when multiple database mutations must happen together.

For example, in the LambdaChair case study, papers have multiple authors, and these authors (and only these authors) have permissions to add other authors. Without set fields, this would require three models: `Paper`, `User`, and `PaperAuthor` which represents the join table between them. When a user creates a paper, they need to create an instance of both `Paper` and `PaperAuthor`. If they create the `Paper` first, they would not be an author and thus could not add other authors. Conversely, if they try to create the join table entry first, there is no `Paper` to reference. In this specific case, we sidestep the problem by making authors a set field. Creating an instance with a set field is (to Scooter) a single action even though it maps to multiple database actions. In this way, set fields allow a specific type of transaction.

It is unclear how (or if) Scooter and Sidecar's approach to policy safety scales up to arbitrary database transactions, where multiple operations can occur atomically. We consider this future work.

2.6.4 Surprising Semantics

As described in Section 2.4, Sidecar tracks data equivalences during migrations. This allows policy-relevant fields to change representation—like `isAdmin` becoming `adminLevel`—and still pass verification. While this is a useful feature, it has subtle behavior that can produce surprising results.

The first surprise is that a sequence of migrations can be valid when it is in one file and invalid when it is split across two. Migration scripts are atomic, and thus Sidecar tracks equivalences across migration commands within a script file. But because writing to the database can invalidate equivalences, we do not track equivalences between scripts.

Which policies Sidecar considers equal, due to equivalences, may also be surprising. Consider the migration from `isAdmin` to `adminLevel` from Section 3.2, where admins are given `adminLevel` 2 and regular users are given `adminLevel` 0. Until `isAdmin` is removed from the model, Sidecar will track the relationship between the two fields. In doing so, Sidecar will deduce that:

```
    u -> User::Find({isAdmin: true})
```

is equivalent to:

```
    u -> User::Find({adminLevel: 2})
```

and, more surprisingly, also equivalent to:

```
    u -> User::Find({adminLevel >= 1})
```

By tracking equivalence, Sidecar knows that—at the time of the migration—there are no users with an `adminLevel` 1 and that all users with `adminLevel` 2 were admins.

This could have unintended effects. Until `isAdmin` is removed, for example, we can use {`adminLevel >= 1`} instead of {`adminLevel: 2`} without Sidecar raising an alert—even if the semantic meanings of the two policies differ: one includes moderators, the other does not. Since equivalences let Sidecar succeed in many cases users expect it to and prevent them from relying on unchecked policy relaxations, we think this is a worthwhile trade-off. Of course, developers can turn equivalence tracking off; without them, Sidecar can still catch policy weakenings and leaks due to data flows but cannot incorporate knowledge from earlier migration steps.

2.7 Related Work

There is a long and rich history of work on the topic of security for database-backed application, and more broadly for general purpose applications. Broadly speaking our paper distinguishes itself from prior work in this space by offering a specific kind of enforcement that had not been previously explored: *static* verification of *data migrations* and *policy migrations* in database-backed applications. We see our effort as complimentary to the existing literature on secure database-backed applications. We now highlight the most closely related work in this area.

Dynamic Enforcement Dynamic enforcement of security policies is a well-explored idea, including the seminal work on Execution Monitoring [229], hardware-level information-flow tracking [245], language-based information-flow control [137, 246, 241], fine-grain discretionary access-control for databases [91, 247, 177, 281] and new programming models that incorporate dynamic enforcement [277, 41].

The dynamic approaches most closely related to our work are those developed for database-backed web frameworks (e.g., Jacqueline [276], Hails [119, 118], LWeb [203], and IFDB [230]). These frameworks allow the programmer to state policies separately from code, and the system automatically enforces the policy dynamically. While some of these systems enforce policies in a mandatory fashion (e.g., using IFC), Scooter only enforces access control policies at the ORM level, and thus our security guarantees do not extend to leaky application code. Since the ORM code is generated, though, Scooter could be extended to, say, generate ORM code that uses an IFC system for enforcement. Since our core focus is on verifying migrations, which none of these systems address, we see our work as largely tackling a complementary problem.

Static Enforcement Another approach is to enforce policies statically, before the program runs. For example, there is a long line of work on static type systems that enforce fine-grained policies (e.g., information flow control and fine-grain access control) in various languages [185, 171, 30, 75, 171, 30, 209, 79, 260, 129, 168, 224, 144, 225, 40, 205, 236]. The static approach most closely related to our work is UrFlow [86], a tool for static analysis of database-backed applications. UrFlow takes data policies specified as SQL queries, and statically ensures that the application code adheres to those policies. The Scooter ORM enforces policies dynamically and thus introduces runtime overheads that can be partially avoided with static checking. Conversely, neither UrFlow nor these other systems account for ways in which the applications (and, specifically, underlying database schema and policies) might change. Extending Scooter to systems like UrFlow that enforce policies statically is an interesting future direction.

Program Partitioning Yet another approach to security enforcement is to use some kind of program partitioning, thus enforcing certain security properties by construction. For example, the Diesel [107] web framework for writing database-backed applications restricts each module of the application to use a database connection that can only access data in the corresponding database module. While this prevents certain kinds of cross-module leaks, it does not directly prevent secret leakage to an unprivileged user. Program partitioning can also be done automatically to enforce a stated policy, as is done in the Swift [87] system and the Jit/split [283] compiler. However, again, none of these approaches address data or policy migrations.

Schema Migration There is a long history of work on schema migration. The most commonly explored topics include mechanisms for expressing schema mappings [211, 160], techniques for automatically inferring schema mappings [181, 34, 120, 30, 266], and support for automatically evolving and verifying queries and databases in the face of schema migration [97, 96, 265, 266]. None of this work addresses how security policies interact with migrations, which is the aim of our work. We, conversely, don't automatically update or verify the application code that uses the Scooter ORM.

Acknowledgments

We thank the reviewers, and our shepherd Adam Chlipala for his in-depth and insightful feedback on all aspects of the paper—and for finding a bug in one of our proofs. We thank Ranjit Jhala for his always insightful guidance. This work was supported in part by gifts from Cisco; by the NSF under Grant Numbers CCF-1918573, CCF-1955457, CNS-1514435, and CAREER CNS-2048262; and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, Deian Stefan. ACM, 2020. The dissertation author was a primary investigator and author of this paper.

Chapter 3

Towards a Verified Range Analysis for JavaScript JITs

We present VeRA, a system for verifying the *range analysis* pass in browser just-in-time (JIT) compilers. Browser developers write range analysis routines in a subset of C++, and verification developers write infrastructure to verify custom analysis properties. Then, VeRA automatically verifies the range analysis routines, which browser developers can integrate directly into the JIT. We use VeRA to translate and verify Firefox range analysis routines, and it detects a new, confirmed bug that has existed in the browser for six years.

3.1 Introduction

On May 30, 2019, employees of the cryptocurrency startup Coinbase were targeted by a phishing campaign that lured them to visit a Web page hosting attack code [175]. This code exploited previously unknown bugs in Firefox to take over the victim's machine; the first bug arose from incorrect type deduction in the *just-in-time* (JIT) compiler component of Firefox's JavaScript engine [15].

In 2020, Google's Threat Analysis Group identified websites, apparently aimed at people "born in a certain geographic region" and "part of a certain ethnic group," that would install a malicious spyware implant on any iPhone used to visit them. Two bugs exploited in this campaign, according to analysis by Google's Project Zero [55, 127], were in the JIT component of Safari's JavaScript engine [43, 5].

The JavaScript JITs shipped in modern browsers are mature, sophisticated systems developed by compilers experts. Yet bugs in JIT compilers have emerged in recent months as *the single largest threat to Web platform security*, and the most dangerous attack surface of Web-connected devices.

Unlike other compilers, browser JITs are exposed to adversarial program input. Remote attackers can craft JavaScript that will trigger JIT compiler bugs and break out of the sandbox on victim users' machines. These attacks are possible in spite of the fact that JavaScript is a memory-safe language, because its safety and isolation guarantees only apply if they are correctly maintained by JIT compilation.

This is easier said than done. Consider JavaScript arrays, which are maps from index positions to values. These maps can be "sparse," in the sense that indices can be missing in the middle. Any out-of-bounds accesses must be checked to return the special value `undefined`. Furthermore, values stored in an array can be of any type, so array elements must be tagged or boxed. In a naïve implementation, numerical kernel performance would be unacceptably slow.

JavaScript JITs speculatively optimize accesses to dense arrays and to arrays whose elements are all the same type, with bailouts to a slow path should the array shape change. And they perform *range analysis* on values that could be used as array indices to facilitate bounds-check elimination. When range analysis confirms that the array indices are guaranteed to be within a given range, the JIT compiler generates code that allocates the array sequentially in memory and allows sequential access without any bounds checks.

These optimizations are crucial—but also risky. Failing to bail out of the speculative fast path when an array's shape changes leads to type confusion; incorrectly eliminating a bounds check allows out-of-bounds memory accesses. Both bug types can be exploited for arbitrary code execution [122].

The implications of JavaScript JIT bugs for security were recognized as early as 2011 [221]; attackers have turned to JIT bugs as other avenues for browser exploitation become more rare and more difficult to exploit (see, e.g., [279]). Since late 2017, industrial security researchers have uncovered a dozen or more bugs in the JIT compilers shipping with Chrome (e.g., [22, 23, 19]), Firefox (e.g., [3, 18, 20, 15]), Safari (e.g., [43, 5, 16, 17, 21]), and Edge (e.g., [12, 13, 14]). They have documented JIT compiler internals and developed generic techniques for exploiting JIT compiler bugs for code execution in blog posts [263, 287, 67, 110, 239, 109, 223, 213], in industrial security conference presentations [123, 258, 152, 222, 126, 108], and in industrial journals such as *Phrack* [128]. And, as we noted above, JIT bugs are being weaponized against real users.

In short, the status quo is untenable. The JIT's adversarial setting means even obscure bugs that no normal program would *ever* hit become exploitable attack vectors. To secure the Web, it is necessary to build and deploy JIT compilers free of security bugs.

In this chapter, we explore, for the first time, the possibility of using *compiler verification* to secure browser JavaScript JIT compilers. Verification proves that the compiler is correct using formal methods that consider all possible corner cases.

There is, of course, much prior work on compiler verification (§3.8). But JavaScript JITs are a new and challenging domain for verification. JavaScript is an unusual language with complicated semantics; the ARMv8.3-A architecture revision even adds an instruction, `FJCVTZS`, to support floating-point conversion with JavaScript semantics [1]. And because browser JITs are supposed to improve the perceived runtime performance of Web programs, compilation time is a cost. A JIT that is verified but slow will not be acceptable.

As a first step in this direction, we build a system, VeRA, for expressing and verifying

JavaScript JIT range analyses. VeRA supports a subset of C++ for expressing range analysis routines, and a Satisfiability Modulo Theories [47] (SMT)-based verification tool that proves the correctness of these analyses with respect to JavaScript semantics. Browser developers can write their range analysis code using VeRA C++, prove it correct, and incorporate the verified code directly into their browser.

Compared to prior work in compiler verification, VeRA distinguishes itself by handling the details of a realistic range analysis: we use VeRA to express and verify the range analysis used *in practice by the Firefox browser*. This requires handling many challenges: complicated corner cases of JavaScript semantics; complex dataflow facts whose semantics are often disjunctive predicates; and complex propagation rules for those dataflow facts. Our verification uncovered a new Firefox range analysis bug and confirmed an old bug from a previous version of the browser. We also find that our verified routines work correctly in Firefox—they pass all 140 thousand Firefox JIT tests—and perform comparably to the original routines in both micro and macro benchmarks.

This chapter, later revisits the original VeRA work by reimplimenting the range analysis routines using a generic automated verification language, showing that VeRA's results do not particlarly benefit from a specialized solver. The reimplemenation includes two performance optimization which allow for verification of routines that VeRA timed out on. Furthermore, we demonstrate the limits of SMT-based verification of range analysis by showing that even naive analysis of floating point addition times out after a week.

3.2 Overview

This section gives an overview of range analysis in JIT compilers and the ramifications of range analysis bugs. Then, it walks through using VeRA to verify a piece of the Firefox JIT's range analysis logic.

3.2.1 Range Analysis in JITs

Range analysis is a dataflow analysis that compilers use to deduce the range—typically upper and lower bounds—of values at different points in program execution. These range deductions, or value constraints, are then used by different optimization passes to generate efficient code. For example, Firefox's dead code elimination (DCE) pass eliminates blocks guarded by contradictory constraints [6]:

```
if (x > 0)
  if (x < 0) /* ... dead code... */
```

It also eliminates redundant checks—in the redundant check elimination (RCE) pass—when it can prove that values are within a certain range [8]:

```
var uintArray = new Uint8Array(...); // buffer

function foo(value) {
  if (value >= 0) { // always true; redundant check
    return value;
  } else { /* ... dead code ... */ }
}

for(let i = 0; i < uintArray.length; i++) {
  foo(uintArray[idx]); // call foo with unsigned value
}
```

Here, the comparison in `foo` is always true—`uintArray` can only contain unsigned integers—and thus can be eliminated (along with the `else` branch).

More significantly, Firefox relies on range analysis to move and eliminate internal JavaScript array bounds checks [7]. Since JavaScript is a memory safe language, the compiler inserts bounds checks around every array access. For instance, in the example above, the array indexing operation `uintArray[idx]` internally performs a bounds check, which returns `undefined` if the access is out of bounds, i.e., when `idx < 0` or when `idx >= uintArray.length`. In practice, this incurs overhead—real applications make heavy use of arrays—and JITs aggressively try to eliminate bounds checks. In the example above, for instance, Firefox can prove that the array accesses are in-bounds and eliminate *all* the internal bounds checks.

3.2.2 From Range Analysis Bugs to Browser Exploits

Bugs in the range analysis code can, at best, manifest as application correctness errors—and at worst as memory safety vulnerabilities. For example, an incorrect range deduction can cause the JIT to eliminate a bounds check, which can in turn allow an attacker to read and write beyond JavaScript array bounds and hijack the control flow of the browser renderer process (e.g., using JIT-ROP techniques [235, 169]).

Until recently, TurboFan—the JIT compiler component of Chrome's V8 JavaScript engine—used to deduce that `indexOf`, when applied to a string, would return an integer in the range $[-1, \texttt{String::kMaxLength} - 1]$, where `String::kMaxLength` is the longest allowed V8 JavaScript string ($2^{28} - 16$ characters). Unfortunately, the actual V8 implementation of `indexOf` can return `String::kMaxLength`—one more than the range analysis deduced. As the following example (from the original bug report [22]) shows, this bug allowed attacker-supplied JavaScript to create a variable `i` that TurboFan deduced to be 0, but actually held an arbitrary value (in this case, $100,000$):

```
var i = 'A'.repeat(2**28 - 16).indexOf("", 2**28);
i += 16; // real value: i = 2**28, optimizer: i = 2**28-1
i >>= 28; // real value: i = 1, optimizer: i = 0
i *= 100000; // real value: i = 100000, optimizer: i = 0
```

Since TurboFan thought `i` was zero, it would eliminate all bounds checks on `i`—so attackers could use `i` as an index into *any* array in order to access that array out-of-bounds.

```
class Range : public TempObject {
  int32_t lower_;
  int32_t upper_;
  bool hasInt32LowerBound_;
  bool hasInt32UpperBound_;
  // possibly not a whole number
  FractionalPartFlag canHaveFractionalPart_;
  // possibly negative zero
  NegativeZeroFlag canBeNegativeZero_;
  // the maximum exponent needed to represent the number
  // 0-1023, 1024 indicates inf, 65536 indicates NaN or inf
  uint16_t exp_;
}
```

Figure 3.1: Parts of Firefox's range object

```
Range* Range::rsh(TempAllocator& alloc, const Range* lhs, int32_t c) {
  MOZ_ASSERT(lhs->isInt32());
  int32_t shift = c & 0x1f;
  return Range::NewInt32Range(alloc, lhs->lower() >> shift,
  lhs->upper() >> shift);
}
```

Figure 3.2: Firefox's range analysis logic for the right shift operator.

```
range rsh(range& lhs, int32_t c) {
    int32_t shift = c & 0x1f;
    return newInt32Range(lhs.lower >> shift,
    lhs.upper >> shift);
}
```

Figure 3.3: Simplified VeRA implementation of the right shift operator.

3.2.3 Why Range Analysis is Hard to Get Right

The `indexOf` example is not the only case of an exploitable browser vulnerability introduced by a buggy JavaScript range analysis. Similar bugs have been a problem in practice for all major browsers. This is because JavaScript range analysis is hard to get right.

First, it requires reasoning about double-precision floating point numbers. To do so correctly and efficiently, browsers can't just implement ranges as lower-bound and upper-bound pairs; instead, their range objects are necessarily complicated structures. Figure 3.1 shows Firefox's range object, which keeps track of, among other things, integer bounds, special values, and whether the range includes non-integrals. Not only is the data structure itself complex, but there are also subtle invariants that it must maintain.

Second, tracking special floating-point values like NaN, Infinity, -Infinity, and -0.0 is error-prone. For example, until recently, Turbofan's range analysis incorrectly deduced that `Math.expm1`, the JavaScript builtin used to compute $e^x - 1$, must either return a number value or NaN—it didn't account for floating point negative zero. The browser's implementation of `Math.expm1`, applied to -0.0, correctly returned -0.0 [274, 23]—a mismatch that again allowed an attacker to hijack the browser renderer's control flow.

A constellation of other factors make writing range analysis routines even harder. For example, JITs internally distinguish 32-bit integer values from double-precision floats. This is crucial for performance.[1] But this also means that the range analysis must correctly determine whether an output can be within the range of possible 32-bit numbers—for the Firefox range object, whether the fields `hasInt32LowerBound` and `hasInt32UpperBound` should be set. As another example, since JIT speed directly affects users' experience, the range analysis must be *fast*—it must run just in time—and usefully *precise*—it must produce information useful enough to assist other optimization passes (e.g., DCE, RCE, and bounds-check elimination (BCE)). To

[1]This makes it possible for the JIT to efficiently compile array lookups into a direct memory accesses (as opposed to lookups in a hash map) [240]

this end, JIT developers implement range analysis in low-level languages like C++ and eschew verbose, readable code for fast, terse code. They also explore the trade-offs between speed and precision: Firefox, for example, tracks integer ranges precisely—by tracking lower bounds and upper bounds—but approximates wider floating-point ranges by only tracking exponents (in addition to tracking special values).

3.2.4 Using VeRA to Express Range Analysis

VeRA is a subset of C++ for writing verified range analysis routines. If browser developers write their analysis logic in VeRA C++, they can compile it into automatic correctness proofs for custom properties. We ported twenty-two Firefox routines to VeRA C++; Figure 3.2 shows Firefox's implementation of range analysis for the right shift operator, while Figure 3.3 shows the VeRA version. To calculate the range of possible output values for >>, it masks constant `c` with 31, per JavaScript semantics [105]. Then, it shifts the left-hand operand's lower and upper bound by the masked constant. If `rsh` is given an `lhs` range of [10,100] and a `c` constant of 2, it will return a range of [2, 25].

3.2.5 Using VeRA to Verify Range Analysis

VeRA also provides an internal DSL for expressing the semantic meaning of each computed fact. Given a range analysis dataflow fact R, its semantic meaning is a predicate $[\![R]\!]$ over values. For expository purposes, the following is a simplified semantic meaning for ranges over 32-bit integers:

$$[\![R]\!](v) \triangleq R.\texttt{lower} \leq v \leq R.\texttt{upper}$$

The actual semantic meaning we use in practice (described in Section 3.5) is far more complicated since it includes floating point numbers, special values, and implementation-specific

Firefox invariants.

Once the semantic meaning of range facts is defined, the verification condition can be described as follows. We show the case for a binary operator `op`, but a similar approach can be used for other kinds of operators. We use $\texttt{op}_{ra}$ to denote the range analysis flow function for `op` (where "ra" stands for "range analysis"). This flow function, implemented in a subset of C++, takes two range objects and returns a resulting range object. Finally, we use $\texttt{op}_{js}$ to denote the JavaScript semantics of `op`.

The verification condition for `op` is defined as follows (where ranges R_1, R_2, and JavaScript values v_1, v_2, are universally quantified):

$$[\![R_1]\!](v_1) \wedge [\![R_2]\!](v_1) \Rightarrow [\![\texttt{op}_{ra}(R_1, R_2)]\!](\texttt{op}_{js}(v_1, v_2))$$

This condition states that the flow function for `op` "preserves" the semantic meaning of range facts: if the semantic meaning of the incoming range facts holds on certain incoming values to the operator, then the semantic meaning of the output range fact will hold on the value produced by the JavaScript semantics of the operator on those values.

Let's apply this definition to the `rsh` flow function from Figure 3.3. In this case `op` is >> ; $\texttt{op}_{ra}$ is `rsh` ; and $\texttt{op}_{js}$ is $>>_{js}$. While >> is a binary operator, the version of the operator described in Figure 3.3 is the one where the second parameter is a constant, and so we only need to consider range inputs for the first parameter. As result, we have the following verification condition (where range R and JavaScript values v, n are universally quantified):

$$[\![R]\!](v) \Rightarrow [\![\texttt{rsh}(R, n)]\!](v >>_{js} n)$$

Once a verification developer specifies the predicate $[\![R]\!]$, VeRA automatically proves it (using an SMT solver) for each range analysis function. The browser developer need only write the flow functions using a familiar subset of C++; Figure 3.3 is an example of such a flow function.

Feature	**Syntax**
Declarations	
Classes	`class MyClass {}`
Variables/Fields	`uint32_t x = 4;`
Functions/Methods	`uint32_t f() {}`
Statements	
If/Else	`if (expr) {} else {}`
Assignment	`a = b;`
Void Call	`func(a, b);`
Return	`return foo;`
Expressions	
Casting	`(uint16_t) expr`
Member Access	`s->m;`
Function Calls	`func(a,b)`
Numeric Literals	`1234`, `0xffff`, `47.0`
Comparison	`==`, `!=`, `>=`, `>`, `<=`, `<`
Binary Ops	`+`, `-`, `*`, `/`, `&`, `|`, `^`, `>>`, `<<`
Unary Ops	`~`, `!`, `-`

Figure 3.4: C++ constructs that VeRA supports

The above is intended to give a sense for how verification works in VeRA, but the details are described in Section 3.5.

3.3 VeRA C++

In this section, we describe the programming language that browser developers use to implement range analysis flow functions in VeRA. This language is called VeRA C++, and is a subset of C++. As shown in Figure 3.4, various standard C++ features are not included in VeRA C++, the most notable of which may be loops (including recursion). Looping would make the verification much more complicated, because it would require determining loop invariants. In practice, though, omitting looping constructs does not affect VeRA C++'s expressiveness, since flow functions for realistic range analysis don't rely on loops; for example, none of Firefox's analysis functions use loops. Note that even though there is no looping allowed *inside* a flow function, iteration *does* occur in the dataflow analysis algorithm that finds a solution to the flow functions: indeed, that algorithm applies flow functions repeatedly until reaching a fixed point.

To allow programmers to describe the data structures for their range analysis, VeRA supports C++ classes. For example, the VeRA C++ code in Figure 3.3 uses a `range` class, which is a modified version of the `Range` class from the Firefox codebase (Figure 3.1). Finally, since VeRA C++ is a subset of C++, all VeRA C++ range analyses can be directly incorporated into the Firefox codebase once they are automatically verified.

3.4 Translating VeRA C++ to SMT

Before we describe verification conditions (Section 3.5), we show how to translate VeRA C++ programs to SMT. We describe the challenges with C++ to SMT translation, and then describe how we address theses challenges; Section 3.7 explains why we choose this particular approach.

3.4.1 Challenges in Compiling C++ to SMT

One challenge in compiling C++ to SMT is that the theories defined in SMT solvers [48] provide clients with low-level operators like logical right shift, bitwise and, and assignment; they do not provide high-level C++ control-flow constructs like branches and functions. Instead, the compiler must build these constructs out of lower-level SMT primitives.

The next challenge when translating C++ to SMT is that the semantics of SMT types and operators are different from their C++ counterparts. For example, translating the C++ right-shift operator directly to the raw SMT right-shift operator would be incorrect because it does not take into account the subtle interaction of sign bits and the difference between logical and arithmetic shifts. Furthermore, VeRA verification uses the theory of fixed-sized bit-vectors, and bitvectors do not come with a notion of sign; instead, for each operator (e.g., C++'s less-than operator), VeRA

Feature	Operations
Cast	(*type*)
Conditional Ternary	? :
C++ Comparison	==, !=, >=, >, <=, <
C++ Binary Ops	+, -, *, /, &, |, ^, >>, <<
C++ Unary Ops	~, !, -

Figure 3.5: VeRA IR operations

must choose a corresponding SMT operator (e.g., signed or unsigned comparison) depending on the C++ typing context. Some C++ operators work on both integer and floating point types (e.g., +), but this is not the case in SMT. Thus, VeRA must also choose which version of the operator (e.g., floating-point or bitvector addition) to select based on the operator's input types.

One final challenge is that the C++ standard includes undefined behavior—instruction and operand combinations whose behavior is explicitly *not* prescribed by the standard [28]. For example, the compiler is allowed to replace undefined "`x / 0`" with anything. In contrast, the functions defined by the theory of bit-vectors are *total functions*, i.e. there is a well-defined result for all inputs, and they are free from side effects. Thus, a translation from C++ to SMT should not accidentally ascribe SMT's well-definedness to C++.

3.4.2 Overcoming Challenges with an IR

Because of the differences between C++ and SMT, we create an Intermediate Representation (IR) that sits between the two languages. The compilation from C++ to IR takes care of rewriting control flow constructs like branches, function calls, and return values. The translation from IR to SMT takes care of the gap in semantics between the languages' operators and types.

The IR consists of assignment statements in SSA form. The right-hand side of each assignment is an expression tree. Each node in the tree is labeled by an IR operator, and has child nodes that represent the operator's parameters. Figure 3.5 outlines the operators in the IR.

Compiling C++ to IR We compile VeRA C++ into control-flow-free IR using a series of transformations. The transformations must eliminate if statements, return statements, function calls, and method calls, and must alter variable assignments to satisfy SSA (which the IR expects).

The transformation to SSA is standard. To handle control flow, VeRA re-writes if statements into straight-line code using predicated execution. For example,

```
if (c) { x = 1; } else { x = 2; }
```

becomes:

```
x_1 =  c ? 1 : x_0; x_2 = !c ? 2 : x_1;
```

VeRA handles function calls using inlining, which works well because VeRA C++ disallows recursion. Thus, each call to a function gets its *own* set of assignments; calling "`foo(1); foo(2);`" will generate two separate, versioned copies of `foo`. The rewrites for method calls, return values, nested conditionals, and returns from within conditionals are all similar to the above translations.

Compiling IR to SMT Compiling the IR to SMT requires some additional information to be stored in the IR. In particular, each IR node will include the following three pieces of information (in addition to an operator and children):

1. The SMT term generated for this IR node
2. The C++ type of this node (e.g., `int32` or `double`), which is used to generate the correct version of SMT operations
3. An "undef" bit, which indicates whether the node is the result of an operation with undefined behavior. For example, in the statement `x = 4 << -1`, `x`'s undef bit would be set since a left-shift by a negative value is undefined.

This information, computed during the translation from IR to SMT, is critical for generating SMT terms that faithfully replicate the C++ semantics of the original VeRA C++ code.

We compile IR to SMT as follows. First, we translate assignment nodes into equality, which is seamless because the IR is already in SSA form. Second, we translate the expression tree that appears on the right-hand side of each assignment using a bottom-up traversal that simultaneously computes the SMT term, the C++ type, and the "undef" bit (the three pieces of information described above).

As an example, consider a comparison operator. If either argument is unsigned, the compiler generates an unsigned SMT comparison; otherwise, it generates a signed one.

As another example, consider an IR left-shift `a << b`. If `a` is unsigned, the generated SMT term is the logical left-shift of `a` by `b`; the result type is unsigned; and the undef bit is equal to $\texttt{undef}(\texttt{a}) \vee \texttt{undef}(\texttt{b}) \vee \texttt{isNegative}(\texttt{b})$.

If `a` is signed, the generated SMT term is again the logical left-shift of `a` by `b`, and the result type is unsigned. However, the "undef" bit is more complicated, because C++ 14 dictates that `a << b` has undefined behavior if the shift operation discards any bits of `a` that were set. To detect discarded bits in the 32-bit case, we perform the shift in 64-bits (`(int64)a << (int64)b`), and then check if any of the high 32 bits of the result are set. If so, some bit was shifted off the end of `a`, and the operation has undefined behavior. As a result, the "undef" bit is equal to: $\texttt{undef}(a) \vee \texttt{undef}(b) \vee \texttt{isNegative}(b) \vee (a <<_{64} b)[63:32] \neq 0_{32}$.

3.5 Verification

In this section, we describe how VeRA proves range analyses correct; dataflow analysis "correctness" means that the facts concluded by the analysis are correct with respect to the semantics of the program. For range analysis: if the analysis concludes that a certain variable is in a range, the analysis is correct if the variable is *actually* within that range according to the program's semantics. Note that we only consider correctness of range analysis facts, not their precision.

People typically prove dataflow analyses correct through abstract interpretation [93]. In abstract interpretation, we start by setting up a connection between computed dataflow facts and the semantics of the program. In our setting, we do so by defining, for each range fact R that is computed about a variable, a *semantic meaning*—a predicate $[\![R]\!]$ over runtime values. This predicate establishes the connection between the computed dataflow facts and the semantics of the program: we say that a range fact R on a variable x is *correct* at a program point if, for all possible values v that x can take at runtime, we have $[\![R]\!](v)$.

The goal of our verification is to establish that the implemented range analysis is *correct*, meaning that when the dataflow analysis algorithm terminates, all facts it has computed are correct. In abstract interpretation, this is achieved by proving *local preservation* conditions on all flow functions. These conditions can be shown to imply that the entire range dataflow analysis is correct [93]. It is these local preservation conditions that we ask an SMT solver to prove.

Recall that we use $\texttt{op}_{ra}$ to denote the range analysis flow function for an operator $\texttt{op}$ (which for simplicity we assume to be a binary operator). Recall also that we use $\texttt{op}_{js}$ to denote the JavaScript semantics of $\texttt{op}$. As mentioned in Section 3.2, we define the *local preservation* condition for an operator $\texttt{op}$ as:

$$[\![R_1]\!](v_1) \wedge [\![R_2]\!](v_1) \Rightarrow [\![\texttt{op}_{ra}(R_1,R_2)]\!](\texttt{op}_{js}(v_1,v_2))$$

This condition states that the flow function for $\texttt{op}$ "preserves" the semantic meaning of range facts: if the incoming range facts are correct, then the propagated range fact is correct.

In this section, we first define the semantic meaning $[\![R]\!]$ of a given range fact R. Then, we discuss how VeRA translates $\texttt{op}_{js}$ to SMT (since Section 3.4 explains translation of $\texttt{op}_{ra}$), and talk about how proofs work in practice.

$\texttt{inRange}(R,v) \triangleq$

$$
\begin{aligned}
& R.\texttt{exp} < \texttt{e\_INF} \implies \neg\texttt{isInf}(v) && \text{(R1)}\\
\wedge & R.\texttt{exp} \neq \texttt{e\_INF\_OR\_NAN} \implies \neg\texttt{isNaN}(v) && \text{(R2)}\\
\wedge & \neg R.\texttt{canBeNegZero} \implies v \neq -0.0 && \text{(R3)}\\
\wedge & \neg R.\texttt{canHaveFraction} \implies \texttt{round}(v) = v && \text{(R4)}\\
\wedge & R.\texttt{hasInt32LowerBound} \implies (\texttt{isNaN}(v) \vee v \geq R.\texttt{lower}) && \text{(R5)}\\
\wedge & R.\texttt{hasInt32UpperBound} \implies (\texttt{isNaN}(v) \vee v \leq R.\texttt{upper}) && \text{(R6)}\\
\wedge & R.\texttt{exp} \geq \texttt{expOf}(v) && \text{(R7)}
\end{aligned}
$$

Figure 3.6: The definition of the predicate $\texttt{inRange}(R,v)$ that states whether a floating-point number v is in range R.

3.5.1 Semantic Meaning of Predicate Facts

We split the semantic meaning $[\![R]\!]$ of a range fact R into two parts: (1) a predicate that connects R to values and (2) a well-formedness invariant expressed solely on R. In particular:

$$[\![R]\!](v) \triangleq \texttt{inRange}(R,v) \wedge \texttt{wellFormed}(R)$$

The `inRange` predicate connects the range fact to values from JavaScript semantics. The `wellFormed` predicate is more unique: it isn't about a connection to semantic values, but instead about implementation-specific Firefox invariants that are necessary for proving the flow functions correct. We explain each piece of $[\![R]\!]$ in turn below.

The `inRange` predicate Figure 3.6 shows the `inRange` predicate, which we derived from Firefox code and comments. An important point here is that this predicate is unusually complex—far more complex than any of the semantic meanings used in prior automated verification efforts for program analyses [166]. A typical meaning for a range analysis fact in prior work is:

$$\texttt{inRange}(R,v) \ \triangleq\ R.\texttt{lower} \leq v \leq R.\texttt{upper}$$

$\texttt{wellFormed}(R) \triangleq$

$$
\begin{aligned}
& R.\texttt{lower} \geq \texttt{JS\_INT\_MIN} \wedge R.\texttt{lower} \leq \texttt{JS\_INT\_MAX} && \text{(W1)}\\
& \wedge R.\texttt{upper} \geq \texttt{JS\_INT\_MIN} \wedge R.\texttt{upper} \leq \texttt{JS\_INT\_MAX} && \text{(W1)}\\
& \wedge \neg R.\texttt{hasInt32LowerBound} \implies R.\texttt{lower} = \texttt{JS\_INT\_MIN} && \text{(W2)}\\
& \wedge \neg R.\texttt{hasInt32UpperBound} \implies R.\texttt{upper} = \texttt{JS\_INT\_MAX} && \text{(W2)}\\
& \wedge R.\texttt{canBeNegZero} \implies \texttt{contains}(0, R) && \\
& \wedge (R.\texttt{exp} = \texttt{e\_INF} \vee R.\texttt{exp} = \texttt{e\_INF\_OR\_NAN} \vee R.\texttt{exp} \leq 1023) && \text{(W3)}\\
& \wedge (R.\texttt{hasInt32LowerBound} \wedge R.\texttt{hasInt32UpperBound}) && \\
& \qquad \implies R.\texttt{exp} = \texttt{expOf}(\texttt{max}(|R.\texttt{lower}|, |R.\texttt{upper}|)) && \\
& \wedge R.\texttt{hasInt32LowerBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{lower}) && \text{(W4)}\\
& \wedge R.\texttt{hasInt32UpperBound} \implies R.\texttt{exp} \geq \texttt{expOf}(R.\texttt{upper}) && \text{(W4)}
\end{aligned}
$$

Figure 3.7: Well-formedness for a Firefox range.

This predicate is simpler than our inRange because inRange must handle the complexities of realistic range analysis for JavaScript (§3.2), i.e., tracking floating point numbers, since all JavaScript values are double-precision floats.

Given a range fact R and a JavaScript value v, the predicate $\texttt{inRange}(R, v)$ is true iff the following conditions hold. First, if the exponent field of R is less than a special Firefox value, e_INF, v must not be infinite. Similarly, if R's exponent is less than the special value e_INF_OR_NAN, v must not be NaN. If R's canBeNegativeZero flag is not set, v should not be -0.0; if R's canHaveFractionalPart flag is not set, v should be a whole number. Finally, if R has a lower bound (hasInt32LowerBound flag), v should be greater than or equal to that lower bound (lower). The hasInt32LowerBound flag indicates whether a range contains numbers that are smaller than thirty-two bit integers; it allows Firefox to internally use integers to represent JavaScript numbers when possible.

The `wellFormed` predicate Figure 3.7 shows the wellFormed predicate, which is unusual because it does not relate the range fact to semantic values. Instead, it simply imposes constraints on the range fact itself; it is an invariant that Firefox flow functions depend on to be correct. We derived this invariant from a set of long comments and a handful of invariant-checking functions in the Firefox codebase.

All range facts should have `canBeNegativeZero` set only when zero is contained within their range[2], where `contains` is defined as follows for range R and value v:

$$\texttt{contains}(R,v) \triangleq v \geq R\texttt{.lower} \wedge v \leq R\texttt{.upper}$$

Furthermore, exponents should either be in the range 0 to 1023, or should have the special value `e_INF` (1024) for infinity, or the special value `e_INF_OR_NAN` (65535) for NaN or infinity. In addition, the exponent should also be consistent with the lower and upper bound, if they exist.

Now, we walk through an example showing how Firefox routines break when invariants are violated, focusing on the invariant relating `hasInt32LowerBound` with R`.lower` and `hasInt32UpperBound` with R`.upper`. The Firefox implementation of range analysis requires that if a range's `hasInt32LowerBound` is unset, then its `lower` field equals `JS_INT_MIN`; similarly, if its `hasInt32UpperBound` is unset, its `upper` field equals `JS_INT_MAX`. Consider the following Firefox range analysis code for `max`, which calculates the new `lower` field and the new `hasInt32LowerBound` flag in the returned range fact:

```
int32_t newLower = max(lhs->lower, rhs->lower)
bool newHasLower = lhs->hasInt32LowerBound || rhs->hasInt32LowerBound
```

Now consider two input ranges to the `max` function, R_1 and R_2, and assume the output range is R_3. If the Firefox invariant doesn't hold, we can have $\neg r_1$`.hasInt32LowerBound` and R_1`.lower` $=$ 1000, and R_2`.hasInt32LowerBound` and R_2`.lower` $= -1000$. This means that the above code will set R_3`.hasInt32LowerBound` to true, and R_3`.lower` to 1000 (since the result lower bound is the maximum of the input lower bounds).

Now consider two JavaScript values v_1 in R_1 and v_2 in R_2 (meaning that $\texttt{inRange}(R_1,v_1)$ and $\texttt{inRange}(R_2,v_2)$ are both true). Let v_1 be $-\infty$ (since $\neg R_1$`.hasInt32LowerBound`) and v_2 be

[2]This is only an invariant on optimized ranges; unoptimized may have set flags even if their ranges don't contain zero.

-1000. Then, JavaScript semantics says that $\texttt{max}_{js}(v_1, v_2)$ is -1000, but -1000 is well below R_3's lower bound of 1000.

However, once we add the `wellFormed` invariant, this problem is fixed: when `hasInt32LowerBound` is not set, `lower` must be `JS_INT_MIN`, so R_1`.lower` cannot be 1000. Now that R_3's lower bound is `JS_INT_MIN` instead of 1000, it correctly captures $\texttt{max}_{js}(v_1, v_2)$ by including -1000. This example shows that `max`'s range analysis *relies* on the invariant that `lower` is `JS_INT_MIN` when `hasInt32LowerBound` is not set.

Other Verification Conditions We prove two additional properties of the Firefox range analysis: we prove the correctness of the functions for combining range facts—union and intersection—which the JIT uses within its larger range analysis loop. We will use `union` and `intersection` to refer to Firefox's union and intersection functions for range facts. Conceptually, given two range facts R_1 and R_2, we want to show that $\texttt{union}(R_1, R_2)$ is an overapproximation[3] of the mathematical union operation $\cup$ on the semantic values contained in R_1 and the semantic values contained in R_2. We achieve this as follows. Given two well-formed ranges R_1 and R_2, we let $R_3 = \texttt{union}(R_1, R_2)$. Then we want to show that for all JavaScript values v, we have:

$$\texttt{inRange}(R_1, v) \vee \texttt{inRange}(R_2, v) \implies \texttt{inRange}(R_3, v)$$

Similarly, for intersection we let $R_3 = \texttt{intersect}(R_1, R_2)$, and prove that:

$$\texttt{inRange}(R_1, v) \wedge \texttt{inRange}(R_2, v) \implies \texttt{inRange}(R_3, v).$$

3.5.2 Using VeRA to Express Predicates

VeRA exposes an internal verification domain-specific language (DSL) embedded in Haskell. Verification developers use the language to express verification infrastructure, including verification conditions and the semantic meaning of range facts. The DSL exposes a number of JavaScript operators—i.e., implementations of $\texttt{op}_{js}$—against which to verify range analysis func-

[3] And it is necessarily an overapproximation given how Firefox's range analysis object is implemented: consider a union of two ranges, one that can include fractions and one that can't.

tions. If verification developers wish to verify new operations, it is straightforward to expose new JavaScript routines in the VeRA internal DSL. To express predicates and verification conditions, the DSL also exposes SMT directives. They allow verification developers to make assumptions, call the SMT solver, and push and pop new incremental solver contexts.

In practice, proving the conjunction of `wellFormed` and `inRange` for each operator is suboptimal: if any component of either predicate does not finish, the entire verification proof does not finish. As a result, we prove individual conditions within each predicate separately; Section 3.6 describes each condition that we prove and the time the proof takes to finish (or not). Mechanically, to prove a given predicate part, we query an SMT solver with its negation—e.g., we check whether the operation can produce any values *outside* of the computed range. If the formula is unsatisfiable, there are no such values and the range analysis routine is safe. If the formula is satisfiable, the model provided by the solver is a concrete counterexample that captures that input ranges and values for which the range analysis routine is unsafe.

3.6 Implementation and Evaluation

We implement VeRA—both the compiler for VeRA C++ and the internal verification language—in 1384 lines of Haskell. Since the implementation details of the compiler are standard, we only describe the details relevant to answering our evaluation questions. All our source code and data sets are available at `https://vera.programming.systems`.

We evaluate VeRA by asking six questions:

Q1 Can VeRA prove Firefox range analysis routines correct?

Q2 Can VeRA proofs catch real correctness bugs?

Q3 Are the VeRA proofs correct?

Q4 Do the verified routines work correctly in Firefox?

Q5 How do the verified routines perform in Firefox?

Q6 How hard is it to integrate verified routines into Firefox?

To answer these questions, we port 21 top-level Firefox range analysis flow functions to VeRA C++, try to prove their correctness, and then re-integrate them into the browser. We are able to: prove 137 separate facts about these routines; identify a new Firefox analysis bug; and correctly detect an old analysis bug. A version of Firefox that uses our verified routines still performs comparably to standard Firefox, and it still passes *all* (147,322) Firefox JavaScript tests.

3.6.1 Proofs

We choose to verify 19 Firefox flow functions because they are the complete set of Firefox `Range`-type flow functions for JavaScript operators; we discuss this further in Section 3.7 (e.g., as a result, we don't check division). In addition, we verify the `union` and `intersect` functions, which are not JavaScript operators but instead Firefox-internal functions that combine two different `Range` objects; this brings the total number of routines we attempt to verify to 21. Verification of both `union` and `intersect` times out, but VeRA finds a bug in our port of an older, broken version of `intersect` [9]. Finally, we port 25 helper functions called by each verified function—i.e., every function except the `optimize` function (§3.7). Figure 3.8 summarizes our results.

All operators followed by asterisks in the table (e.g., `rsh`) are only valid for 32-bit ranges.[4] Thus, for each bitwise operator, we only prove (1) the simple predicate from Section 3.2 and (2) the absence of undefined behavior in the result range's upper and lower bounds. We don't worry about `wellFormed` for these operations, either; none of them alter the floating-point-specific fields of the range.

[4]This is an internal Firefox invariant: each one of these functions starts with an assertion that its operands ranges only include 32-bit numbers.

For floating-point operators (e.g., `add`), we separately prove each condition in the `inRange` predicate and the `wellFormed` predicate with two exceptions: since we do not call `optimize` on our result ranges, we only verify the `wellFormed` conditions that apply to non-optimized ranges (i.e., our output ranges are correct but may not have the tightest possible bounds). Figure 3.8 goes into more detail about which columns correspond to which conditions in Figure 3.6 and Figure 3.7.

Multiple proofs fail: `ceil` and broken `intersect` are both real bugs, while `ursh` and `ursh'` require extra invariants on the input ranges. We don't amend them because their failure is actually an interesting manifestation of a comment above both functions [26]:

```
// ursh's left operand is uint32, not int32, but for range
// analysis we currently approximate it as int32. We assume
// here that the range has already been adjusted...
```

In other words, over all possible inputs, `ursh` is *not* correct.

Our proof code, i.e., the code that automatically verifies each part of both the `inRange` and `wellFormed` predicates, amounts to 804 lines of Haskell. We run all proofs on a virtual machine (QEMU-KVM, Linux 5.3.11) running Arch Linux with 16 GB of memory and 4 vCPUs. The processor is an AMD Ryzen Threadripper 2950X 16-Core with a base clock rate of 3.5GHz. We use the Z3 SMT solver (4.8.6), which we call with custom Haskell bindings that extend the haskell-z3 library [29], using a 20-minute timeout.

Can we prove Firefox range analysis routines correct? We successfully prove or refute 137 conditions out of a possible 159, for a success rate of ≈86%; the shortest proofs complete in under a second, while the longest takes ≈ten minutes. The results suggest that **R5.double**, **R6.double**, and **W4** are particularly challenging to verify. **R5.double** and **R6.double** are more challenging than their integer counterparts because they involve reasoning about floating-point values, which is generally more expensive. **W4** is challenging because it involves proving a relationship between

Operation	**R1**	**R2**	**R3**	**R4**	**R5.i32**	**R5.double**	**R6.i32**	**R6.double**	**R7**	**W1**	**W2**	**W3**	**W4**	**Undef**
`add`	15	2	5	386	2	∞	2	∞	21	2	1	2	80	1
`sub`	13	2	11	445	8	∞	5	∞	14	2	2	2	78	1
`and*`	-	-	-	-	2	-	1	-	-	-	-	-	-	1
`or*`	-	-	-	-	2	-	2	-	-	-	-	-	-	1
`xor*`	-	-	-	-	2	-	2	-	-	-	-	-	-	1
`not*`	-	-	-	-	2	-	1	-	-	-	-	-	-	1
`mul`	92	65	22	362	∞	∞	∞	∞	94	4	4	4	∞	11
`lsh*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`rsh*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`ursh*`	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
`lsh'*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`rsh'*`	-	-	-	-	1	-	1	-	-	-	-	-	-	1
`ursh'*`	-	-	-	-	**X**	-	**X**	-	-	-	-	-	-	1
`abs`	1	1	1	5	1	∞	1	224	1	1	1	4	∞	1
`min`	2	20	2	2	5	224	2	∞	3	2	1	2	∞	1
`max`	3	17	2	2	15	∞	2	∞	4	3	2	12	∞	1
`floor`	4	2	1	5	1	146	1	9	54	1	1	5	∞	1
`ceil`	∞	1	**X**	8	1	5	1	9	266	1	1	∞	∞	1
`sign`	1	1	1	1	1	2	1	2	1	1	1	1	2	1

Figure 3.8: The time it takes, in seconds, for VeRA to verify predicates the predicate from Section 3.5. **R1**-**7** correspond to the lines in the `inRange` predicate, while **W1**-**4** correspond to line groups one through four in `wellFormed`. Broadly, **R1** makes sure the routine handles infinities correctly, **R2** NaNs, **R3** -0.0, **R4** fractions, **R5**s lower bounds (over both 32-bit integer and double values), and **R6**s upper bounds. **W1** ensures missing upper and lower bounds imply a minimum or maximum value for `lower` and `upper`; **W2** ensures `lower` and `upper` are always valid JavaScript 32-bit numbers; **W3** ensures the range's exponent is valid, and **W4** ensures the exponent is consistent with the upper and lower bound. Finally, **Undef** shows the time it takes to verify that the range computations for `upper` and `lower` are free of undefined behavior. ∞indicates timeout, while **X** indicates verification failure.

two properties of the range, both of which may be modified by the range analysis. Finally, **R1** and **W3** of `Math.ceil` may timeout because they involve bounding the size of an exponent, since `Math.ceil` involves extracting the exponent from the absolute value of the range bounds.

There is hope, however. Though VeRA does not verify conditions for certain routines (e.g., correctness of `intersect`), it *is* able to catch multiple bugs in broken versions of unverified routines. For example, as we discuss later in this section, we port an older, broken version of `intersect` to VeRA C++. VeRA is able to detect the bug in this version in 173 seconds, even though the proof for the current version of `intersect` never finishes. VeRA also catches a number of errors that we introduced while copying code over from Firefox; as we improved VeRA, it went from being a custom DSL to a subset of C++, so porting was not always as easy as copy-pasting browser code. For example, we switched an upper and lower bound in `sub`, which caused it to fail the floating-point lower bounds check—even though this check never verifies in the fixed version of `sub`. VeRA caught one other porting error in `sub` (use of the field `canBeNegativeZero` instead of the function `canBeZero`), and at least two more in `mul` (switched `lhs` and `rhs`, and use of `canBeFiniteNonNegative` in place of `canBeFiniteNegative`).

A new Firefox bug VeRA found a bug in Firefox's range analysis for the `Math.ceil` operator [4], which rounds its input up to the nearest integer (e.g., the ceiling of 2.5 is three). The bug, which follows, has existed since the routine's introduction six years ago [2]:

```
Range* Range::ceil(TempAllocator& alloc, const Range* op) {
  Range* copy = new (alloc) Range(*op);
  if (copy->hasInt32Bounds())
    copy->max_exponent_ = copy->exponentImpliedByInt32Bounds();
  else if (copy->max_exponent_ < MaxFiniteExponent)
    copy->max_exponent_++;

  copy->canHaveFractionalPart_ = ExcludesFractionalParts;
  copy->assertInvariants();
```

```
    return copy;
  }
```

The routine looks straightforward. Given an input range, it adjusts the range's exponent upwards by one—to account for upward rounding—and unsets the `canHaveFractionalPart` flag—since the result is always a whole number.

The problem lies in what `ceil` *doesn't* do: it never adjusts the input range's `canBeNegativeZero` flag. JavaScript semantics, though, dictate that `Math.ceil(x) = -0` when `x` is between zero and negative one. Therefore, given a range with lower and upper bound [-1, 0] and an unset `canBeNegativeZero` flag, `ceil` will *not* correctly set the flag.

VeRA identifies the error after about three seconds, and provides the following (shortened) counterexample:

```
result_range_canBeNegativeZero : 0
start_range_canBeNegativeZero : 0
start_range_lower : -128
start_range_upper : 0
start : -1.166614929399505e-301
```

It finds a start value, `start`, within the `start_range` of [-128, 0] with the `canBeNegativeZero` flag unset. It notes that `ceil` of `start` is `result`, -0, but that the result range does not have the `canBeNegativeZero` flag set.

After confirming the bug with Mozilla engineers, we tried to patch the bug and use VeRA to verify the patch. VeRA rejected our first attempt—it was wrong—but approved the next one [4]:

```
copy->canBeNegativeZero_ = ((copy->lower_ > 0) || (copy->upper_ <= -1))
                          ? copy->canBeNegativeZero_
                          : IncludesNegativeZero;
```

Now, the function sets the `canBeNegativeZero` flag to be true when the resulting range can be includes values between negative one and zero.

An old Firefox bug To test VeRA further, we port a buggy version of Firefox's `intersect` operator, which takes the intersection of two ranges [10]. The buggy operator deduces that the intersection of two ranges $r_1 \cap r_2 = \emptyset$ if the upper bound of r_1 doesn't overlap with the lower bound of r_2. This behavior is correct—unless both ranges contain NaN, in which case the result range should include NaN, too. VeRA identifies the error after 173 seconds, and provides a counter example showing an element in the two input ranges (NaN) that was not included in the output range.

Are VeRA proofs correct? Verifiers can be as broken as the code they are intended to verify. To guard against this possibility, we use Haskell's QuickCheck [88] to automatically random-test the semantics of (1) the operators in the VeRA IR and (2) the JavaScript operators that VeRA uses for verification.[5] For each JavaScript operator, we generate JavaScript code that performs the operations, evaluates it with Node.js, and compares the result against that produced by our SMT model. We use Node.js (version 10.1.0) because it uses the Chrome V8 JavaScript engine and is thus likely to have different bugs from Firefox (and thus VeRA). Our C++ operator tests are similar; we use Clang version 9.0.0. This checking proved useful—e.g., QuickCheck found a bug in our implementation of the C++ floating-point `abs` operator. For further assurance, we also cross-checked our JavaScript semantics against KJS [201].

3.6.2 Verified Routines in the Browser

We integrate our verified range analysis into Firefox 72.0a1 (commit 10c8c9240d). Our versions of the Firefox range analysis functions amount to 621 lines of C++ code. In this section, we describe the effort it took to retrofit Firefox and measure the performance of our modified browser when compared with *vanilla*, unmodified Firefox.

[5]We use QuickCheck 2.13.2 on GHC 8.6.5 and configure it to test each operator on different types (e.g., integers of varying widths) 1,000 times.

How hard is it to integrate VeRA code into Firefox? We retrofit Firefox in two steps. First, we extend Firefox's `Range` class with a `verifedRange` field—pointing our verified object—and modify the class setters and getters to forward all accesses to the corresponding `verifiedRange` fields. For example, we rewrite,

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->lower_;
  int32_t u = op->upper_;
  ...
}
```

to:

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  int32_t l = op->verifiedRange.lower;
  int32_t u = op->verifiedRange.upper;
  ...
}
```

Then, we replace individual function bodies with calls to our verified functions. For example, we rewrite `abs` to:

```
Range* Range::abs(TempAllocator& alloc, const Range* op) {
  auto vRange = verified::abs(op->verifiedRange);
  return Range::fromVerifiedRange(alloc, vRange);
}
```

Our porting effort was surprisingly low: two engineers—neither of whom is a Firefox core developer—integrated our VeRA C++ routines into Firefox over the course of two days. We think this effort can be reduced even further: both steps are mechanical and can be automated to only require human intervention when tests fail.

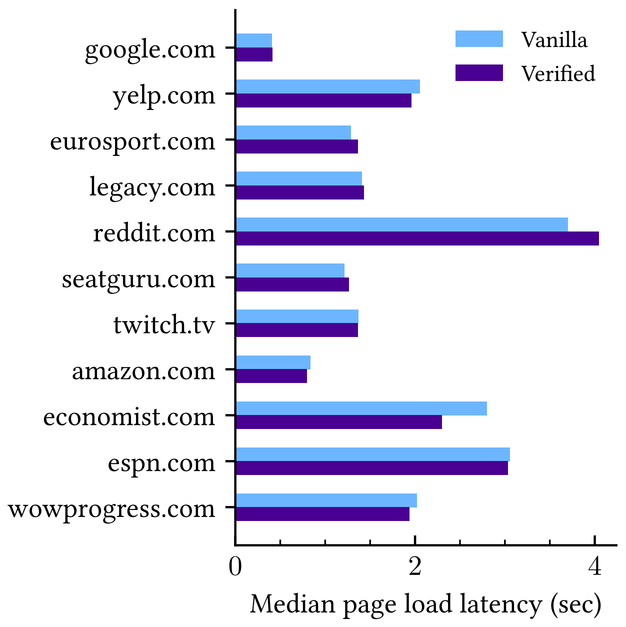


Figure 3.9: Page load latency of popular and unpopular websites.

Do the verified routines work correctly? To test our ports, we run all of Firefox's JavaScript suites (using their `mach` build tool): the 7,364 JIT tests, 394 JSAPI tests (which use the `Range` interface directly), and 139,564 general JavaScript tests. Though VeRA passes all of Firefox's tests now, it failed some tests along the way.

After altering Firefox to call VeRA routines whenever applicable, all but three of the JIT tests passed; one timed out and two failed. This was because one of our ported functions did not initialize every field of the range object; the function was only called by a range analysis routine for a bitwise operator, so to verify it, we did not need to set the floating-point-specific range fields. We also failed two of the JSAPI tests, both due to typos in our porting of `intersect` and `sign` functions. The VeRA verification actually caught the `sign` bug, but due to a miscommunication within the team, the fix didn't make it into Firefox immediately. The bug in `intersect` caused it to over-approximate the possibility of negative zeroes—but our verification specifically allows over-approximation by design (§3.5).[6] Once we fixed these bugs, all JIT and JSAPI tests passed, and when we ran the other 139,564 Firefox JavaScript tests, all of those passed, too.

[6]Range combinations in Firefox are necessarily over-approximations.

How do the verified routines perform? We evaluate the performance impact of VeRA on JavaScript execution and end-to-end page latency. We run all the performance benchmarks on an Intel 8 core (i7-6700K machine with a base clock rate of 4.0GHz) machine running Arch Linux with 64 GB of memory. We compare our browser against a vanilla, unmodified Firefox, and find the impact of VeRA to be on par. This is not surprising: our C++ code is very similar to Firefox's original range analysis code.

To measure the performance of VeRA on JavaScript execution, we run the JetStream 2 benchmarking suite [38], created by the WebKit team. This benchmark subsumes the now-deprecated SunSpider [39] benchmarking suite. JetStream 2 consists of 64 benchmarking tests that measure representative JavaScript and WebAssembly workloads for both start-up times, code execution, and "smoothness".[7] Each test reports a *score*, corresponding to how well the browser performed. We present the JetStream 2 results, as run on 2019-11-22, in the appendix of the original Scooter paper; the overall performance of our modified Firefox is on-par with unmodified Firefox—our browser scored 75.688 while vanilla Firefox scored 77.206. JetStream 2 computes this score by "taking the geometric mean over each individual benchmark's score". Our mean and median scores are within 3.2% and 8.5%, respectively. The maximum difference in scores is in 52%, in the string-unpack-code-SP benchmark which stresses string manipulation. We think the big differences are largely due to noise; running the string-unpack-code-SP benchmark outside the browser (1,000 iterations in the js shell) we only observed a 0.48% difference.

We measure the impact of our verified code on end-to-end page latency by browsing a representative sample of both popular and unpopular websites. In particular, we use the list of 11 sites curated by the Chrome team in their recent Chrome sandboxing work [214]. For each site, we use Mozilla's Talos benchmarking tool to measure the time it takes to render a page (i.e., the time to first paint [27]), taking the median of 50 runs (after a 5 run warm-up). Figure 3.9 presents our measurements. The median and average latencies of browsing these sites with our

[7]The WebAssembly pipeline in Firefox actually uses the JavaScript pipeline and thus VeRA in the verified browser.

browser are within 5% of vanilla Firefox. The biggest slowdown is on reddit.com (18%), while the biggest speedup is on economist.com (-9%); like [214], we attribute these bigger difference to the inherently noise introduced by media content and the network. Overall, these results are encouraging: VeRA does not impose overheads that are prohibitive to its adoption.

3.6.3 PrimaVeRA

Two years after the initial development of VeRA, we reimplement its range analysis routines in an automated verification language based on corral[163]. We call this reimplementation *PrimaVeRA*, and demonstrate two key findings:

1. PrimaVeRA can replicate and even improve on the results of VeRA using a generic automated verifier.
2. Even naive range analysis routines for floating point arithmetic are out of reach for today's SMT solvers.

Rather than baking the verification routine directly into the solver, PrimaVeRA relies on programmer-provided verification conditions declared with inline `assert` and `assume` statements. For example, VeRA's verification routine for absolute value is written in PrimaVeRA like so:

```
var r: Range;
var d: Double;
fn absSpec() {
  assume wellFormed(r);
  assume inRange(r, d);
  let dOut = JS::abs(d);
  let rOut = Range::abs(r);
  assert wellFormed(rOut);
  assert inRange(rOut, dOut)
}
```

Operation	Time (s)
`abs`	667
`ceil`	3099
`floor`	2592

Figure 3.10: PrimaVeRA verifies the correctness of routines that VeRA is incapable of.

In PrimaVeRA, we do not divide the `wellFormed` and `inRange` into subproperties. This more naturally matches the usage of debug asserts in the Firefox codebase, but does not allow for partial results. A naive port of VeRA's routines and verification conditions to PrimaVeRA, results in timeouts for all routines which timed out in VeRA. However, now that the verification conditions were expressed in PrimaVeRA code rather than embedded in the solver, we were able to optimize the verification condition to successfully verify 3 more routines.

Optimizing the `wellFormed` predicate **W4** makes use of an `expOf`(n) function which yields $\lfloor log_2(|n|) \rfloor$ for a signed integer, *n*. VeRA implements this function using `popcnt`, which is difficult for the SMT solver to optimize. In PrimaVeRA, we implement `expOf`(n) using a chain of 31 `if`-statements corresponding to the range of possible values for each return value. This optimization allowed the `abs` routine to fully verify.

Optimizing the `inRange` predicate. **R7** states that for a range to contain a double, the `R.exp` must be greater than or equal to `expOf`(R). This means that any overapproximation of `expOf`(R) that yields a larger value is sound, but possibly incomplete. We also know that `R.exp` is an 16-bit unsigned integer, meaning it could never be negative. Combined these facts mean that an overapproximation of `expOf`(R) which yields `0` for all floating points with negative exponents, is both sound and complete. VeRA, however, does not do any approximation; it implements a precise version of `expOf` which accounts for subnormal floating points and supports floating points with negative exponents. In PrimaVeRA, we implement this approximation, which allows us to avoid special cases for subnormals and drastically shrinks the potential range of possible values. Coincidentally, Firefox already implements this version of `expOf` in order to determine the range for a floating point constant. With these optimizations, PrimaVeRA is capable of verifying 3 new ops (see Figure. 3.10). Alas, arithmetic binary operations such as `add` remain out of reach. To investigate the feasibility of verifying range analysis over floating point addition, we implement a simpler analysis using range objects with only optional 32-bit signed integer bounds:

```
fn add(lhs: Range, rhs: Range) -> Range {
  let minSum = (lhs.min as Int64) + (rhs.min as Int64);
  let maxSum = (lhs.max as Int64) + (rhs.max as Int64);
  Range {
    hasMin: lhs.hasMin && rhs.hasMin && minSum >= INT32_MIN,
    hasMax: lhs.hasMax && rhs.hasMax && maxSum <= INT32_MAX,
    min: minSum as Int32,
    max: maxSum as Int32
  }
}

fn inRange(r: Range, n: Double) {
  if Double::is_nan(n) { return true; }
  if r.has_min && n < r.min { return false; }
  if r.has_max && n > r.max { return false; }
  true
}

fn wellFormed(r: Range) {
  r.min <= r.max
}
```

Our hypothesis was that removing all the floating point math, i.e. using a less-precise analysis, the verification condition would be tractable, but the solver still timed out This indicates that further verification of range analysis will require improvements to the underlying SMT solver, or will require alternate techniques such as manual theorem proving.

3.7 Discussion, Limitations and Future Work

Why not existing proof tools? We started out building a DSL for range analysis verification, and ended up building a compiler from both C++ and an internal verification language to SMT. There are many existing tools that can translate programming languages into SMT [80, 212], and they primarily operate on an existing compiler IR (e.g., LLVM IR) instead of defining their own IR. Verifying JavaScript JIT optimization passes at the LLVM IR level is something to strive for in the future, but using a small language is an easier start; to our knowledge, no LLVM-IR-level tool supports JavaScript semantics, some don't support C++ reliably [24], and the ones that do can get lost in complex class object hierarchies—in IR, series of pointer offset calculations—and as a consequence struggle to verify anything at all. Furthermore, using these tools requires integrating them with (very complicated) browser build systems. We, on the other hand, don't provide any guarantees about the final machine code.

What we don't verify We only verify range analysis routines for Firefox. Chrome's range analysis pass is tied to its type inference pass—different ranges correspond to different types (e.g., 32-bit integers have a bound). We consider extending VeRA to other browsers future work.

Within Firefox, the range analysis functions that we verify all return the basic Firefox `Range` object type. These functions contain local range analysis logic, and are called during the range analysis computation for different MIR nodes, Firefox's middle-level intermediate representation of JavaScript programs. For example, the `computeRange` method for the `MCeil` node (representing `Math.ceil`) simply wraps `Range::ceil` and thus reaps the benefits of our verification effort. Many MIR nodes, however, do not correspond to a JavaScript-level constructs (e..g, `MSpectreMaskIndex` is used to represent a masked array index) and we thus do not verify them. We also do not verify the range analysis algorithm itself nor Firefox's use of ranges in code generation or other optimization passes (e.g., DCE or BCE). These are natural extensions to our work. Similarly, since other JIT components (e.g., type inference) have been a source of security vulnerabilities, we hope to address them as future work (e.g., by building on JIT type inference foundations [131, 134]).

3.8 Related Work

VeRA lies at the intersection of work on compiler verification, JIT compilation, and browser security.

Verifying optimizations using SMT Various prior systems have used DSLs combined with SMT solvers to express and verify compiler optimization correctness: for example, Cobalt [165], Rhodium [166], PEC [162] and Alive [172]. These systems mostly focus on proving correctness of *transformations*, e.g., scalar optimizations in Cobalt and Rhodium, control flow rewrites in PEC, and peephole optimizations in Alive. There has been much less work on using DSLs and SMT solvers to prove the correctness of *analyses*, with the notable exception of the Rhodium system [166]. While conceptually the techniques in VeRA are similar to Rhodium's techniques for analysis correctness, the analyses in Rhodium are relatively simple, with the semantic predicate of facts often containing a single term. In contrast, VeRA demonstrates how to use DSLs and SMT solvers to verify the correctness of analyses in a realistic setting: our work handles all the corner cases of Firefox's range analysis, which in turn includes a semantic predicate for the range facts with 16 cases.

Another difference between our work and prior work is the type of constructs we support (in VeRA C++). Alive supports pointers and arrays (with static, known sizes), while VeRA does not support either construct—but neither system handles loops, since neither peephole optimizations nor range analysis computations typically require them.

Compiler and analysis verification in a proof assistant Another approach to general compiler verification is *foundational verification*. In foundational verification, the programmer writes the compiler in a *Proof Assistant*, and then uses the proof assistant to interactively prove that the compiler is correct. Examples of foundationally verified compilers include CompCert [167], CompCertTSO [262], Compositional CompCert [242], and CakeML [161]. Examples of semantic IR frameworks include the Vellvm system [285], which provides a formal semantics for LLVM IR that others can use to verify IR optimizations/transformations.

In addition to entire compilers or IR frameworks, there has also been work on specific analyses and optimizations that are foundationally verified. For example, Versaco [151] is a foundationally verified static analyzer for CompCert; developers can use it to write their own analyses that prove properties of analyzed programs. For specific optimizations, Zhao et al. use Vellvm to verify a version of LLVM's mem2reg transformation, which changes memory references to register references [286, 25]. Mullen et al. use the Coq proof assistant to verify peephole optimizations for the CompCert verified C compiler [167, 183]. Finally, Tatlock et al. extend CompCert with a DSL for expressing optimizations [251]—combining the DSL approach with foundational verification.

Because proofs in foundational verification are performed in full detail, foundational verification provides the strongest possible correctness guarantees. However, these proofs often require a significant amount of expert human guidance, making them very difficult to complete. In contrast, VeRA allows browser developers to express their analysis in a subset of C++, and then provides automated verification to the developer without any additional effort or verification knowledge.

Verification of JIT compilation There has also been work specifically on verifying correctness of JIT compilers, including work on verifying a non-optimizing JIT compiler [187], and work on defining correctness criteria for trace-based JIT compilation [132]. There is also ongoing work on verifying a JIT in Coq [49], which includes support for some optimizations, though it is not clear which ones; similarly, there is ongoing work on verifying a JIT using symbolic execution [233]. None of this work focuses on the specific challenge that we are addressing, namely verifying the complex analyses that drive optimizations in browser JITs.

Translation validation Another approach to compiler correctness is translation validation [228, 190, 204]: each time the compiler runs, a validator tries to prove that the transformed code behaves the same as the original code. While translation validation can find compiler bugs, it does not guarantee the absence of bugs, as VeRA attempts to. For translation validation to guarantee the absence of bugs, it would have to do validation on production runs, which incurs compilation overhead—not ideal for a JIT.

Recent work by Taneja et. al. goes further by proposing an algorithm for sound and *maximally precise* dataflow facts like integer ranges (similar to this work) and known bits, among others [248]. For a given code fragment, they (1) use their algorithm (implemented with an SMT solver) to compute dataflow facts about that fragment and (2) compare those facts to the ones LLVM has computed. This technique has identified several precision errors in LLVM's analyses, and adapting it to help developers design tighter ranges is interesting future work.

Verification and floating point numbers Other systems also handle the challenge of verifying floating-point code. Recently, Boldo et al. formalize IEEE-754 semantics in Coq in order to extend CompCert to support programs that use floating-point numbers [72]. More similar to VeRA, Icing [53], which builds on the verified ML compiler CakeML [161], is a language for writing "fast-math" optimizations. Multiple projects also extend the Alive system to support peephole optimizations related to floating point numbers [178, 195]. Recently, Becker et al. use the Daisy tool [98] to verify optimizations to floating-point computations—in programs, not in compilers—that the Herbie [200] tool generates [54].

Testing compilers Beyond verification, there are other approaches to compiler correctness. One is automatic testing, or "fuzzing," which finds bugs but does not guarantee the absence of errors. For example, the CSmith tool automatically generates *useful*—well-formed, undefined-behavior-free—inputs for testing C compilers [278]. Dewey et al. present a system for fuzzing the Rust compiler's type checker [103]. Finally, there are fuzzers specific to JavaScript interpreters: Fuzzilli [124, 125], and CodeAlchemist [136]. Fuzzilli has been remarkably effective at finding bugs in JavaScript engines; its bug showcase lists two dozen security issues [11].

Verified JavaScript semantics There has been a significant effort to formalize (parts of) the JavaScript language. Most of these efforts start with a simple core language and extend it with unwieldy JavaScript features (e.g., `eval`, property descriptors, and `with`) [173, 206, 130, 114]. The JSCert JavaScript subset was even mechanized in Coq, from where they extracted a verified correct interpreter [71, 70, 115]. KJS [201] provides a complete JavaScript semantics in the K framework. Though we cross-check our semantics against KJS and JSCert's (where possible), this work is complimentary: they focus on verifying JavaScript semantics, we focus on the JITs that should preserve these semantics.

Verification in the browser A verified browser kernel, Quark [148], demonstrates that verification is possible in the browser setting. Though we're a long way from a verified Firefox, browsers are interested in verified software. For example, both Firefox and Chrome incorporated verified cryptographic primitives into their TLS stacks [289, 288].

Safer browser JITs Another approach to JIT safety is to limit the damage of bugs, not prevent them. NaClJIT sandboxes both the JIT compiler and the code it produces [37]. RockJIT applies a control-flow integrity policy to the JIT compiler and the code it produces [193]. NoJITSu prevents code-reuse and data-only attacks [202]. Browser vendors have also modified their JITs to reduce the effectiveness of JIT spraying [170], a technique that allows attackers to introduce bytes of their choice into pages marked executable in browser memory [68].

Large, real-world verification Finally, beyond what we have discussed so far, there are many other large, real-world verification efforts. The Astrée static analyzer has verified safety properties of Airbus software [101]; the miTLS project provides verified reference implementations of TLS 1.0, 1.1, and 1.2 [66]; Barbosa et. al. [44] give an overview of verification efforts for crypto code, including discussion of an ongoing effort to formally verify the TLS 1.3 protocol. seL4 [154] is a verified operating systems kernel; and various recent efforts have proved properties of systems software like file systems [83, 82, 234] and in-kernel interpreters [264].

3.9 Conclusion

This chapter presents VeRA, a system for verifying the range analysis pass in browser JITs. VeRA allows browser developers to write range analysis routines directly in the browser (in a subset of C++) and provides a DSL that verification developers can use to encode verification properties (e.g., range analysis invariants). VeRA automatically verifies these properties using SMT.

We use VeRA to encode a semantics for Firefox's range analysis, and then port 22 Firefox analysis routines to VeRA C++ in order to verify them. The ported version of the browser performs on-par with the original. Moreover, VeRA detects a bug that has existed in the browser for six years—and verifies the Firefox patch we wrote to fix the bug.

We reimplement VeRA as PrimaVeRA in a generic verification language, and demonstrate two optimizations which allow for full verification of 3 new range operations. Furthermore, we demonstrate that existing SMT solvers struggle to verify a simple range analysis in the presence of floating point arithmetic operators.

3.10 Acknowledgements

We thank the anonymous reviewers for their comments, and especially our shepherd John Regehr for helping us think through both the content and presentation of this chapter. Thanks to Dawson Engler for technical discussions about this work and other applications of these techniques. Thanks to Riad S. Wahby for uploading our huge artifact (and more!), and to Ranjit Jhala for early discussions. Finally, thanks to Chris Fallin, Nick Fitzgerald, Dan Gohman, Jan de Mooij, Nicolas B. Pierron, and Luke Wagner at Firefox; we can't imagine a more thoughtful and responsive group of people. This work was supported in part by gifts from Cisco, Fastly, Google, Mozilla, and Qualcomm; by the NSF under Grant Numbers CCF-1918573 and CNS-1514435; and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) Fraser Brown, John Renner, Andres Noetzli, Sorin Lerner, Hovav Schacham, Deian Stefan. ACM, 2020. The dissertation author was a primary investigator and author of this paper.

Conclusion

Without formal verification, attackers will continue to find and exploit vulnerabilities in web applications and in the browser. Existing tools are either too difficult (i.e. theorem provers) or too unpredictable (i.e. generic automated verifiers). This dissertation shows that co-designing languages and verifiers can result in powerful and reliable tools that help developers build secure systems rooted in formal methods. We describe three such tools. CT-Wasm (chapter 1) enabls submillisecond verification of constant-time crypto by encoding the constant time discipline into a typesystem. Scooter (chapter 2) prevents data leakage by arming developers with a DSL for specifying policies and database migrations and efficiently lowering these policies and migrations to efficient SMT queries. Lastly, VeRA and PrimaVeRA (chapter 3) shows how highly expressive DSLs can prove complex invariants such as correct range analysis, but struggle to predictably terminate. Together, this work, shows that co-designing languages and verifiers is a promising approach to making the web more secure.

Bibliography

[1] 18.23 FJCVTZS. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0801g/hko1477562192868.html.

[2] Bug 1027510. https://bugzilla.mozilla.org/show_bug.cgi?id=1027510.

[3] Bug 1493900. https://bugs.webkit.org/show_bug.cgi?id=1493900.

[4] Bug 1595329. https://bugzilla.mozilla.org/show_bug.cgi?id=1595329.

[5] Bug 185694. https://bugs.webkit.org/show_bug.cgi?id=185694.

[6] Bug 765127. https://bugzilla.mozilla.org/show_bug.cgi?id=765127.

[7] Bug 765128. https://bugzilla.mozilla.org/show_bug.cgi?id=765128.

[8] Bug 943303. https://bugzilla.mozilla.org/show_bug.cgi?id=943303.

[9] Bug 950438. https://bugzilla.mozilla.org/show_bug.cgi?id=950438.

[10] Bug 950438. https://bugzilla.mozilla.org/show_bug.cgi?id=950438.

[11] googleprojectzero/fuzzilli. https://github.com/googleprojectzero/fuzzilli.

[12] Issue 1390. https://bugs.chromium.org/p/project-zero/issues/detail?id=1390.

[13] Issue 1396. https://bugs.chromium.org/p/project-zero/issues/detail?id=1396.

[14] Issue 1530. https://bugs.chromium.org/p/project-zero/issues/detail?id=1530.

[15] Issue 1544386. https://bugzilla.mozilla.org/show_bug.cgi?id=1544386.

[16] Issue 1669. https://bugs.chromium.org/p/project-zero/issues/detail?id=1699.

[17] Issue 1775. https://bugs.chromium.org/p/project-zero/issues/detail?id=1775.

[18] Issue 1791. https://bugs.chromium.org/p/project-zero/issues/detail?id=1791.

[19] Issue 1809. https://bugs.chromium.org/p/project-zero/issues/detail?id=1809.

[20] Issue 1810. https://bugs.chromium.org/p/project-zero/issues/detail?id=1810.

[21] Issue 1876. https://bugs.chromium.org/p/project-zero/issues/detail?id=1876.

[22] Issue 762874. https://bugs.chromium.org/p/chromium/issues/detail?id=762874.

[23] Issue 880207. https://bugs.chromium.org/p/chromium/issues/detail?id=880207.

[24] Klee and C++ Analysis Target. https://github.com/klee/klee/issues/852.

[25] Promote Memory to Register. https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register.

[26] Range::ursh. https://searchfox.org/mozilla-central/source/js/src/jit/RangeAnalysis.cpp#1026.

[27] Testengineering/performance/Talos/tests. https://wiki.mozilla.org/TestEngineering/Performance/Talos/Tests.

[28] The C++ Standard. https://isocpp.org/std/the-standard.

[29] z3: Bindings for the Z3 Theorem Prover. https://hackage.haskell.org/package/z3.

[30] Bogdan Alexe, Balder ten Cate, Phokion G. Kolaitis, and Wang-Chiew Tan. Designing and refining schema mappings via data examples. SIGMOD '11, 2011.

[31] 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 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[32] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time mee-cbc. In *Revised Selected Papers of the International Conference on Fast Software Encryption*. Springer-Verlag New York, Inc., 2016.

[33] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2016.

[34] Yuan An, Alex Borgida, Renée Miller, and John Mylopoulos. A semantic approach to discovering schema mapping expressions. ICDE'07, 05 2007.

[35] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015.

[36] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[37] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-Independent Sandboxing of Just-in-Time Compilation and Self-Modifying Code. In *PLDI*, 2011.

[38] Apple. JetStream 2. https://browserbench.org/JetStream/.

[39] Apple. SunSpider 1.0.2 JavaScript Benchmark. https://webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.

[40] O. Arden and A. C. Myers. A calculus for flow-limited authorization. CSF'16, 2016.

[41] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. PLAS '13, 2013.

[42] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018.

[43] Saam Barati. Spread's Effects are Modeled Incorrectly Both in AI and in Clobberize. https://bugs.webkit.org/show_bug.cgi?id=181867.

[44] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. https://eprint.iacr.org/2019/1393.

[45] Manuel Barbosa, David Castro, and Paulo F. Silva. Compiling cao: From cryptographic specifications to c implementations. In Martín Abadi and Steve Kremer, editors, *Proceedings of Principles of Security and Trust*. Springer Berlin Heidelberg, 2014.

[46] Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. Type checking cryptography implementations. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*. Springer Berlin Heidelberg, 2012.

[47] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.smt-lib.org.

[48] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *Workshop on Satisfiability Modulo Theories*, 2010.

[49] Aurèle Barrière, Sandrine Blazy, and David Pichardie. Towards Formally Verified Just-in-Time Compilation. In *CoqPL*, 2020.

[50] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[51] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Provably secure compilation of side-channel countermeasures. Cryptology ePrint Archive, Report 2017/1233, 2017. https://eprint.iacr.org/2017/1233.

[52] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *Proceedings of the IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2018.

[53] Heiko Becker, Eva Darulova, Magnus O. Myreen, and Zachary Tatlock. Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler. In *CAV*, 2019.

[54] Heiko Becker, Pavel Panchekha, Eva Darulova, and Zachary Tatlock. Combining Tools for Optimization and Analysis of Floating-Point Computations. In *International Symposium on Formal Methods*, 2018.

[55] Ian Beer. A Very Deep Dive into iOS Exploit Chains Found in the Wild. https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html, August 2019.

[56] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[57] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *Proceedings of the International Workshop on Fast Software Encryption*. Springer, 2005.

[58] Daniel J Bernstein. salsa20-ref.c, 2005.

[59] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Proceedings of the International Workshop on Public Key Cryptography*. Springer, 2006.

[60] Daniel J Bernstein. Writing high-speed software, 2007.

[61] Daniel J Bernstein. The salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 2008.

[62] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. NaCl: Networking and cryptography library, 2016.

[63] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A crypto library in 100 tweets. In *Proceedings of the International Conference on Cryptology and Information Security in Latin America*. Springer, 2014.

[64] Benjamin Beurdouche. Verified cryptography for Firefox 57, 2017.

[65] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Defensive javascript. In *Foundations of Security Analysis and Design VII*. Springer, 2014.

[66] Karthikeyan Bhargavan, Cedric Fournet, and Markulf Kohlweiss. mitls: Verifying Protocol Implementations Against Real-World Attacks. *IEEE Security & Privacy*, 2016.

[67] Andrea Biondo. Exploiting the Math.expm1 typing bug in V8. https://abiondo.me/2019/01/02/exploiting-math-expm1-v8, January 2019.

[68] Dionysus Blazakis. Interpreter exploitation. In *WOOT*, August 2010.

[69] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Proceedings of the European Symposium on Research in Computer Security*. Springer International Publishing, 2017.

[70] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *POPL*, 2014.

[71] Martin Bodin and Alan Schmitt. A Certified JavaScript Interpreter. In *JFLA*, 2013.

[72] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *Symposium on Computer Arithmetic*, 2013.

[73] 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*. USENIX Association, 2017.

[74] Michele Boreale. Quantifying information leakage in process calculi. *Information and Computation*, 207(6), 2009.

[75] Niklas Broberg, Bart van Delft, and David Sands. Paragon–practical programming with information flow control. *Journal of Computer Security*, 25, 2017.

[76] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A Static/Symbolic tool for finding good bugs in good (browser) code. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 199–216. USENIX Association, August 2020.

[77] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5), 2005.

[78] Mykola Bubelich. Js-salsa20, 2017.

[79] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing static and dynamic typing for information-flow control in haskell. ICFP'15, 2015.

[80] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[81] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *Proceedings of the IEEE Cybersecurity Development Conference*. IEEE, 2017.

[82] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *SOSP*, 2017.

[83] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *SOSP*, 2015.

[84] Dmitry Chestnykh. TweetNaCl.js, 2016.

[85] Adam Chlipala. Ur/flow calendar source code.

[86] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. OSDI'10, 2010.

[87] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. SOSP '07, 2007.

[88] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool For Random Testing of Haskell Programs. In *ICFP*, 2000.

[89] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*. ACM, 1995.

[90] Brad Conte. crypto-algorithms, 2012.

[91] Brian J Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. SIGMOD '09, 2009.

[92] Daniel Cousens. pbkdf2, 2014.

[93] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.

[94] Cryptography Coding Standard. Coding rules, 2016.

[95] Jerry Cuomo. Mobile app development, javascript everywhere and "the three amigos". White paper, IBM, 2013.

[96] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22, 2013.

[97] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: The prism workbench. VLDB'08, 2008.

[98] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *TACAS*, 2018.

[99] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. TACAS'08/ETAPS'08, 2008.

[100] Randall Degges. User migration: The definitive guide. https://developer.okta.com/blog/2019/02/15/user-migration-the-definitive-guide, nov 2020.

[101] David Delmas and Jean Souyris. Astrée: from Research to Industry. In *International Static Analysis Symposium*, 2007.

[102] Denis, Frank. libsodium, 2018.

[103] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the Rust Typechecker using CLP. In *ASE*, 2015.

[104] ECMA International. ECMAScript 2018 language specification, 2018.

[105] ECMA ECMAScript, European Computer Manufacturers Association, et al. Ecmascript language specification, 2011.

[106] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[107] Adrienne Felt, Matthew Finifter, Joel Weinberger, and David Wagner. Diesel: Applying privilege separation to database access. ASIACCS '11, 2011.

[108] Jeremy Fetiveau. Attacking TurboFan. TyphoonCon, June 2019. https://doar-e.github.io/presentations/typhooncon2019/AttackingTurboFan_TyphoonCon_2019.pdf.

[109] Jeremy Fetiveau. Circumventing Chrome's Hardening of Typer Bugs. https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/, May 2019.

[110] Jeremy Fetiveau. Introduction to TurboFan. https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/, January 2019.

[111] Richard Forster. *Non-Interference Properties for Nondeterministic Processes*. PhD thesis, University of Cambridge, 1999.

[112] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[113] Galois. Cryptol: The language of cryptography. https://cryptol.net/files/ProgrammingCryptol.pdf, 2016.

[114] Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a Program Logic for JavaScript. In *POPL*, 2012.

[115] Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood. A Trusted Mechanised Specification of JavaScript: One Year On. In *CAV*, 2015.

[116] Ghost. Contributor should not be allowed to edit a post when not being a primary author. https://github.com/TryGhost/Ghost/issues/10238, dec 2018.

[117] Ghost. Fix access to the common article for multiple editors. https://github.com/TryGhost/Ghost/issues/10214, nov 2018.

[118] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. *Journal of Computer Security*, 25, 2017.

[119] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. OSDI'12, 2012.

[120] Georg Gottlob and Pierre Senellart. Schema mapping discovery from data instances. *Journal of the ACM*, 57, 2010.

[121] Matthew Green. The anatomy of a bad idea, 2012.

[122] Samuel Groß. The Art of Exploitation: Attacking JavaScript Engines: A Case Study of JavaScriptCore and CVE-2016-4622. *Phrack*, October 2016. http://phrack.org/papers/attacking_javascript_engines.html.

[123] Samuel Groß. Attacking Client-Side JIT Compilers. Black Hat, August 2018. https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf.

[124] Samuel Groß. FuzzIL: Coverage guided fuzzing for JavaScript engines. Master's thesis, Karlsruhe Institute of Technology, 2018. https://saelo.github.io/papers/thesis.pdf.

[125] Samuel Groß. Fuzzilli: (Guided)-Fuzzing for JavaScript Engines. OffensiveCon 2019, February 2019. https://saelo.github.io/presentations/offensivecon_19_fuzzilli.pdf.

[126] Samuel Groß. JIT Exploitation Tricks. 0x41Con 2019, May 2019. https://saelo.github.io/presentations/41con_19_jit_exploitation_tricks.pdf.

[127] Samuel Groß. JSC Exploits. https://googleprojectzero.blogspot.com/2019/08/jsc-exploits.html, August 2019.

[128] Samuel Groß. The Art of Exploitation: Compile Your Own Type Confusions: Exploiting Logic Bugs in JavaScript JIT Engines. *Phrack*, May 2019. http://phrack.org/papers/jit_exploitation.html.

[129] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. Information-flow control for database-backed applications. EuroS&P'19, 2019.

[130] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.

[131] Shu-yu Guo and Jens Palsberg. The Essence of Compiling with Traces. In *POPL*, 2011.

[132] Shu-yu Guo and Jens Palsberg. The Essence of Compiling with Traces. In *POPL*, 2011.

[133] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017.

[134] Brian Hackett and Shu-yu Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *PLDI*, 2012.

[135] Harry Halpin. The W3C web cryptography API: Design and issues. In *Proceedings of the International Workshop on Web APIs and RESTful design*, 2014.

[136] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *NDSS*, 2019.

[137] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. SAC '14, 2014.

[138] David Herman, Luke Wagner, and Alon Zakai. asm.js, 2014.

[139] Julio C Hernandez and Pedro Isasi. Finding efficient distinguishers for cryptographic mappings, with an application to the block cipher tea. *Computational Intelligence*, 20(3), 2004.

[140] Peter V. Homeier. Quotient types. In *Supplemental Proceedings of the International Conference on Theorem Proving in Higher Order Logics*. University of Edinburgh, 2001.

[141] Seokhie Hong, Deukjo Hong, Youngdai Ko, Donghoon Chang, Wonil Lee, and Sangjin Lee. Differential cryptanalysis of tea and xtea. In *Proceedings of the International Conference on Information Security and Cryptology*. Springer, 2003.

[142] Alicia Hope. Data breach index site leaks over 23,000 hacked databases exposing over 13 billion user records. https://www.cpomagazine.com/cyber-security/data-breach-index-site-leaks-over-23000-hacked-databases-exposing-over-13-billion-user-records/, nov 2020.

[143] Brian Huffman and Ondřej Kunăar. Lifting and transfer: A modular design for quotients in isabelle/hol. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*. Springer-Verlag, 2013.

[144] J. Hughes. Generalising monads to arrows. *Sci. Computer Programming*, 37, 2000.

[145] Troy Hunt. Have i been pwned: Check if your email has been compromised in a data breach. https://haveibeenpwned.com/, nov 2020.

[146] Fedor Indutny. Elliptic, 2014.

[147] Intel. Intel® 64 and IA-32 architectures software developer's manual. *Volume 2: Instruction Set Reference, A-Z*, 2016.

[148] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing Browser Security Guarantees Through Formal Shim Verification. In *USENIX Security*, 2012.

[149] Paul Johnston and Contributors. sha.js, 2017.

[150] Peter Jonsson. Automated testing of database schema migrations. Master's thesis, KTH Royal Institute of Technology, Sweden, 2003.

[151] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In *POPL*, 2015.

[152] Bruno Keith. Attacking Edge through the JavaScript Compiler. OffensiveCon 2019, February 2019. https://github.com/bkth/Attacking-Edge-Through-the-JavaScript-Compiler.

[153] John Kelsey, Bruce Schneier, and David Wagner. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. In *Proceedings of the International Conference on Information and Communications Security*. Springer, 1997.

[154] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal Verification of an OS Kernel. In *SOSP*, 2009.

[155] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *Proceedings of the IEEE European Symposium on Security and Privacy*. IEEE Computer Society, 2017.

[156] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of Advances in Cryptology*. Springer, 1996.

[157] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2017.

[158] Eddie Kohler. Fix critical permissions error, jan 2014.

[159] Eddie Kohler. Minor refactor, jan 2014.

[160] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. PODS '05, 2005.

[161] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *POPL*, 2014.

[162] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving Optimizations Correct Using Parameterized Program Equivalence. In *PLDI*, 2009.

[163] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 427–443, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[164] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010.

[165] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *PLDI*, 2003.

[166] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *POPL*, 2005.

[167] Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *POPL*, 2006.

[168] Peng Li and Steve Zdancewic. Encoding information flow in haskell. CSFW '06, 2006.

[169] Wilson Lian, Hovav Shacham, and Stefan Savage. Too LeJIT to Quit: Extending JIT Spraying to ARM. In *NDSS*, 2015.

[170] Wilson Lian, Hovav Shacham, and Stefan Savage. A Call to ARMs: Understanding the Costs and Benefits of JIT Spraying Mitigations. In *NDSS*, 2017.

[171] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. SOSP'09, 2009.

[172] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. In *PLDI*, 2015.

[173] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *APLAS*, 2008.

[174] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2000.

[175] Philip Martin. Responding to Firefox 0-days in the Wild. https://blog.coinbase.com/responding-to-firefox-0-days-in-the-wild-d9c85a57f15b, August 2019.

[176] J. Mccarthy. Towards a mathematical science of computation. IFIP Congress, 1962.

[177] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. USENIX Security'17, 2017.

[178] David Menendez, Santosh Nagarakatte, and Aarti Gupta. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *International Static Analysis Symposium*, 2016.

[179] Microsoft. Type compatibility - typescript, 2018.

[180] Microsoft. Typescript, 2018.

[181] Renée Miller, Laura Haas, and Mauricio Hernández. Schema mapping as query discovery. VLDB'00, 01 2000.

[182] Max Moroz and Kostya Serebryany. Guided in-process fuzzing of chrome components, Aug 2016.

[183] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified Peephole Optimizations for CompCert. In *PLDI*, 2016.

[184] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1999.

[185] Andrew C. Myers. JFlow: Practical mostly-static information flow control. POPL'99, 1999.

[186] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

[187] Magnus O. Myreen. Verified Just-In-Time Compiler on x86. In *POPL*, 2010.

[188] Amos Ndegwa. What is page load time?, 2016.

[189] Nebulet. Lachlan sneff, 2018.

[190] George C. Necula. Translation Validation for an Optimizing Compiler. In *PLDI*, 2000.

[191] Jaideep Nijjar and Tevfik Bultan. Unbounded data model verification using smt solvers. ASE'12, 2012.

[192] NIST. Secure hash standard. *FIPS PUB 180-2*, 2002.

[193] Ben Niu and Gang Tan. RockJIT: Securing Just-in-Time Compilation using Modular Control-Flow Integrity. In *CCS*, 2014.

[194] Node.js Foundation. Node.js, 2018.

[195] Andres Nötzli and Fraser Brown. LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM. In *SOAP*, 2016.

[196] Open Whisper Systems. Signal protocol library for JavaScript, 2016.

[197] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[198] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference*. Springer, 2006.

[199] D. Page. A note on side-channels resulting from dynamic compilation. In *Cryptology ePrint Archive*, 2006.

[200] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically Improving Accuracy for Floating Point Expressions. In *PLDI*, 2015.

[201] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *PLDI*, 2015.

[202] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. NOJITSU: Locking Down JavaScript Engines. 2020.

[203] James Parker, Niki Vazou, and Michael Hicks. LWeb: Information flow security for multi-tier web applications. POPL'19, 2019.

[204] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *TACAS*, 1998.

[205] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid information flow control. In *ICFP*, 2020.

[206] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *DLS*, 2012.

[207] Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Proving concurrent noninterference. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*. Springer-Verlag, 2012.

[208] Thomas Pornin. Why constant-time crypto?, 2017.

[209] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1), 2003.

[210] Project Everest. Hacl*, a formally verified cryptographic library written in f*, 2018.

[211] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4), 2001.

[212] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. In *CAV*, 2014.

[213] Vignesh S Rao. Writeup for CVE-2019-11707. https://blog.bi0s.in/2019/08/18/Pwn/Browser-Exploitation/cve-2019-11707-writeup/, August 2019.

[214] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security*, 2019.

[215] John Renner. Visit day source code.

[216] John Renner, Sunjay Cauligi, and Deian Stefan. Constant-time webassembly. In *Principles of Secure Compilation*, 2018.

[217] John Renner and Alex Sanchez-Stern. Scooter and sidecar, 2021.

[218] John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, and Deian Stefan. Scooter & sidecar: A domain-specific approach to writing secure database migrations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 710–724, New York, NY, USA, 2021. Association for Computing Machinery.

[219] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2017.

[220] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. pages 199–212, 01 2009.

[221] Chris Rohlf and Yan Ivnitskiy. Attacking Clientside JIT Compilers. Black Hat, August 2011. https://media.blackhat.com/bh-us-11/Rohlf/BH_US_11_RohlfIvnitskiy_Attacking_Client_Side_JIT_Compilers_WP.pdf.

[222] Stephen Röttger. A Guided Tour through Chrome's JavaScript Compiler. Zer0Con, April 2019. https://docs.google.com/presentation/d/1DJcWByz11jLoQyNhmOvkZSrkgcVhllIlCHmal1tGzaw.

[223] Stephen Röttger. Trashing the Flow of Data. https://googleprojectzero.blogspot.com/2019/05/trashing-flow-of-data.html, May 2019.

[224] Alejandro Russo. Functional pearl: Two can keep a secret, if one of them uses haskell. ICFP'15, 2015.

[225] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. Haskell'08, 2008.

[226] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2006.

[227] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2000.

[228] Hanan Samet. *Automatically Proving the Correctness of Translations Involving Optimized Code*. PhD thesis, 1975.

[229] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1), feb 2000.

[230] David Schultz and Barbara Liskov. Ifdb: decentralized information flow control for databases. EuroSys'13, 2013.

[231] Alex Scroxton. Human error blamed in Welsh Covid-19 patient data leak. https://www.computerweekly.com/news/252492123/Human-error-blamed-in-Welsh-Covid-19-patient-data-leak, nov 2020.

[232] Amy Shen. Moved addUsers to policy module, jul 2013.

[233] Boris Shingarov. Formal Verification of JIT by Symbolic Execution. In *VMIL*, 2019.

[234] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button Verification of File Systems via Crash Refinement. In *OSDI*, 2016.

[235] Alexey Sintsov. JIT-Spray Attacks & Advanced Shellcode. *HITBSecConf*, 2010.

[236] E.G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F.B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. SOSP'11, 2011.

[237] Ryan Sleevi. W3c web crypto api update. IETF 86, 2013.

[238] Tyson Smith, Jesse Schwartzentruber, and Sylvestre Ledru. Browser fuzzing at mozilla – mozilla hacks - the web developer blog, Feb 2021.

[239] Axel Souchet. A Journey into IonMonkey: Root-Causing CVE-2019-9810. https://doar-e.github.io/blog/2019/06/17/a-journey-into-ionmonkey-root-causing-cve-2019-9810/, June 2019.

[240] Vincent St-Amour and Shu-yu Guo. Optimization Coaching for JavaScript. In *ECOOP*, 2015.

[241] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. Haskell'11, September 2011.

[242] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *POPL*, 2015.

[243] Dominik Strohmeier and Peter Dolanjski. Comparing browser page load time: An introduction to methodology, 2017.

[244] Torsten Stüber. TweetNacl-WebAssembly, 2017.

[245] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. ASPLOS XI, 2004.

[246] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in Fine. ESOP'10, 2010.

[247] Nikhil Swamy, Brian J Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. SP'08, 2008.

[248] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing Static Analyses for Precision and Soundness. In *CGO*, 2020.

[249] Dominic Tarr. crypto-browserify, 2013.

[250] Ryan Tate. Apple's worst security breach: 114,000 iPad owners exposed. https://gawker.com/5559346/apples-worst-security-breach-114000-ipad-owners-exposed, jun 2010.

[251] Zachary Tatlock and Sorin Lerner. Bringing Extensibility to Verified Compilers. In *PLDI*, 2010.

[252] BIBIFI Team. Bibifi source code.

[253] Hails Team. Gitstar source code.

[254] Hails Team. Lambdachair source code.

[255] Hails Team. Learnbyhacking source code.

[256] Lifty Team. Lifty conference source code.

[257] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe Haskell. In *ACM SIGPLAN Notices*, volume 47. ACM, 2012.

[258] Luca Todesco. A Few JSC Tales. Objective by the Sea, June 2019. http://iokit.racing/jsctales.pdf.

[259] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[260] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. MAC: a verified static information-flow control library. *Journal of logical and algebraic methods in programming*, 95, 2018.

[261] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3), 1996.

[262] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM*.

[263] Nan Wang. V8 exploit. http://eternalsakura13.com/2018/05/06/v8/, May 2018.

[264] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *OSDI*, 2014.

[265] Yuepeng Wang, Isil Dillig, Shuvendu K Lahiri, and William R Cook. Verifying equivalence of database-driven applications. POPL'17, 2017.

[266] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. Data migration using datalog program synthesis. VLDB'20, 2020.

[267] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 2018.

[268] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven secure cryptography for the web ecosystem, 2018.

[269] WebAssembly Community Group. Module instances, 2018.

[270] WebAssembly Community Group. reference-types, 2018.

[271] WebAssembly Community Group. Webassembly, 2018.

[272] WebAssembly Community Group. Webassembly, 2018.

[273] David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. In *Lecture Notes in Computer Science*. Springer, 1994.

[274] Allen Wirfs-Brock. ECMAScript 2015 Language Specification – Math.expm1(x). https://www.ecma-international.org/ecma-262/6.0/#sec-math.expm1, 2015.

[275] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.

[276] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. PLDI'16, 2016.

[277] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. POPL'12, 2012.

[278] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, 2011.

[279] Mark Vincent Yason. Understanding the Attack Surface and Attack Resilience of Project Spartan's (Edge) New EdgeHTML Rendering Engine. Black Hat, August 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Yason-Understanding-The-Attack-Surface-And-Attack-Resilience-Of-Project-Spartans-New-EdgeHTML-Rendering-Engine-wp.pdf.

[280] 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 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009.

[281] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. SOSP'09, 2009.

[282] Alon Zakai. Compiling to WebAssembly: It's Happening!, 2015.

[283] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3), 2002.

[284] Google Project Zero. 0day "in the wild".

[285] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL*, 2012.

[286] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formal Verification of SSA-Based Optimizations for LLVM. In *PLDI*, 2013.

[287] Qixun Zhao. Story1 Mom What Is Zero Multiplied By Infinity. https://blogs.projectmoon.pw/2019/01/13/Story1-Mom-What-Is-Zero-Multiplied-By-Infinity, January 2019.

[288] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[289] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *CCS*, 2017.