View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Apollo

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

CONRAD WATT, University of Cambridge, UK
JOHN RENNER, University of California San Diego, USA
NATALIE POPESCU, University of California San Diego, USA
SUNJAY CAULIGI, University of California San Diego, USA
DEIAN STEFAN, University of California San Diego, USA

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.

CCS Concepts: • **Security and privacy** → **Cryptography**; *Formal methods and theory of security*; *Web application security*; • **Theory of computation** → *Operational semantics*; • **Software and its engineering** → *Assembly languages*;

Additional Key Words and Phrases: WebAssembly, cryptography, constant-time, information flow control

ACM Reference Format:
Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.* 3, POPL, Article 77 (January 2019), 29 pages. https://doi.org/10.1145/3290390

Authors' addresses: Conrad Watt, University of Cambridge, UK, conrad.watt@cl.cam.ac.uk; John Renner, University of California San Diego, USA, jmrenner@eng.ucsd.edu; Natalie Popescu, University of California San Diego, USA, npopescu@ucsd.edu; Sunjay Cauligi, University of California San Diego, USA, scauligi@eng.ucsd.edu; Deian Stefan, University of California San Diego, USA, deian@cs.ucsd.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/1-ART77
https://doi.org/10.1145/3290390

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 [Oren et al. 2015; Page 2006; Van Goethem et al. 2015]. Even worse, much of the JavaScript cryptography used in the wild is implemented by "unskilled cryptographers" [Sleevi 2013] 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 [Cuomo 2013] 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 [Chestnykh 2016; Cousens 2014; Indutny 2014; Tarr 2013]), 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) [Haas et al. 2017; WebAssembly Community Group 2018c].

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 [Haas et al. 2017]; well-typed Wasm programs enjoy standard *progress* and *preservation* properties [Wright and Felleisen 1994], which have even been mechanically verified [Watt 2018]. These formal foundations are a crucial first step towards developing in-browser crypto with guarantees of security.

In this paper, 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* [Sabelfeld and Myers 2006] and *constant-time* programming disciplines [Barthe et al. 2014; Pornin 2017] 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 [Barthe et al. 2014]. 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 paper 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 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 [Bernstein et al. 2014]. 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 [Microsoft 2018b], 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 [Watt et al. 2018].

Paper Organization. We first review WebAssembly and the constant-time programming paradigm (Section 2) and give a brief overview of CT-Wasm (Section 3). In Section 4 we describe the CT-Wasm language and its semantics. Our mechanized model and formal security guarantees are detailed in Section 5. We describe our implementations, supporting tools, and evaluation in Sections 6. We review related work in Section 7. Finally we discuss future work in Section 8 and conclude.

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.

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 [Kocher 1996], 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 [Brumley and Boneh 2005] and AES [Bernstein 2005a; Osvik et al. 2006]). 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 [Bernstein 2005b, 2006, 2008].

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 [Cryptography Coding Standard 2016; Pornin 2017].

Verifying the constant-time property (or detecting lack thereof) for a given implementation is considered one of the most important verification problems in cryptography [Almeida et al. 2017, 2016a,b; Blazy et al. 2017; Bond et al. 2017; Cauligi et al. 2017; Erbsen et al. 2019; Zinzindohoué et al. 2017]. To facilitate formal reasoning, these works typically represent this constant-time property using a *leakage model* [Boreale 2009] 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 [Almeida et al. 2016b]. 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 [Almeida et al. 2016b]. 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 [Zinzindohoué et al. 2017] has been adopted by Firefox [Beurdouche 2017], while Fiat [Erbsen et al. 2019] 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 [ECMA International 2018]. Even if we restrict ourselves to well-behaving subsets of JavaScript (e.g., asm.js [Herman et al. 2014] or defensive JavaScript [Bhargavan et al. 2014]), 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 [Chestnykh 2016; Cousens 2014; Indutny 2014; Kobeissi et al. 2017; Open Whisper Systems 2016; Tarr 2013], even in the presence of native libraries which were intended to curb their use in the web ecosystem [Halpin 2014]. Unfortunately, these competing solutions proved inadequate. Native crypto libraries differ wildly across platforms. For example the Web Crypto [Halpin 2014] and the Node.js crypto [Node.js Foundation 2018] 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*" [Green 2012]) when compared to state-of-the-art libraries like NaCl [Bernstein et al. 2016]. And, worst of all, none of these native libraries implement modern cryptographic algorithms such as the Poly1305 Message Authentication Code [Bernstein 2005b], a default in modern crypto libraries [Bernstein et al. 2016]). As we argue in this paper, 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.

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 [Yee et al. 2009]) or subsets of JavaScript (e.g., asm.js [Herman et al. 2014]). 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 [Haas et al. 2017] 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 [Terei et al. 2012], 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 [Haas et al. 2017]. Together, these ensure that well-typed WebAssembly programs are "safe", i.e., they satisfy progress and preservation [Watt 2018; Wright and Felleisen 1994]. 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 [Haas et al. 2017]. 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 [Barthe et al. 2014].

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 [Barthe et al. 2014; Sabelfeld and Myers 2006; Volpano et al. 1996]. In the next section, we give an overview of our extension: CT-Wasm.

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 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 [Ndegwa 2016; Strohmeier and Dolanjski 2017], 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 [Almeida et al. 2016b; Barthe et al. 2014], 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 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 6.3.1).

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 [Myers 1999; Myers et al. 2001] or FlowCaml's [Pottier and Simonet 2003]). 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 [Cryptography Coding Standard 2016; Pornin 2017]: 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 Fig. 1 shows, to reveal (and, 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.

```
(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)))
```

Fig. 1. Verified decryption in TweetNaCl relies on a declassification to terminate early (if verification fails).

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

$$
\begin{array}{lll}
unop_{\text{i}N} & ::= & \textbf{clz} \mid \textbf{ctz} \mid \textbf{popcnt} \\
unop_{\text{f}N} & ::= & \textbf{neg} \mid \textbf{abs} \mid \textbf{ceil} \mid \textbf{floor} \mid \\
 & & \textbf{trunc} \mid \textbf{nearest} \mid \textbf{sqrt} \\
binop_{\text{i}N} & ::= & \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div_}sx \mid \\
 & & \textbf{rem_}sx \mid \textbf{and} \mid \textbf{or} \mid \textbf{xor} \mid \\
 & & \textbf{shl} \mid \textbf{shr_}sx \mid \textbf{rotl} \mid \textbf{rotr} \\
binop_{\text{f}N} & ::= & \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div} \mid \\
 & & \textbf{min} \mid \textbf{max} \mid \textbf{copysign} \\
testop_{\text{i}N} & ::= & \textbf{eqz} \\
relop_{\text{i}N} & ::= & \textbf{eq} \mid \textbf{ne} \mid \textbf{lt_}sx \mid \textbf{gt_}sx \mid \\
 & & \textbf{le_}sx \mid \textbf{ge_}sx \\
relop_{\text{f}N} & ::= & \textbf{eq} \mid \textbf{ne} \mid \textbf{lt} \mid \textbf{gt} \mid \textbf{le} \mid \textbf{ge} \\
cvtop & ::= & \textbf{convert} \mid \textbf{reinterpret} \mid \\
 & & \textbf{classify} \mid \textbf{declassify} \\
sx & ::= & \textbf{s} \mid \textbf{u}
\end{array}
$$

$$
\begin{array}{llll}
\text{(constants)} & k & ::= & \dots \\
\text{(instructions)} & e & ::= & \textbf{unreachable} \mid \textbf{nop} \mid \textbf{drop} \mid \textbf{select}\ sec \mid \\
 & & & \textbf{block}\ ft\ e^*\ \textbf{end} \mid \textbf{loop}\ ft\ e^*\ \textbf{end} \mid \\
 & & & \textbf{if}\ ft\ e^*\ \textbf{else}\ e^*\ \textbf{end} \mid t.\textbf{const}\ k \mid \\
 & & & \textbf{br}\ imm \mid \textbf{br_if}\ imm \mid \textbf{br_table}\ imm^+ \mid \\
 & & & \textbf{return} \mid \textbf{call}\ imm \mid \textbf{call_indirect}\ (tr, ft) \mid \\
 & & & \textbf{get_local}\ imm \mid \textbf{set_local}\ imm \mid \\
 & & & \textbf{tee_local}\ imm \mid \textbf{get_global}\ imm \mid \\
 & & & \textbf{set_global}\ imm \mid \\
 & & & t.\textbf{load}\ (pt_sx)^?\ a\ o \mid t.\textbf{store}\ pt^?\ a\ o \mid \\
 & & & \textbf{memory.size} \mid \textbf{memory.grow} \mid \\
 & & & t.unop_t \mid t.binop_t \mid t.testop_t \mid \\
 & & & t.relop_t \mid t.cvtop\ t_sx^?
\end{array}
$$

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

$$
\begin{array}{llll}
\text{sec}\ (\text{i}N'\ sec) \triangleq sec & & \text{i}N & ::= \text{i}N'\ \text{public} \\
\text{sec}\ \text{f}N \triangleq \text{public} & & \text{s}N & ::= \text{i}N'\ \text{secret}
\end{array}
$$

Fig. 2. CT-Wasm abstract syntax as an extension (highlighted) of the grammar given by [Haas et al. 2017].

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 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 will occur.

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 Fig. 2, the core type system in Fig. 3, and an illustrative selection of the runtime reduction rules in Fig. 4.

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

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

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

$$\frac{}{C \vdash t.\mathbf{const}\ c : \epsilon \to t} \qquad \frac{}{C \vdash t.unop : t \to t} \qquad \frac{}{C \vdash t.binop : t\ t \to t}$$

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

$$\frac{t_1 \neq t_2 \qquad sx^? = \epsilon \Leftrightarrow (t_1 = \mathbf{in}_1'\ sec \land t_2 = \mathbf{in}_2'\ sec \land |t_1| < |t_2|) \lor (t_1 = \mathbf{f}n \land t_2 = \mathbf{f}n')}{C \vdash t_1.\mathbf{convert}\ t_2_sx^? : t_2 \to t_1}$$

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

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

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

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

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

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

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

$$\frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{get_local}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{set_local}\ i : t \to \epsilon} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{tee_local}\ i : t \to t}$$

$$\frac{C_{\mathsf{global}}(i) = \mathsf{mut}^?\ t}{C \vdash \mathbf{get_global}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{global}}(i) = \mathsf{mut}\ t}{C \vdash \mathbf{set_global}\ i : t \to \epsilon}$$

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

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

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

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

Fig. 3. CT-Wasm typing rules as an extension (highlighted) of the typing rules given by [Haas et al. 2017].

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

$$s; vs;\ (\text{s}N.\mathbf{const}\ k)\ t_2.\mathbf{declassify}\ t_1 \rightsquigarrow_i s; vs;\ (\text{i}N.\mathbf{const}\ k)$$

$$s; vs;\ (\text{i}N.\mathbf{const}\ k)\ t_2.\mathbf{classify}\ t_1 \rightsquigarrow_i s; vs;\ (\text{s}N.\mathbf{const}\ k)$$

$$s; vs;\ v_1\ v_2\ ((\mathbf{i32}'\ sec).\mathbf{const}\ 0)\ \mathbf{select}\ sec' \rightsquigarrow_i s; vs; v_2$$

$$s; vs;\ v_1\ v_2\ ((\mathbf{i32}'\ sec).\mathbf{const}\ k+1)\ \mathbf{select}\ sec' \rightsquigarrow_i s; vs; v_1$$

$$s; vs;\ (\mathbf{i32.const}\ k)\ \mathbf{call_indirect}\ (tr, ft) \rightsquigarrow_i s; vs; \mathbf{callcl}\ cl \qquad \begin{array}{l}\text{if table_i}\ ((\text{inst}\ s)!i) = a\\ \text{and}\ ((\text{table i})!a)!k = cl\\ \text{and type}\ cl = (tr, ft)\end{array} \quad (*)$$

$$s; vs;\ (\mathbf{i32.const}\ k)\ \mathbf{call_indirect}\ (tr, ft) \rightsquigarrow_i s; vs; \mathbf{trap} \qquad \text{otherwise}$$

(∗) **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 [Haas et al. 2017].

Fig. 4. Selected CT-Wasm semantic definitions, extended (highlighted) from [Haas et al. 2017]. Since the vast majority of the reduction rules are unchanged, we give only a few examples here.

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 Fig. 3, the typing context C abstracts the *current instance*. In Fig. 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.

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 $t^* \rightarrow t'^*$, 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.

[1] In [Haas et al. 2017], 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 [WebAssembly Community Group 2018a]. 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".

Base Wasm has four value types: i32, i64, f32, and f64, representing 32 and 64 bit integer 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 [Andrysco et al. 2015, 2018; Kohlbrenner and Shacham 2017].

As shown in Fig. 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 5.2.

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 [Cryptography Coding Standard 2016], and as discussed throughout this paper, 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 Fig. 4. Importantly, **select** can do this without branching: conditional move instructions allow **select** to be implemented using a single, constant-time hardware instruction [Intel 2016] and, for processors without such instructions a multi-instruction arithmetic solution exists [Cryptography Coding Standard 2016]. 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.

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 [Brumley and Boneh 2005; Osvik et al. 2006], 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 [WebAssembly Community Group 2018b], 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 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).

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 Fig. 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 6.4.

To restrict the use of **declassify**, as Fig. 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 5.8).

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 [Watt 2018], 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.

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

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 [Watt et al. 2018]. 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.

$$t_1.\mathbf{const}\ k_1 \sim_{\mathbf{v}} t_2.\mathbf{const}\ k_2 \triangleq t_1 = t_2 \wedge (k_1 = k_2 \vee \text{sec}\ t_1 = \text{sec}\ t_2 = \text{secret})$$

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

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

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

Fig. 5. Definition of $\sim_{\mathbf{c}}$.

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 Fig. 5. This relation is required for the expression of the constant-time property, and mirrors the equivalence relation used for the same purpose by [Barthe et al. 2014] between program states. We prove that typeability of a WebAssembly configuration is invariant with respect to $\sim_{\mathbf{c}}$.

Lemma 5.3.1 (equivp_config_indistinguishable)**.**
$\sim_{\mathbf{c}}$ is an equivalence relation.

Lemma 5.3.2 (config_indistinguishable_imp_config_typing)**.**
If $\vdash_i c : (tr, ts)$, then for all c' such that $c \sim_{\mathbf{c}} c'$, $\vdash_i c' : (tr, ts)$.

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 [Almeida et al. 2016b], 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.

$$s; vs;\ (\text{s32.}\mathbf{const}\ k_1)\ (\text{s32.}\mathbf{const}\ k_2)\ \text{s32.}binop \quad \overset{a}{\rightsquigarrow}_i \quad s; vs;\ (\text{s32.}\mathbf{const}\ (k_1\ binop\ k_2))$$
$$\text{with } a \quad \triangleq \quad \text{binop_action}(binop, (\text{s32.}\mathbf{const}\ k_1), (\text{s32.}\mathbf{const}\ k_2))$$

$$s; vs;\ v_1^n\ \mathbf{callcl}\ cl \quad \overset{a}{\rightsquigarrow}_i \quad s'; vs';\ v_2^m \quad (*)$$
$$\text{with } cl \quad \triangleq \quad \{\text{type } (tr, t_1^n \rightarrow t_2^m), \text{host } \ldots\}$$
$$a \quad \triangleq \quad \text{host_action}(s, v_1^n, s', v_1^m, cl)$$

$$a_1 \sim_{\mathbf{a}} a_2 \triangleq \begin{cases} op_1 = op_2 & \text{if} \quad \begin{array}{l} a_1 = \text{binop_action}(op_1, v_1, v_2) \\ a_2 = \text{binop_action}(op_2, v_1', v_2') \end{array} \quad \text{and} \quad \text{is_safe_binop}(op_1) \\ \begin{array}{l} s_1 \sim_{\mathbf{s}} s_2 \\ \land (v_1 \sim_{\mathbf{v}} v_2)^n \\ \land s_1' \sim_{\mathbf{s}} s_2' \\ \land (v_{1'} \sim_{\mathbf{v}} v_{2'})^m \\ \land cl_1 = cl_2 \end{array} & \text{if} \quad \begin{array}{l} a_1 = \text{host_action}(s_1, v_1^n, s', v_{1'}^m, cl_1) \\ a_2 = \text{host_action}(s_2, v_2^n, s_2', v_{2'}^m, cl_2) \end{array} \quad \text{and} \quad \text{trust}(cl_1) = \text{untrusted} \\ \ldots & \\ a_1 = a_2 & \text{otherwise} \end{cases}$$

(∗) The full axiomatic description of host function behavior is not reproduced here. Full details can be found in [Watt 2018], or in our mechanization.

Fig. 6. Example of CT-Wasm action annotations and equivalence.

We augment the WebAssembly reduction relation with state-parameterized actions, as $c \overset{a}{\rightsquigarrow}_i c'$, effectively defining a labelled transition system. Traditionally, a constant-time proof in the style of [Barthe et al. 2014] 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 [Sabelfeld and Myers 2006], which are used to embody an attacker's view of a system. An illustration of our definitions can be found in Fig. 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 5.9.

Lemma 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) 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 [Forster 1999; Mantel 2000]. A formal statement and proofs related to this generalized definition will follow in Section 5.8.

Definition (sketch) 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.

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 [Popescu et al. 2012], 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 5.5.1 (config_indistinguishable_imp_reduce)**.**
If $\vdash_i c : (\text{untrusted}, ts)$ 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.

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* [Sabelfeld and Sands 2000]. Configurations in WebAssembly always reduce with respect to an instance, so we define bisimulation in terms of configurations together with their instances.

Definition 5.6.1 (config_bisimulation**).**
config_bisimulation R $\triangleq$
$\forall((c, i), (c', i')) \in$ 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 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 5.6.3 (typed_indistinguishable_pairs**).**
typed_indistinguishable_pairs $\triangleq$ { $((c, i), (c', i))$ | $\vdash_i c :$ (untrusted, ts) $\wedge\ c \sim_{\mathbf{c}} c'$ }

Lemma 5.6.4 (config_bisimulation_typed_indistinguishable_pairs**).**
config_bisimulation typed_indistinguishable_pairs

Theorem 5.6.5 (config_indistinguishable_imp_config_bisimilar**).**
If $\vdash_i c :$ (untrusted, ts) *for some ts, then for all* c' *such that* $c \sim_{\mathbf{c}} c'$, $((c, i), (c', i)) \in$ config_bisimilar..

5.7 Non-interference

We define a reflexive, transitive version of our reduction relation, given as $c \overset{as}{\leadsto}{}^*_i c_{as}$, 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 5.7.1 (config_indistinguishable_trace_noninterference**).**
If $\vdash_i c :$ (untrusted, ts) *for some ts, then for all* c' *such that* $c \sim_{\mathbf{c}} c'$*, if* $c \overset{as}{\leadsto}{}^*_i c_{as}$ *then there exists* c'_{as} *and* as' *such that* $c' \overset{as'}{\leadsto}{}^*_i c'_{as}$ *and* $c_{as} \sim_{\mathbf{c}} c'_{as}$ *and as pairwise* $\sim_{\mathbf{a}}$ *with* as'*.*

5.8 Constant-time

We now formally discuss the constant-time property we originally sketched (Section 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 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 5.8.2 (config_trace_set).
config_trace_set $(c, i) \triangleq \{\ as \mid$ config_is_trace $(c, i)\ as\ \}$

Definition 5.8.3 (rel_set).
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 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 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 5.8.6 (config_untrusted_constant_time_traces).
If $\vdash_i c : (\text{untrusted}, ts)$ *for some ts, then (c,i) is constant-time.*

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 [Homeier 2001]. 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 [Huffman and Kunăar 2013]. 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 5.9.1 (observation).
observation $\triangleq$ *action llist* / (llist_all2 $\sim_{\mathbf{a}}$)

Lemma 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 5.9.3 (constant_time).
constant_time $(c, i) \triangleq \forall c'.\ c \sim_{\mathbf{c}} c' \longrightarrow$ config_obs_set $(c, i) =$ config_obs_set (c', i)

We additionally give weaker versions of all of the results in 5.8 and 5.9 using a stronger definition of config_is_trace that is inductive rather than coinductive in [Watt et al. 2018].

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 [Watt et al. 2018].

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.

6.1.1 Reference implementation. We extend the Wasm reference interpreter [WebAssembly Community Group 2018d] 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.

6.1.2 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" [Click and Paleczny 1995] 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.

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 [Watt 2018], 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.

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.

6.3.1 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 [Microsoft 2018b]. 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 6.1.2, the **select** instruction is not always compiled as constant-time. Like the TypeScript compiler [Microsoft 2018a], 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 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 [Nebulet 2018]) that do not rely on a JavaScript host to execute.

6.3.2 wasm2ct. CT-Wasm is a useful low-level language for implementing cryptographic algorithms from the start, much like qhasm [Bernstein 2007] and Jasmin [Almeida et al. 2017]. 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 development of a more sophisticated tool (e.g., based on symbolic execution [Baldoni et al. 2018]) that can precisely reason about memory—at least for crypto implementations—to future work.

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?

[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.

To answer these questions, we manually implement three cryptographic algorithms in CT-Wasm: the Salsa20 stream cipher [Bernstein 2008], the SHA-256 hash function [NIST 2002], and the TEA block cipher [Wheeler and Needham 1994].[3] Following [Barthe et al. 2014], 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 [Bernstein 2005c; Conte 2012; Wheeler and Needham 1994]. For both Salsa20 and TEA, we label keys and messages as secret; for SHA-256, like [Barthe et al. 2014], we treat the input message as secret.

Beyond these manual implementations, we also port an existing Wasm implementation of the TweetNaCl library [Bernstein et al. 2014; Stüber 2017]. This library implements the full NaCl API [Bernstein et al. 2016], 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.

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 [Bubelich 2017; Johnston and Contributors 2017]. 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 [Wheeler and Needham 1994]. Finally, for TweetNaCl, we use the Wasm library's test suite [Stüber 2017].

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.

6.4.1 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).

[3] TEA and several variants of the block cipher are vulnerable and should not be used in practice [Hernandez and Isasi 2004; Hong et al. 2003; Kelsey et al. 1997]. We only implement TEA to evaluate our language as a measure of comparison with [Barthe et al. 2014].

	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

Fig. 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.

The TweetNaCl library requires a single **declassify** instruction, in the crypto_secretbox API—the API that implements secret-key authenticated encryption. As shown in Fig. 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 [Bernstein et al. 2016]. 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.

6.4.2 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 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 [Myers et al. 2001].

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. Fig. 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 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.

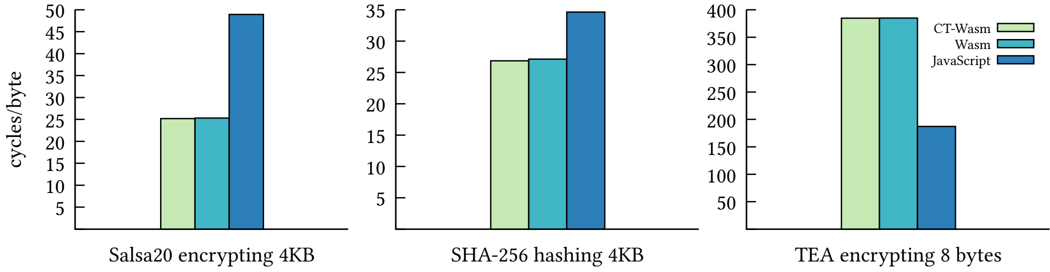


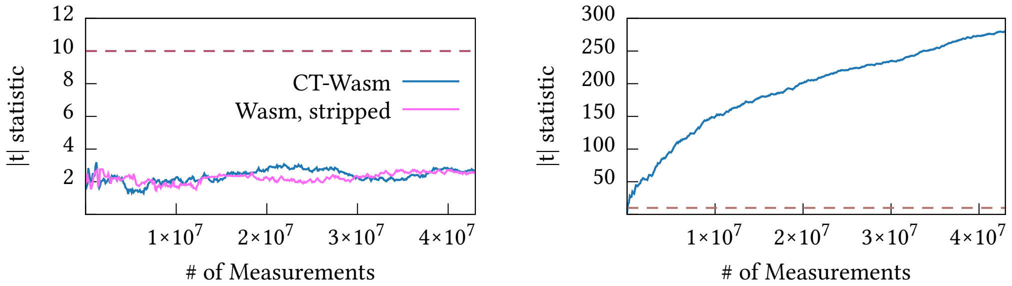
Fig. 8. Runtime performance of handwritten crypto primitives.

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. Fig. 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.

6.4.3 Security. To empirically evaluate the security of our implementations, we run a modified version of dudect [Reparaz et al. 2017]. 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 [Reparaz et al. 2017], 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 Fig. 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.

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 Fig. 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).



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

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

Fig. 9. dudect measurements for various cryptographic algorithms.

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.

7 RELATED WORK

An initial high-level design for CT-Wasm, which this work entirely supersedes, has been previously described [Renner et al. 2018].

Low-level crypto DSLs. Bernstein's qhasm [Bernstein 2007] 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 [Barbosa et al. 2012] and Cryptol [Galois 2016] are higher-level DSLs for crypto implementations, but do not have verified non-interference guarantees.

Vale [Bond et al. 2017] and Jasmin [Almeida et al. 2017] 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 [Leino 2010]. 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* [Zinzindohoué et al. 2017] 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 [Project Everest 2018]. FaCT [Cauligi et al. 2017] is a high-level language that compiles to LLVM which it then verifies with ct-verif [Almeida et al. 2016b]. CAO [Barbosa et al. 2014, 2012] and Cryptol [Galois 2016] 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 [Cryptography Coding Standard 2016; Reparaz et al. 2017] and theoretical [Almeida et al. 2016b; Barthe et al. 2014; Blazy et al. 2017]. 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 [Sabelfeld and Myers 2006] 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 [Barthe et al. 2014].

Our type system—which facilitates the non-interference result—can be characterized as a specialization of the Volpano-Irvine-Smith security type system [Volpano et al. 1996]. Our equivalence relation-based observations are similar to the abstractions used by [Barthe et al. 2017, 2018]. 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 [Sabelfeld and Myers 2006] 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. [Popescu et al. 2012] give a coinductive treatment of the non-interference property, but for an idealized language, and do not connect it to the constant-time property.

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* [Zinzindohoué et al. 2017] and FaCT [Cauligi et al. 2017]) 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 [Denis, Frank 2018] to Wasm (e.g., with Emscripten [Zakai 2015]) and use wasm2ct to ensure they are secure.

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.

REFERENCES

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

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016a. 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.

Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016b. Verifying Constant-Time Implementations. In *Proceedings of the USENIX Security Symposium*. USENIX Association.

Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. 2018. Towards Verified, Constant-time Floating Point Operations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (2018).

Manuel Barbosa, David Castro, and Paulo F. Silva. 2014. Compiling CAO: From Cryptographic Specifications to C Implementations. In *Proceedings of Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). Springer Berlin Heidelberg.

Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. 2012. Type Checking Cryptography Implementations. In *Fundamentals of Software Engineering*, Farhad Arbab and Marjan Sirjani (Eds.). Springer Berlin Heidelberg.

Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM.

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

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *Proceedings of the IEEE Computer Security Foundations Symposium*. IEEE Computer Society.

Daniel J Bernstein. 2005a. Cache-timing attacks on AES. (2005).

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

Daniel J Bernstein. 2005c. salsa20-ref.c. (2005). Retrieved July 5, 2018 from https://cr.yp.to/snuffle/salsa20/ref/salsa20.c

Daniel J Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *Proceedings of the International Workshop on Public Key Cryptography*. Springer.

Daniel J Bernstein. 2007. Writing high-speed software. (2007). Retrieved June 11, 2018 from http://cr.yp.to/qhasm.html

Daniel J Bernstein. 2008. The Salsa20 family of stream ciphers. In *New stream cipher designs*. Springer.

Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2016. NaCl: Networking and cryptography library. (2016). Retrieved June 23, 2018 from https://nacl.cr.yp.to

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

Benjamin Beurdouche. 2017. Verified cryptography for Firefox 57. (2017). https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/

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

Sandrine Blazy, David Pichardie, and Alix Trieu. 2017. Verifying Constant-Time Implementations by Abstract Interpretation. In *Proceedings of the European Symposium on Research in Computer Security*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing.

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the USENIX Security Symposium*. USENIX Association.

Michele Boreale. 2009. Quantifying Information Leakage in Process Calculi. *Information and Computation* 207, 6 (2009).

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

Mykola Bubelich. 2017. JS-Salsa20. (2017). Retrieved July 5, 2018 from https://www.npmjs.com/package/js-salsa20

Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *Proceedings of the IEEE Cybersecurity Development Conference*. IEEE.

Dmitry Chestnykh. 2016. TweetNaCl.js. (2016). Retrieved June 23, 2018 from https://www.npmjs.com/package/tweetnacl

Cliff Click and Michael Paleczny. 1995. A Simple Graph-based Intermediate Representation. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*. ACM.

Brad Conte. 2012. crypto-algorithms. (2012). Retrieved July 5, 2018 from https://github.com/B-Con/crypto-algorithms

Daniel Cousens. 2014. pbkdf2. (2014). Retrieved June 23, 2018 from https://www.npmjs.com/package/pbkdf2

Cryptography Coding Standard. 2016. Coding rules. (2016). Retrieved June 11, 2018 from https://cryptocoding.net/index.php/Coding_rules

Jerry Cuomo. 2013. Mobile app development, JavaScript everywhere and "the three amigos". (2013). White paper, IBM.

Denis, Frank. 2018. libsodium. (2018). Retrieved July 12, 2018 from https://github.com/jedisct1/libsodium

ECMA International. 2018. ECMAScript 2018 Language Specification. (2018). https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Richard Forster. 1999. *Non-Interference Properties for Nondeterministic Processes*. Ph.D. Dissertation. University of Cambridge.

Galois. 2016. Cryptol: The Language of Cryptography. https://cryptol.net/files/ProgrammingCryptol.pdf. (2016).

Matthew Green. 2012. The anatomy of a bad idea. (2012). Retrieved June 23, 2018 from https://blog.cryptographyengineering.com/2012/12/28/the-anatomy-of-bad-idea/

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

Harry Halpin. 2014. The W3C Web Cryptography API: Design and Issues. In *Proceedings of the International Workshop on Web APIs and RESTful design*.

David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js. (2014). Retrieved October 7, 2017 from http://asmjs.org/spec/latest

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

Peter V. Homeier. 2001. Quotient Types. In *Supplemental Proceedings of the International Conference on Theorem Proving in Higher Order Logics*. University of Edinburgh.

Seokhie Hong, Deukjo Hong, Youngdai Ko, Donghoon Chang, Wonil Lee, and Sangjin Lee. 2003. Differential Cryptanalysis of TEA and XTEA. In *Proceedings of the International Conference on Information Security and Cryptology*. Springer.

Brian Huffman and Ondřej Kunăar. 2013. 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.

Fedor Indutny. 2014. Elliptic. (2014). Retrieved June 23, 2018 from https://www.npmjs.com/package/elliptic

Intel. 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 2: Instruction Set Reference, A-Z* (2016).

Paul Johnston and Contributors. 2017. sha.js. (2017). Retrieved July 5, 2018 from https://www.npmjs.com/package/sha.js

John Kelsey, Bruce Schneier, and David Wagner. 1997. 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.

Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. 2017. 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.

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

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

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

Heiko Mantel. 2000. Possibilistic Definitions of Security - An Assembly Kit. In *Proceedings of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society.

Microsoft. 2018a. Type Compatibility - TypeScript. (2018). Retrieved June 11, 2018 from https://www.typescriptlang.org/docs/handbook/type-compatibility.html

Microsoft. 2018b. TypeScript. (2018). Retrieved June 11, 2018 from https://www.typescriptlang.org/

Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2001. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif* 2005 (2001).

Amos Ndegwa. 2016. What is Page Load Time? (2016). Retrieved June 11, 2018 from https://www.maxcdn.com/one/visual-glossary/page-load-time/

Nebulet. 2018. Lachlan Sneff. (2018). Retrieved July 4, 2018 from https://github.com/nebulet/nebulet

NIST. 2002. Secure Hash Standard. *FIPS PUB 180-2* (2002).

Node.js Foundation. 2018. Node.js. (2018). Retrieved June 11, 2018 from https://nodejs.org/docs/latest-v10.x/api/crypto.html

Open Whisper Systems. 2016. Signal Protocol library for JavaScript. (2016). Retrieved June 23, 2018 from https://github.com/signalapp/libsignal-protocol-javascript

Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. 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.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference*. Springer.

D. Page. 2006. A Note On Side-Channels Resulting From Dynamic Compilation. In *Cryptology ePrint Archive*. https://eprint.iacr.org/2006/349

Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. 2012. Proving Concurrent Noninterference. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*. Springer-Verlag.

Thomas Pornin. 2017. Why Constant-Time Crypto? (2017). Retrieved June 11, 2018 from https://www.bearssl.org/constanttime.html

François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems* 25, 1 (2003).

Project Everest. 2018. HACL*, a formally verified cryptographic library written in F*. (2018). Retrieved July 12, 2018 from https://github.com/project-everest/hacl-star

John Renner, Sunjay Cauligi, and Deian Stefan. 2018. Constant-Time WebAssembly. In *Principles of Secure Compilation*.

Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, is My Code Constant Time?. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association.

A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2006).

Andrei Sabelfeld and David Sands. 2000. Probabilistic Noninterference for Multi-Threaded Programs. In *Proceedings of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society.

Ryan Sleevi. 2013. W3C Web Crypto API Update. (2013). https://datatracker.ietf.org/meeting/86/materials/slides-86-saag-5 IETF 86.

Dominik Strohmeier and Peter Dolanjski. 2017. Comparing Browser Page Load Time: An Introduction to Methodology. (2017). Retrieved June 11, 2018 from https://hacks.mozilla.org/2017/11/comparing-browser-page-load-time-an-introduction-to-methodology/

Torsten Stüber. 2017. TweetNacl-WebAssembly. (2017). https://github.com/TorstenStueber/TweetNacl-WebAssembly

Dominic Tarr. 2013. crypto-browserify. (2013). Retrieved June 23, 2018 from https://www.npmjs.com/package/crypto-browserify

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM.

Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2-3 (1996).

Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM.

Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2018. CT-Wasm: Type-driven Secure Cryptography for the Web Ecosystem. (2018). http://ct-wasm.programming.systems/

WebAssembly Community Group. 2018a. Module Instances. (2018). Retrieved June 11, 2018 from https://webassembly.github.io/spec/core/exec/runtime.html?highlight=instance#module-instances

WebAssembly Community Group. 2018b. reference-types. (2018). Retrieved June 11, 2018 from https://github.com/WebAssembly/reference-types

WebAssembly Community Group. 2018c. WebAssembly. (2018). Retrieved June 11, 2018 from http://webassembly.org

WebAssembly Community Group. 2018d. WebAssembly. (2018). Retrieved June 11, 2018 from https://github.com/WebAssembly/spec/tree/master/interpreter

David J. Wheeler and Roger M. Needham. 1994. TEA, a tiny encryption algorithm. In *Lecture Notes in Computer Science*. Springer.

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994).

Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Alon Zakai. 2015. Compiling to WebAssembly: It's Happening! (2015). Retrieved July 12, 2018 from https://hacks.mozilla.org/2015/12/compiling-to-webassembly-its-happening/

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM.