Constant-Time WebAssembly

John Renner, Sunjay Cauligi, and Deian Stefan
UC San Diego

Abstract

As ever more applications are designed to run inside browsers and other JavaScript runtime systems, there is an increasing need for cryptographic primitives that can be used client-side. Unfortunately, implementing cryptographic primitives securely in high-level languages is extremely difficult—runtime system components such as garbage collectors and just-in-time compilers can trivially introduce timing leaks in seemingly secure code. We argue that runtime system designs should be rethought with such applications—applications that demand strong guarantees for the executed code—in mind. As a concrete step towards this goal, we propose changes to the recent WebAssembly language and runtime system, supported by modern browsers. Our *Constant-Time WebAssembly* enables developers to implement crypto algorithms whose security guarantees will be preserved through compiler optimizations and execution in the browser.

1 Introduction

Modern application development has largely migrated to JavaScript and the Web platform. Even desktop applications (e.g., Spotify, Slack, Visual Studio Code) are now simply JavaScript and HTML bundled with a browser-rendering engine.[1] While this affords many benefits, secure applications that demand encryption (e.g., Signal, Cryptocat, Keybase) are notoriously difficult to implement.

Writing secure cryptographic code in low-level languages like C, where developers have total control over program execution, is challenging; writing secure code in a language like JavaScript is even more so. To ensure that code handling secrets runs in constant time or that secrets are securely erased, C developers often introduce semantic gaps to "trick" the compiler into producing secure code [7, 8]. JavaScript developers must *additionally* trick the runtime system—in particular, the just-in-time compiler (JIT) and garbage collector (GC) of JavaScript engines—to not introduce time variabilities or retain secrets that should otherwise be erased. Unsurprisingly, no existing JavaScript crypto library—from SJCL [16] to libsignal [13]—even attempts to tackle these challenges. In many cases, this may not even be possible at the JavaScript layer—e.g., we cannot prevent a copying GC from making additional copies of (secret) data in JavaScript alone. Implementing JavaScript APIs in C++ is not necessarily better either—e.g., WebCrypto [17] and Node.js's crypto libraries do not provide constant-time guarantees and, unfortunately, are still at the mercy of the JavaScript GC [6].

We argue that high-level runtime systems (e.g., Chrome or Electron) should make it possible for developers to express and securely execute computations on secret data. In particular, developers should be able to declare which data are sensitive and implement cryptographic algorithms without worrying about the underlying runtime system introducing covert timing channels or optimizing away security-critical code (e.g., secret-memory erasing code) [8].

Retrofitting JavaScript runtimes to ensure that the high-level semantics of secure computations are preserved when executed is, at best, a daunting task. Luckily, most browsers have added support for WebAssembly (wasm)—a strongly-typed, low-level machine language [4, 9]. By extending this still-evolving language with data security types [15], and by retrofitting the still-simple wasm compilers and execution engines to respect these types, we can provide a means for developers to write secure cryptographic computations in the context of high-level runtime systems.

To this end, we outline three extensions to wasm: First, we extend the wasm language and type system to add support for secret integer values and memory spaces. Second, we extend the wasm validator, which ensures that wasm code is well-formed, to additionally ensure that code does not leak secrets via data or control flow. Finally, we modify the wasm execution environment to ensure that these high-level security guarantees are preserved during execution, even in the presence of low-level compiler optimizations. We call this modified language and runtime *Constant-Time WebAssembly (ct-wasm).*

2 Constant-Time WebAssembly

The lack of language support for expressing data sensitivity has been a key contributor to many vulnerabilities in crypto implementations. By not distinguishing secret data from public data, languages have made it easy for developers to accidentally and repeatedly leak sensitive data [5, 10, 12, 14]. Even worse, they have made it easy for compilers to introduce low-level vulnerabilities (e.g., timing attacks), for code that is deemed secure at the high-level [7, 8, 18].

We propose to make data security explicit at the language-level precisely to avoid the pitfalls of existing approaches. In particular, we propose to extend the wasm language to allow developers to declare the sensitivity of values and memories using *labeled* types. For example, values of type `i32'secret` or `i64'secret` are considered secret in ct-wasm; values of

[1] https://electron.atom.io/

PriSC'18, January 13, 2018, Los Angeles, CA, USA

```
(memory $pub 10)        ; public linear memory (10 pages)
(memory $sec 10 secret) ; secret linear memory (10 pages)
...
(i32.load $pub 3)       ; load $pub[3], i32'public value
(i32.load $sec 5)       ; load $sec[5], i32'secret value
```

Figure 1. Memories can be declared secret (like $sec) or public (like $pub); reads from these memories receive their respective labeled types.

type `i32` or `i64`, on the other hand, are public.[2] Similarly, memory regions declared `secret`, as shown in Figure 1, are considered secret whereas memories lacking such a `secret` qualifier are considered public.

To prevent developers from writing wasm code that would leak secret data, we extend the wasm type checker to enforce several information flow control (IFC) typing rules [7, 15]. More concretely, we modify the wasm validation pass to: (1) track `secret`-labeled data at the instruction-level, assigning labels to values in a standard flow-insensitive fashion [15]; (2) disallow branches on secret values; and (3) prevent using secret values where public values are expected (e.g., public-memory write instructions or public function-call arguments). By conservatively and explicitly extending wasm with `secret`-typed memories[3] and instructions—essentially blessing language primitives that run in constant-time—we ensure that ct-wasm remains safe as wasm is extended with new, potentially dangerous, features (e.g., threads and garbage collection): by default, ct-wasm instructions will only compute on public data and thus cannot leak sensitive data.

Though these rules are very conservative, we do not envision crypto implementations to be written directly in ct-wasm; instead, we believe ct-wasm to be a good compilation target for high-level crypto-oriented programming languages, such as HACL* [19] and FaCT [7]. These languages already produce low-level code that follow an IFC discipline that is (at least) as restricting as ct-wasm's. FaCT, for example, disallows writing secret data to public buffers. Similarly, it ensures that there are no branches on secrets by transforming such branches to constant-time arithmetic operations [7], like most constant-time crypto libraries (e.g., libsodium [1], OpenSSL [3], and mbedTLS [2]).

By enforcing IFC at the low-level, high-level languages like HACL* and FaCT can offload the burden of ensuring that computations on secret data run in constant-time to ct-wasm—with ct-wasm, the HACL* and FaCT compilers simply need to correctly track labels. Developers can even import third-party cryptographic modules or modules compiled by highly optimized but "constant-time oblivious" compilers and have the guarantee that they cannot leak secrets via timing channels. In wasm, and thus ct-wasm, developers specify the types of their imports. By further specifying the secrecy of arguments or return values of imported functions, for example, developers can ensure—via the ct-wasm label-type checker—that such code cannot leak the secret data.

[2]ct-wasm does not introduce secret float types due to the complexity of operating on floating point values in constant-time [11].

[3]The availability of multiple memories is referenced in the wasm specification [4], though not yet implemented.

Optimization	Safe?
Constant folding	Yes
Dead store elimination	May remove writes
Common subexpression elimination	No (remove writes)
Global value numbering	No (remove writes)
Strength reduction	No (emit unsafe instructions)

Figure 2. Safety of different wasm optimization passes.

The demand for IFC labels at a low-level has the additional role of carrying security-relevant information to wasm compilers and execution environments. Without this information, an optimization pass may introduce vulnerabilities. Indeed, most of the optimizations used in Chromium's V8 wasm implementation, shown in Figure 2, are unsafe. For example, common subexpression elimination (CSE) and global value numbering (GVN) may remove writes to secret memories that deliberately exist to scrub sensitive information. Similarly, a strength reduction pass may introduce branches on secret values or emit variable-time instructions that compute on secret operands.

The ct-wasm execution environment leverages labels to prevent such pitfalls when optimizing code. For example, the V8 engine can disable all optimizations or introduce whole new optimization pipelines for computations that handle `secret` data. Though secure, this approach gives up on performance. Hence, we instead propose to slightly modify existing optimization passes to preserve the security guarantees expected at the language-level. Specifically, we propose to modify optimizations such as CSE, GVN, and strength reduction to inspect labels and: (1) leave instructions that write to secret memories intact, (2) avoid emitting branch instructions on secret values, and (3) avoid emitting variable-time instructions when secret values are used as operands. While these three modifications are necessary for any constant-time system, the low-level nature of ct-wasm makes them sufficient as well. To our knowledge, ct-wasm is the first proposal to address both security and performance for constant-time code in optimizing compiler pipelines.

Acknowledgments

We thank Michael Smith and Dan Gohman for fruitful discussions on the design and implementation of ct-wasm. We thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] libsodium. https://github.com/jedisct1/libsodium.

[2] mbed TLS. https://github.com/armmbed/mbedtls.

[3] OpenSSL. https://github.com/openssl/openssl.

[4] Webassembly specification. https://webassembly.github.io/spec/core/.

[5] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P*, 2013.

[6] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in JavaScript bindings. In *IEEE S&P*, 2017.

[7] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan. FaCT: A flexible, constant-time programming language. In *IEEE SecDev*, 2017.

[8] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *IEEE SPW*, 2015.

[9] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200. ACM, 2017.

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

[11] D. Kohlbrenner and H. Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*, 2017.

[12] B. Moeller et al. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. https://www.openssl.org/~bodo/tls-cbc.txt, 2004.

[13] Open Whisper Systems. Signal protocol C library. https://github.com/WhisperSystems/libsignal-protocol-c.

[14] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*. Springer, 2006.

[15] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1), 2003.

[16] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *IEE ACSAC*, 2009.

[17] M. Watson. Web cryptography API. W3C, https://www.w3.org/TR/WebCryptoAPI/, 2017.

[18] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko. Dead store elimination (still) considered harmful. In *USENIX Security*, 2017.

[19] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *ACM CCS*, 2017.