Robust Constant-Time Cryptography

Matthew Kolosick[*] Basavesh Ammanaghatta Shivakumar[†] Sunjay Cauligi[†]
Marco Patrignani[‡] Marco Vassena[§] Ranjit Jhala[*] Deian Stefan[*]
[*]UC San Diego [†]MPI-SP [‡]University of Trento [§]Utrecht University

1 Extended abstract

The constant-time property is considered the security standard for cryptographic code. Code following the constant-time discipline is free from secret-dependent branches and memory accesses, and thus avoids leaking secrets through cache and timing side-channels [2, 5]. Though security against side-channel attacks is an important concern for secure cryptographic implementations [4], the constant-time property makes a number of implicit assumptions that are fundamentally at odds with the reality of cryptographic code.

Constant-time is not robust. The first issue with constant-time is that it is a *whole-program* property: It relies on the *entirety* of the code base being constant-time. But, cryptographic developers do not generally write whole programs; rather, they provide libraries and specific algorithms for other application developers to use. As such, developers of security libraries must maintain their security guarantees even when their code is operating within (potentially untrusted) application contexts.

Constant-time requires memory safety. The whole-program nature of constant-time also leads to a second issue: constant-time *requires* memory safety of *all* the running code. *Any* memory safety bugs, whether in the library or the application, will wend their way back to side-channel leaks of secrets if not direct disclosure. And although cryptographic libraries should (and are) written to be memory-safe, it's unfortunately unrealistic to expect the same from every application that uses each library.

We provide an example from the libsodium cryptographic library [7]: The code below shows the (abridged) C implementation of the Salsa20 stream cipher, a constant-time encryption primitive.

```
1  static int stream_ref(u8 *c, u64 clen, u8 *n, u8 *k)
2  {
3      ... u8 kcopy[32]; ...
4      for (i = 0; i < 32; i++) {
5          kcopy[i] = k[i];
6      }
7      ...
8      while (clen >= 64) {
9        crypto_core_salsa20(c, in, kcopy, NULL); ...
10     }
11     ...
12     sodium_memzero(kcopy, sizeof kcopy);
13     return 0;
14 }
```

The local buffer `kcopy` holds a copy of the key, which is left unchanged by `crypto_core_salsa20`. The data held in `kcopy` is not returned from the function, and (since the algorithm is constant-time) it is not leaked through any timing side-channels, so it should not matter if its contents are wiped or not. However, if an attacker is able to, e.g., exploit a memory safety vulnerability in the linked application code, they might be able to read arbitrary (or targeted) bits from memory, allowing them to steal the key data from the leftover `kcopy`. Clearly, the classical definition of constant-time security is insufficient to capture this notion.

Different attackers require different defenses. While libraries like libsodium add memory clearing defenses like those shown above, others choose to elide them. We argue that neither of these choices is universally correct: For example if libsodium is run in a safe Rust application, clearing the intermediate memory is unnecessarily defensive; whereas if a linked application contains memory disclosure bugs, then a library without such mitigations will be leaving sensitive data vulnerable to an attacker. Ideally, software security properties for cryptographic code should allow us to reason about which protections are needed based on the what kinds of applications it will be linked with.

Spectre complicates everything. Finally, albeit quite unsurprisingly, *speculative execution* complicates even further any discussion about cryptographic software security. Just as with classical constant-time, we already have a variety of tools and formal techniques to ensure that cryptographic code *itself* is protected from Spectre attacks [5]. However, these techniques usually come with substantial performance tradeoffs, making them impractical to use for the entirety of an application. Due to Spectre attacks, applications that appear memory-safe can still be tricked into revealing arbitrary memory data; once again cryptographic libraries must not only harden their own code, but must also defend against these *speculative* vulnerabilities in the application. Although there has been much work developing constant-time properties for speculative execution [5], speculative constant-time is still a whole-program property; it, too, is wholly insufficient if we want to make guarantees for cryptographic libraries.

1.1 Robust constant-time

Our answer to this problem is *robust constant-time*. Like other robustness properties [1, 3, 6, 8–10], robust constant-time ensures that a given library does not leak secrets *regardless of*

the linked application. In addition, we capture the varying assumptions about applications—memory safety, the presence of read gadgets, speculation, etc.—as *classes* of application contexts. This allows us to formally examine different mitigation strategies when linking against, for instance, safe Rust applications, buggy C applications, or even applications full of Spectre gadgets. Our notion of robust constant-time underpins our formal security model for developing cryptographic libraries: Not only must a library be secure, it must remain secure even in a given attacker context.

To formally define robust constant-time, we first define *libraries*, which we parameterize with a set of *API functions* Γ and a set of *secrets* Δ:

Definition 1 (Γ-Δ-libraries)**.** Given an API context Γ and secret context Δ, we say L is a Γ-Δ-*library* with private context Γ_p when L is a closing substitution for $\Gamma_p \uplus \Gamma$ with codomain `lib`-labeled functions well-formed under Δ.

1.2 Characterizing attackers

Since application/attacker assumptions differ, we also parameterize our definition of robust constant-time with a *class of contexts* to capture the variation in attacker models. For instance, we can assume an application written in safe Rust is memory safe, and thus we don't need to worry about memory disclosure bugs from the application. We can thus apply different protections to our library than if we were linking against applications written in C or where Spectre gadgets are a concern.

We formally define an *attacker* in terms of a *trace safety property* $\Gamma \vdash e$ which captures the set of operations the attacker/application is allowed make when interacting with the library. The four concrete attack classes we define are: 1) Memory-safe attackers, where the application can neither read nor write out-of-bounds; 2) read-only attackers, where the application might read out-of-bounds; 3) memory-unsafe attackers, where the application contains arbitrary memory safety vulnerabilities; and 4) speculative attackers, where the application contains Spectre-style vulnerabilities. We define speculative attackers via a novel, high-level speculative semantics, parameterized by a *speculator* that controls when and how speculation and rollback occur. This definition of speculation allows us to capture a wide variety of possible speculative attacks.

As an example, our formal definition of a read-only attacker is as follows:

Definition 2 (Read-only attacker)**.** $\Gamma \vdash e$ is a *read-only attacker* if, for all Γ-Δ libraries L, initial states $S \vDash \Delta$, and $\overline{\alpha} \in \text{traces}(\langle S \mid L(e)\rangle)$, we have $(\emptyset, \emptyset) \vdash \text{read-only } \overline{\alpha}$.

The notation $(\emptyset, \emptyset) \vdash \text{read-only } \overline{\alpha}$ captures that the only "bad" actions e performs are reading out-of-bounds; the definition of read-only partitions the trace into alternating sequences of application and library events. We impose restrictions on the application events under the assumption that the library events are well-behaved, in essence treating the library as a "context" for executing the application.

We can now define robust constant-time in the context of read-only attackers:

Definition 3 (Read-only robust constant-time)**.** We say a Γ-Δ-library L is *read-only robustly constant-time* if, for all read-only attackers $\Gamma \vdash e$ and secret states S and S' such that $S \vDash \Delta$ and $S' \vDash \Delta$, we have $\text{traces}(\langle S \mid L(e)\rangle) = \text{traces}(\langle S' \mid L(e)\rangle)$.

We can define the other attacker models similarly. For speculative execution, we further parameterize by a class of speculators and we relate the *speculative* traces (those produced by the speculative semantics).

1.3 A robust constant-time compiler

Robust constant-time makes implicit security guarantees concrete. For example, with robust constant-time we can show which functions *must* clear temporary data—like in the libsodium example—and which functions can get away without additional mitigations. Unfortunately, current cryptographic libraries implement these mitigations manually. As such, they fundamentally limit themselves to a single attacker model, and must make compromises in their level of security—always picking the strongest possible protections would lead to significant (and unnecessary) overhead. For example, the libsodium developers explicitly chose not to add speculative protections,[1] instead opting for best-effort protections in such cases to maintain performance.

We instead build a compiler that is aware of robust constant-time and is *parameterized by the attacker model*, allowing us to compile a library with different mitigations for different attacker models. Thus the same library code can be used safely and efficiently whether called from Rust or from C, or even when Spectre is a concern. During compilation, our compiler performs a static taint analysis of the library code to determine which data is secret or can reveal secret information. Then, depending on the attack model, the compiler moves secret data to protected memory regions, clears accessible temporary secrets, and sanitizes context switches between the library and application code.

Formally, we assume the cryptographic library is already classically constant-time; this can be achieved through existing tools such as Blade [11]. For each attacker class A, we then have a compiler $\mathbb{C}_\mathsf{A}$ such that compiling a constant-time library L with $\mathbb{C}_\mathsf{A}$ guarantees that it is robustly constant-time against A:

Theorem 1 (Compiler is secure)**.** *Let L be a Γ-Δ-library such that L is classically constant-time. Then $\mathbb{C}_\mathsf{A}(L)$ is robustly constant-time w.r.t. attacker class* A.

[1] libsodium forgoes an appropriate memory fence in their implementation of `sodium_memzero`: https://github.com/jedisct1/libsodium/issues/802.

References

[1] Michael Backes, Catalin Hritcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security* 22, 2 (2014), 301–353. https://doi.org/10.3233/JCS-130493

[2] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. 2019. System-Level Non-interference of Constant-Time Cryptography. Part I: Model. 63, 1 (2019), 1–51. https://doi.org/10.1007/s10817-017-9441-5

[3] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (Feb. 2011), 45 pages. https://doi.org/10.1145/1890028.1890031

[4] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. 48, 5 (2005), 701–716. https://doi.org/10.1016/j.comnet.2005.01.010

[5] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022-05). 666–680. https://doi.org/10.1109/SP46214.2022.9833707

[6] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11, 4 (July 2003), 451–519. http://dl.acm.org/citation.cfm?id=959088.959090

[7] LibSodium. 2022. https://doc.libsodium.org/.

[8] Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 1 (Feb 2021), 41 pages. https://doi.org/10.1145/3436809

[9] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* 4, POPL (2020), 32:1–32:32. https://doi.org/10.1145/3371100

[10] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

[11] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434330