Fast verified post-quantum software, part 1: RAM subroutines

Daniel J. Bernstein

23 April 2021

"Warning to potential users: NISTPQC, despite being an important and timely project, has produced the largest regression ever *in the quality of cryptographic software. This will not be easy to fix."*

—October 2018, `https://tinyurl.com/fwu5jyce`

"It's actually a bug within SymCrypt, the core cryptographic library responsible for implementing asymmetric crypto algorithms in Windows 10 and symmetric crypto algorithms in Windows 8."

—"Warning: Google Researcher Drops Windows 10 Zero-Day Security Bomb", June 2019, Forbes, `https://tinyurl.com/y69fx3nh`

"Produced signatures were valid but leaked information on the private key. . . . The fact that these bugs existed in the first place shows that the traditional development methodology (i.e. 'being super careful') has failed."

—"OFFICIAL COMMENT" within NISTPQC, September 2019, `https://tinyurl.com/y5w46bde`

"Libgcrypt, wolfSSL, and Crypto++ have issued patches over the summer to fix this bug. Maintainers of MatrixSSL fixed some issues, but the library remains vulnerable. Oracle's SunEC library remains open to attacks."

—"Minerva attack can recover private keys from smart cards, cryptographic libraries", October 2019, ZDNet, `https://tinyurl.com/y6rlkov4`

"Experiments show that the attack code is able to extract the secret key for all security levels using about 2^{30} decapsulation calls."

—"A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM", June 2020, `https://ia.cr/2020/743`

"It looks like the FrodoKEM team also fixed the timing oracle [GJN20] badly and caused a more serious security problem while trying to do that."

—"OFFICIAL COMMENT" within NISTPQC, December 2020, `https://tinyurl.com/yyq4mcy2`

Cryptographic *software* is a security disaster, even when there are no public breaks of the cryptographic *primitives* that the software is supposed to be providing. This problem isn't specific to post-quantum cryptography, as the SymCrypt and Minerva vulnerabilities show; but post-quantum cryptography makes the problem worse.

On the defense side, there is ongoing work on cryptographic software verification, aiming to ensure that the software is as hard to break as the primitives. The success rate of this work drops as the software complexity increases. Drivers of complexity in cryptographic primitives, and of even more complexity in software, include (1) performance pressure and (2) pressure to avoid quantum attacks. These drivers interact: post-quantum cryptography has a reputation of being less efficient than pre-quantum cryptography, and performance pressure then prompts even more complications aimed at improving efficiency.

Let's look at one important part of the attack picture, namely timing attacks, and more specifically at timing attacks against a core operation used in algorithms, namely array lookups. Array indices in cryptographic algorithms are often secret—but RAM on most computers takes time that depends on which location in memory is being accessed. Presumably this leaks secret information to the attacker.

Full timing attacks exploiting secret *branch conditions* in cryptographic algorithms were demonstrated in the mid-1990s and were immediately appreciated as a threat. It was not immediately appreciated that secret *array indices* are exploitable. Several years passed before the first demonstrations of attacks exploiting cache misses. This does not mean that the attacks were slow; consider, e.g., `https://eprint.iacr.org/2005/271` stealing a disk-encryption key in 65 milliseconds.

In the meantime the Rijndael designers had claimed that table lookups are “not susceptible to a timing attack”. NIST adopted this incorrect position, claiming in its selection rationale that table lookups are “not vulnerable to timing attacks”. Secret array indices in AES software continue to create security failures two decades later, as illustrated by `https://eprint.iacr.org/2019/996`.

The first announcement of secret array indices in allegedly constant-time NISTPQC software was `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/mtGmXNlMi7o/m/Qs3cw8DIAgAJ` in December 2017. In a fantasy world where the public cryptographic community has endless resources for cryptanalysis, there would be papers analyzing exactly how much is leaked by each secret array index in each software package published for each NISTPQC submission, and one could hope that three or four years would be enough time for the analysis to stabilize. In the real world, cryptanalysts are overloaded. An easy Round2 break wasn't published until 2020: `https://eprint.iacr.org/2020/241`. The occasional cryptanalysts writing timing-attack papers can easily find secret branch conditions to exploit—consider, e.g., the aforementioned June 2020 attack exploiting the use of `memcmp` in the official Frodo software—and don't need to bother studying array indices. So we don't know how easy it is to exploit the secret array indices in NISTPQC software.

What we do know is that secret array indices add complications and errors into security analyses. For example, in response to demonstrated attacks exploiting cache misses created by secret array indices, OpenSSL deployed “scatter-gather” code from Intel where the secret part of an array index was only a position within a cache line. Intel and OpenSSL both claimed that this code was constant-time—but this claim was incorrect, and years later `https://eprint.iacr.org/2016/224` presented a complete attack. It is interesting to note that “scatter-gather” code saves only about 6% compared to true constant-time code, illustrating the performance pressure mentioned above.

How should we act when we see that security analysis is complicated and don't know what the results of a thorough security analysis will be? Common practice is to wait and see, reacting only after an attack has been demonstrated. A safer approach is to proactively eliminate the source of the complications—in this case, to follow a rule of never using secret array indices (and, of course, never using secret branch conditions). But how do we check whether software follows this rule? When it doesn't, how do we fix it? And is the resulting software fast enough for people to happily use it?

There are now many tools, surveyed in `https://neuromancer.sk/article/26`, that try to check whether software follows this constant-time rule. The SUPERCOP benchmarking framework now includes an improved version of Moritz Neikes's TIMECOP and checks 712 constant-time implementations of various primitives. This includes asm implementations, implementations using the OS binary shipment of OpenSSL, etc. Rejection sampling is handled with minor code changes for explicit declassification. These tools are easily usable as part of the software-development process. Other tools are harder to use but provide stronger guarantees; ongoing work aims to combine usability and completeness.

How can algorithms with secret array indices be converted into software without secret array indices? A safe simulation of a RAM lookup is easy—read each array element and use arithmetic to simulate selection of the desired array element—but has cost increasing linearly with the number of possible indices. Generic "ORAM" techniques have better cost asymptotics but much worse constants. These costs are only a minor issue for, e.g., a size-2 array inside a Montgomery ladder, or a slightly larger array inside fixed-window scalar multiplication, but post-quantum algorithms often use arrays with hundreds or thousands of elements.

There are more efficient solutions that use sorting to simulate many *parallel* RAM lookups. Having many RAM lookups to handle at once is a common situation: most of the algorithms of interest for high-speed cryptography are highly parallelizable. The sorting software from `https://sorting.cr.yp.to` takes constant time, has automated tools verifying correctness, and is the fastest software available to sort integer arrays in cache on Intel CPUs; there is a synergy between vectorization and constant-time programming. On smaller CPUs, variable-time sorting software is sometimes faster, but this constant-time software is fast enough to avoid posing problems for the performance of post-quantum cryptography.

Sometimes a single permutation is applied to many different arrays. One can view each application of the permutation as a sorting step, but one can gain speed by precomputing control bits for a Beneš network for the permutation. `https://cr.yp.to/papers.html#controlbits` presents proofs verified by HOL Light for the correctness of a fast parallel algorithm for this precomputation.

This software is already used inside the latest software releases for Classic McEliece (sorting and control bits), NTRU (sorting), and NTRU Prime (sorting). Ongoing work aims for more comprehensive verification, extended platform support, and DSLs to save time in software development.