Improving Software Quality in Cryptography Standardization Projects

Matthias J. Kannwischer
Academia Sinica
Taipei, Taiwan
matthias@kannwischer.eu

Peter Schwabe
MPI-SP, Bochum, Germany
Radboud University, The Netherlands
peter@cryptojedi.org

Douglas Stebila
University of Waterloo
Waterloo, Canada
dstebila@uwaterloo.ca

Thom Wiggers
Radboud University
Nijmegen, The Netherlands
thom@thomwiggers.nl

***Abstract*—The NIST post-quantum cryptography (PQC) standardization project is probably the largest and most ambitious cryptography standardization effort to date, and as such it makes an excellent case study of cryptography standardization projects. It is expected that with the end of round 3 in early 2022, NIST will announce the first set of primitives to advance to standardization, so it seems like a good time to look back and see what lessons can be learned from this effort. In this paper, we take a look at one specific aspect of the NIST PQC project: software implementations.**

We observe that many implementations included as a mandatory part of the submission packages were of poor quality and ignored decades-old standard techniques from software engineering to guarantee a certain baseline quality level. As a consequence, it was not possible to readily use those implementations in experiments for post-quantum protocol migration and software optimization efforts without first spending a significant amount of time to clean up the submitted reference implementations.

We do not mean to criticize cryptographers who submitted proposals, including software implementations, to NIST PQC: after all, it cannot reasonably be expected from every cryptographer to also have expertise in software engineering. Instead, we suggest how standardization bodies like NIST can improve the software-submission process in future efforts to avoid such issues with submitted software. More specifically, we present PQClean, an extensive (continuous-integration) testing framework for PQC software, which now also contains "clean" implementations of the NIST round 3 candidate schemes. We argue that the availability of such a framework—either in an online continuous-integration setup, or just as an offline testing system—long before the submission deadline would have resulted in much better implementations included in NIST PQC submissions and overall would have saved the community and probably also NIST a lot of time and effort.

***Index Terms*—NIST PQC, post-quantum implementations, testing cryptographic software, open source, continuous integration**

This work has been supported by the European Research Council through Starting Grant No. 805031 (EPOQUE), by Canada's Natural Sciences and Engineering Research Council (NSERC) through Discovery Grant RGPIN-2016-05146, by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments EXC 2092 CASA - 390781972, and by Academia Sinica through the Sinica Investigator Award AS-IA-109-M01.

1. Introduction

The selection of cryptographic algorithms for use in applications and standards is increasingly accomplished via public competitions, in which researchers are invited to submit algorithms that are then subject to public review. One significant case study of such a process is the post-quantum cryptography (PQC) standardization project of the United States National Institute of Standards and Technology (NIST). In 2016, NIST announced its intention to standardize quantum-resistant digital signatures and public-key encryption / key-encapsulation mechanisms (KEMs) in a multi-year public process that continues as of March 2022. This PQC standardization project was to be modeled after NIST's earlier public competitions that lead to the Advanced Encryption Standard (1997–2001, 15 submissions total, 2 rounds) and the Secure Hash Algorithm (SHA-3) (2007–2015, 51 submission total, 2 rounds).[1]

The PQC standardization project is NIST's largest cryptography standardization effort to date. There were 82 initial submissions of which 69 have been accepted as "complete and proper" submissions. By now, those have been winnowed down over three rounds of public evaluation to 7 finalists and 8 alternate candidates. The selection of algorithms for standardization is expected in March 2022 leading to a final standard in 2024. A fourth round for algorithms meriting further investigation and an "on-ramp" for new signature scheme designs is also expected.

Compared with the AES and SHA-3 competitions, the PQC standardization project is more complex in several ways, beyond the sheer number of submissions and rounds. The PQC standardization project involves two distinct cryptographic primitives (digital signatures and KEMs) compared to one in each of the previous competitions (block ciphers for AES, hash functions for SHA-3). There is also a much greater variety of mathematical constructions used to build the candidates. As a consequence, there are much more pronounced differences between the speed and output size characteristics of the various candidates. Whereas the AES competition prescribed just three sizes—128-, 192-, or 256-bit keys, all with 128-bit block sizes—post-quantum KEM candidates (even just among round 3 finalists and alternates) have public keys ranging in size from 197 bytes to more than 1.3 MB and encapsulations

1. The PQC standardization process was not explicitly called a "competition" as it might result in several winners.

from 128 bytes to 21 KB; and round 3 signature candidates have public keys from 32 bytes to 1.9 MB and signatures from 66 bytes to 209 KB.

Certainly, this scale and variety made NIST's selection task harder, and also meant a greater burden on the community in reviewing and evaluating the candidates, both in terms of security and performance. The wide range of size and speed characteristics meant that standalone microbenchmarks would not suffice to evaluate the suitability of candidates for adoption, and instead would require integration and testing of candidates in a variety of contexts. This is where the importance of software implementations plays a greater role.

As with previous competitions, NIST required submissions to be accompanied by software implementations. Each submission needed to include a reference implementation and an optimized implementation for Intel x64, both in ANSI C (no assembly or intrinsics allowed, limited use of external libraries), along with known-answer test (KAT) values to check correctness. Submissions could also include additional implementations for other platforms or microarchitectures. NIST provided a C application programming interface (API) for each cryptographic primitive as well as a test harness for known-answer tests. (See Section 2 for a detailed review of NIST's original submission requirements and how they evolved over later rounds.)

In addition to NIST's internal benchmarking, there have been many community and industry projects building on software implementations submitted to NIST. These include: the SUPERCOP benchmarking project [1]; the PQClean project [2] for standalone C implementations on Intel and ARMv8; the pqm4 project [3] for ARM Cortex M4 platform; and the Open Quantum Safe project [4] with a library of C implementations as well as integrations of those algorithms into popular libraries, applications, and protocols. There have also been many research papers and industry experiments building on the above-mentioned projects or directly on software submissions to NIST.

Due to the lack of consistency, organization, and quality of submitted software, each of the above initiatives has involved a repetition of time and effort in getting submitted software to compile and run cleanly on various target platforms.

Admittedly, not all cryptographers should be expected to have the software engineering expertise to create production-quality software, and indeed expecting so may disincentivize the submission of mathematically innovative proposals. Nonetheless, a public cryptography standardization initiative does need software of sufficient quality that works in a variety of settings and has performance characteristics representative of production implementations, in order for good decisions to be made.

We argue that the NIST PQC standardization effort—and future public cryptography standardization initiatives—could be improved by having a more extensive software framework prepared in advance by the organizers for submitters, relying on modern continuous integration and testing tools. Our goal in this paper is to lay out the requirements for such a framework, based on our experience in the PQClean project where we assembled a collection of standalone C implementations of NIST PQC submissions, and developed a continuous integration testing framework to improve the software we assembled.

1.1. Organization of this paper

In Section 2, we review the submission requirements issued by NIST over the lifetime of the PQC standardization project to date, specifically as related to software implementations; understanding the software submission requirements provides context to the types and quality of software submitted.

In Section 3, we begin to examine what went wrong in the process with respect to software implementations. Our main observations in this section are (a) that the reference implementations were not ready to meet all the needs expected of them; (b) that "ANSI C" as the language for both reference and optimized implementations may not be the best choice; in particular as (c) use of standard software-engineering techniques for programming in C were not enforced.

In Section 4, we propose that future cryptography competitions could be improved by having the organizers provide an extensive testing framework for software implementations, and we enumerate desired features of such a framework.

We continue in Section 5 to present details of our PQClean framework, which is an open-source collection of C implementations of NIST PQC candidates, along with an extensive array of compile- and run-time tests via a range of continuous integration tools. Through the process of adding PQC algorithms to PQClean and running our test framework, we identified flaws in the implementations of almost every of the 17 schemes from the NIST PQC project that have been added to PQClean to date; these are summarized in Table 1.

Much of sections 3–5 focus on C implementations. In Section 6, we look beyond PQClean's central focus on "cleaning" C implementations, and discuss alternatives to C for representing specifications as well as extensions beyond testing frameworks for cryptographic standardization processes.

We wrap up in Section 7 with conclusions and recommendations.

1.2. Related work

The risk of flaws in cryptographic software has been well-known for decades [5], [6], [7]. There are many potential causes for such flaws, which are important to distinguish to help put this paper's focus into context. Obviously, there can be cryptographic weaknesses in the cryptographic algorithm itself (weak parameters or broken cryptographic assumptions), applicable to any particular implementation, which is therefore outside the scope of this paper's focus. The cryptographic implementation could be used in a context that does not match the implementation's threat model: side-channel attacks against implementations without countermeasures, for example. It is also possible that applications and protocols might improperly use otherwise good cryptography algorithms and implementations. The latter is quite common; for example, Lazar et al. [8] evaluated 269 flaws for cryptographic software reported in the Common Vulnerabilities and Exposures (CVE) database from 2011–2014 and found that only 17% were in cryptographic libraries, whereas 83% were "misuses of cryptographic libraries by individual applications."

Our focus is on when the implementation of a cryptographic algorithm (for which there are no known cryptanalytic attacks) contains flaws, and the software development steps that lead to those flaws. Blessing et al. [9] examined vulnerabilities specifically in open-source C/C++ cryptographic libraries, and found that only 27% of vulnerabilities were cryptographic issues, whereas 37% of vulnerabilities were memory safety or resource management issues, 11% involved improper input validation, and 5% were numeric issues. There are also high-profile examples that seem to derive from particular C coding styles, such as the lack of braces leading to the so-called `goto fail` bug [10].

In the context of cryptographic standardization projects, in particular, Mouha et al. [11] studied software implementations submitted to the NIST SHA-3 competition. Using solely black-box testing, they found a total of 68 bugs in 41 of the 86 reference implementations submitted, none of which were discovered by the test suite provided by NIST.

A 2015 survey by Braga and Dahab [12] surveys techniques for the development of secure cryptographic software. They identify a sequence of three levels of cryptographic software development: 1) cryptographic library programming and verification; 2) cryptographic software programming and verification; and 3) cryptographic software testing. For each level, they identify a range of techniques that can be used to reduce the risk of flaws, including using secure languages, secure code generation, applying static and dynamic analysis tools, and using functional tests and adversarial tests (fault injection, fuzzing).

One trend is to create implementations of cryptographic primitives domain-specific or specialized languages and then generate lower-level implementations from there, with compilers and code generators yielding certain assurances. Examples include the HACL* library written in F* that generates C code [13]; and Jasmin [14] which generates EasyCrypt code that can be verified for security and functional correctness, and x86_64 assembly code for execution; Jasmin can even include mitigations against microarchitecture attacks such as SPECTRE [15].

For implementations that are originally written in C, there are some tools and techniques available for aiding in secure software development, including a range of general-purpose static and dynamic analysis tools. One specialized technique in the context of cryptography is the use of Valgrind to detect control flow based on secret data, an example of which is the TIMECOP project [16].

2. NIST PQC software submission requirements

In this section we review software requirements laid out by NIST for the PQC standardization project as it evolved. Figure 1 shows a timeline of the main events in the process.

2.1. Call for proposals and round 1 submissions

NIST issued a call for proposals in December 2016 which included a set of submission requirements and evaluation criteria [17]. In addition to the design documents, submissions were required to include several components related to software and testing:

- A reference implementation, written in ANSI C, intended to "promote understanding of how the submitted algorithm may be implemented", in which "clarity... is more important than... efficiency" [17, §2.C.1].
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor.
- A statement by the implementations' owners granting certain rights to use the implementation "for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard" [17, §2.D.3].
- Known-answer test (KAT) values to check the correctness of reference and optimized implementations [17, §5.B].

Submitters could at their discretion include additional implementations for other platforms, for example using intrinsics or assembly [17, §2.C.1].

The evaluation criteria in [17] referred to software and testing in several aspects:

- Performance: schemes will be evaluated based on their computational cost in software and hardware [17, §4.B.2].
- Side channel aspects: schemes that can be made side-channel resistant efficiently are more desirable than those that cannot, and "optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not" [17, §4.A.6].
- Flexibility: schemes that "can be implemented securely and efficiently on a wide variety of platforms" or for which implementations "can be parallelized to achieve higher performance" are desirable [17, §4.C.1].

NIST stated that the goal of the evaluation process during round 1 was to "narrow the candidate pool for more careful study and analysis" and that "this narrowing will be done primarily on security, efficiency, and intellectual property considerations" [17, §5.A]. NIST indicated that submitters would be able to provide updated optimized implementations for evaluation in round 2.

The call for proposals indicated that correctness and efficiency testing would be performed by NIST on the "NIST PQC Reference Platform, an Intel x64 running Windows or Linux and supporting the GCC compiler" [17, §5.B]. NIST further stated: "At a minimum, NIST intends to perform an efficiency analysis on the reference platform; however, NIST invites the public to conduct similar tests and compare results on additional platforms (e.g., 8-bit processors, digital signal processors, dedicated CMOS, etc.). NIST may also perform efficiency testing using additional platforms."

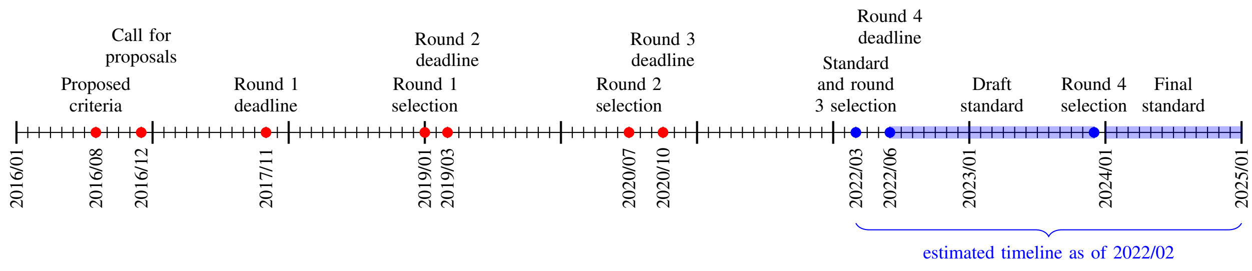

Figure 1. Timeline of NIST post-quantum cryptography standardization project. Dates after Feb. 2022 are estimates based on recent NIST announcements.

In addition to the text of the submission requirements, the call for proposals included several technical notes on API and testing, and some corresponding source code.

NIST's API notes [18] described the C API for signature schemes, public-key encryption schemes, and key-encapsulation mechanisms. The API was derived from the eBATS (ECRYPT Benchmarking of Asymmetric Systems) API in the eBACS project [1]. One notable characteristic of the API was that the signature API generated "attached signatures" (where the output of the signing function is a single variable-length "signed message" containing both the message and the signature together) rather than "detached signatures" (where the output of the signing function is a typically fixed-length signature digest without message). The API also included a function providing random bytes, a pseudorandom expander, and an optional deterministic random bit generator to facilitate known-answer tests.

NIST provided a short document [19] describing the process for generating known-answer test values, as well as a source-code archive [20] containing C files for generating KATs for signature schemes, public-key encryption schemes, and KEMs, and a C header file and implementation of a seeded pseudorandom number generator for generating consistent KAT values. The document also included an example Makefile for building and running the KAT programs.

2.2. Round 1 selection and round 2 submissions

In January 2019, NIST issued a status report [21] on round 1, including a discussion of its selection of round 2 candidates. The report noted that evaluation criteria used for selecting round 2 candidates were, in order of importance, "security, cost and performance, and algorithm and implementation characteristics" [21, §2.3]. Among the comments on individual schemes selected for round 2 were comments on speed (either positively or negatively) as well as side-channel attacks and constant-time implementations.

The round 1 status report included a statement that, as a next step, NIST was interested in more performance data, including "optimized implementations written in assembly code or using instruction set extensions, and analyses of implementation suitability of candidate algorithms in constrained platforms" [21, §4].

2.3. Round 2 selection and round 3 submissions

In July 2020, NIST issued a status report [22] on round 2, including a discussion of its selection of round 3 candidates. At around the same time, NIST issued an additional note with guidelines for submitting tweaks for round 3 [23].

The status report on round 2 noted that the evaluation period saw better data especially for constant-time implementations on Intel x64 as well implementations for ARM Cortex-M4 and hardware implementations, and observed that this included information about resources required by implementations, such as RAM or gate counts. The report indicated NIST's desire to see "more and better data for performance in the third round" including for "implementations that protect against side-channel attacks, such as timing attacks, power monitoring attacks, fault attacks, etc." [22, §2.2]. NIST concluded the report with a clear request for performance evaluation of implementations during round 3: "NIST hopes that with only seven finalists and eight alternate candidates, the public review period will include more work on side-channel resistant implementations, performance data in internet protocols, and performance data for hardware implementations in addition to more rigorous cryptanalytical study" [22, §4].

The guidelines for submitting tweaks for round 3 [23] relaxed the requirements on the optimized implementation included in the submission: "the reference implementation should still be in ANSI C; however, the optimized implementation is not required to be in ANSI C" and recommended "providing an AVX2 (Haswell) optimized implementation and [. . .] other optimized software implementations (e.g., microcontrollers) and hardware implementations (e.g., FPGAs)."

3. Problems with NIST PQC reference implementations

In this paper we will make the point that the reference implementations submitted to the NIST post-quantum competition did not achieve the declared goal of *promoting the understanding of how the submitted algorithm may be implemented* as well as they could have. In this section we give an overview of what we believe to be the reasons for this; in short, those are a combination of

1) different expectations of what a reference implementation should accomplish;
2) the choice of ANSI C as the primary programming language; and
3) insufficient use of standard software-development techniques, paired with a lack of experience with writing cryptographic software in the submission teams.

3.1. Reference implementation expectations

Let us first look at what one might reasonably expect from a reference implementation or what such an implementation might be used for. NIST's call made it clear that the central goal is about *clarity* of the code. Once the programming language—here, ANSI C—is fixed, it is not hard to start heated debates about what exactly "clarity of code" means; however, this is not the central problem. What is much more important is that reference code is typically used for much more than just "promoting understanding", including:

Generation of test vectors. This is probably the most obvious use-case for a reference implementation aside from portraying what the proposed algorithms look like in code. Being able to reliably generate test vectors is the foundation for any kind of regression testing of more optimized implementations.

Basic performance evaluation. A more controversial question is if a reference implementation should also be used as a baseline for performance evaluation. The call made it pretty clear that performance should *not* be a focus for the reference implementation. However, many "reference implementations" submitted to NIST did include pieces of code that were clearly optimized for speed rather than for readability. Also, many papers did report benchmarks of reference implementations [24], [25], [26], [27], [28], sometimes without clear warnings that such benchmarks say absolutely nothing about the performance of the proposed scheme.

Starting point for optimization. A very common approach for performance-oriented implementations, typically with platform-specific optimizations, is to start from a reference implementation. The first step is to identify the routines that take most of the CPU cycles and then step-by-step replace those with optimized routines, often written in assembly and targeting a specific microarchitecture. To enable this approach, it is important that the reference implementation builds for and runs on the platform targeted for optimization. In particular, when considering embedded platforms, the use of large external libraries is often a problem.

Use in protocol experiments. Through the course of the NIST PQC standardization project, various efforts have investigated how to upgrade *protocols* to post-quantum security, using the primitives (and implementations) submitted to NIST; see, e.g., [29], [28], [30], [31], [32]. Integration into cryptographic-protocol frameworks often requires namespacing of the code. A meaningful performance evaluation additionally requires implementations that are optimized for the benchmark platform and adhere to all security guidelines requested by that platform, most notably, not leaking secrets through timing.

Use in (performance-uncritical) production software experiments. We saw some early adopters experimenting with post-quantum primitives in production software; some prominent examples are Google's and Cloudflare's post-quantum TLS experiments [33], [34], [35] and Infineon's implementations of post-quantum cryptography in contactless smartcards [36] and in TPMs [37]. While server-side deployments typically need highly optimized software, a portable (reference) implementation may be perfectly reasonable for less performance-critical client-side deployment. This, however, requires that this implementation is secure in the sense of the threat model the respective application uses.

Portability. Many of these possible uses cases of a reference implementation require that code is portable to different platforms, both hardware (e.g., 32-bit vs. 64-bit platforms or platforms with different endianness) and software (e.g., different operating systems or compilers). In the PQC FAQ, NIST clearly states that *"key requirements are that the submission code should be written in a cross-platform manner [. . .]"*, however without clarifying exactly what this means.

Some of the use cases for a reference implementation have strong synergies. For example, optimized code is also more useful to generate test vectors, and code suitable for academic protocol-level experiments is more likely to also be suitable for use in production software. However, some of these possible use-case scenarios have opposing requirements on a reference implementation and, ironically, almost all of them have some requirements that do not help to promote understanding. Notably, pursuing higher performance often requires unrolling implementations or more advanced (non-"schoolbook") implementations of, e.g., field arithmetic. This may distract from the overall structure of the scheme.

3.2. The problems with "ANSI C"

The first problem with requesting software written in "ANSI C" is that the term is not well defined. As pointed out in a posting to NIST's pqc-forum mailing list by Saarinen, "ANSI C" is commonly understood to refer to the ISO/ANSI C90 standard, which does not even define the `long long` data type used by the API. This issue with the definition of "ANSI C" was clarified in an FAQ entry saying that *"implementations written in C99 and C11 are both perfectly fine"*. Furthermore, with regards to the use of the NTL library that is written in C++, NIST clarified that implementations making use of NTL should still be *"as ANSI C-like as possible, only using C++ functionality where absolutely required in order to interact with NTL"*.

However, these clarifications do not address two other issues inherent to the C programming language: the fact that the language is underspecified and that it offers very little support to the programmer to write safe and correct programs.

Underspecification of C. The C programming language is intentionally underspecified to enable compilation to very fast code on a broad range of platforms. Krebbers [38] summarizes three different kinds of underspecification:

- *Implementation-defined behavior* leaves it to the compiler to make decisions about the semantics of certain expressions. These decisions need to be consistent, i.e., when compiling code for one target, all occurrences of the same kind of expression are required to have the same semantics. Also, these semantics should be documented. It's is near impossible to write (cryptographic) software without any expression that falls into this category. For instance, even the number of bits in

one byte (`char`) is not fixed by the C semantics. Whether `char` is signed or unsigned is also left to implementations. An example of implementation-defined behavior that causes more issues in real-world implementations is the endianness of integers.
- *Unspecified behavior* leaves it to the compiler to define the semantics of certain expressions. These decisions do *not* have to be consistent throughout the compilation and also do not need to be documented. One example is the evaluation order. Carefully written code avoids the pitfalls of unspecified behavior, either by entirely avoiding expressions that fall into this category or by ensuring that semantics does not depend on the compiler's decision.
- *Undefined behavior* allows the compiler to generate literally any code; for example, it would be within the specification of the C programming language to generate a binary that deletes all data in the user's home directory when encountering a case of undefined behavior. The most notable examples of undefined behavior are related to memory safety (i.e., reading or writing out of bounds), but this also includes, for example, signed-integer overflow, division by zero, modifications of string literals, or dereferencing a NULL-pointer. Undefined behavior in a program is generally a bug and very often a security-critical one.

Trusting the programmer. The C programming language, by its design philosophy, gives a lot of power to the programmer but also puts a lot of trust in the programmer. For example, C by itself has no mechanism to prevent programmers from accessing memory at invalid locations, has no mechanisms to ensure that all heap allocations are eventually freed (and freed only once), and has no mechanism to check for integer overflows. C does not guarantee that variables are initialized before they are read, it also features a rather weak type system with somewhat unintuitive rules for implicit casts. This all together makes C a great programming language to write highly optimized software, but it also makes it very easy to write programs with bugs that often have severe security implications – in particular in cryptographic software.

3.3. Software-engineering issues

Many issues with software implementations submitted to NIST PQC could have been avoided by following standard software-development practices:

Compiler warnings. A first step to avoid common pitfalls is to enable compiler warnings and ensure that code compiles without any warnings. This may sound like a rather straight-forward thing to do, but it turns out that "compilation without warnings" is much more complex. First, one needs to determine what warnings should be enabled; enabling "all" warnings with `-Wall` in `gcc` or `Clang` does by far not enable all warnings, neither do the "extra" (`-Wextra`) or "pedantic" (`-Wpedantic`) warnings levels. Compilation with the Microsoft compiler uses different flags and, unsurprisingly, also issues different warnings. However, even standard warning levels are suitable to identify many common patterns that are typically related to bugs.

Static analysis. More generally, there exist many tools for static analysis of C code, i.e., tools that analyze code without actually running it [39], [40], [41], [42]. These tools are typically able to find larger classes of bugs than those identified through compiler warnings. Aside from the general-purpose static-analysis analysis tools, there exist also tools specifically to check that cryptographic software is free of timing leaks [43]. However, the use of these tools is not a default practice even in the development of widely used cryptographic libraries [44]. One example of a bug that, for example, `gcc`'s static analyzer is able to catch is in the following piece of code that we found in the reference implementation of a NIST round-1 submission:

```
int64_t* extEuclid(int64_t a, int64_t b) {
    int64_t array[3];
    int64_t *dxy = array;
    ...
    return dxy;
}
```

The problem with this piece of code is that the function returns a pointer to a local stack variable. If the calling code dereferences this returned pointer—and what else would it do with a pointer?—the result is undefined behavior. In principle both `gcc` (via the `-Wreturn-local-addr` flag) and `clang` (via the `-Wreturn-stack-address` flag) issue compiler warnings for this kind of behavior. However, in our experiments, the indirection through `dxy` hides the bug from this compile-time analysis. The reason that this bug did not actually trigger undefined behavior in this submission is simply that the function was never called from any context – spotting such dead code is another use case for static analysis.

Dynamic analysis. Another approach to finding bugs is running the code with instrumentation or inside special environments. The most commonly used tools for dynamic analysis, at least under Linux, are Valgrind [45], [46], and the address-sanitizer [47] and undefined-behavior sanitizer [48] included with the Clang compiler. Like for static analysis, there also exist specialized tools to identify timing leaks through dynamic analysis [16].

Testing. Extensive testing is still one of the cheapest and most widely used techniques to ensure that software behaves as intended. NIST provided a rather minimalistic framework to generate test vectors for regression and compatibility testing to be used by submitters. Unfortunately, the framework did not include any negative tests (i.e., tests that ensure failure on invalid inputs) or basic tests that the API was used as intended. For example, one of the reference implementations submitted to round 1 of NIST PQC used out-of-bound accesses of the form

```
sk[CRYPTO_SECRETKEYBYTES + j]
```

for positive values of `j` to access bytes of the *public key*. The implementation simply made the assumption that the public key happens to be stored behind the secret key in memory. This is clearly something that any reasonable testing framework should catch.

We do not mean to suggest that submitters to public cryptographic standardization efforts like NIST PQC should be familiar and up-to-date with all the intricacies of these tools for engineering (cryptographic) software. On the contrary: Our proposal, which we detail in the next section, is that the standardization body soliciting submissions ensures a basic level of code quality by providing a suitable code-analysis and testing framework.

4. Proposed features of a software framework for cryptographic competitions

In this section, we propose a testing framework for software submitted to future cryptographic competitions. This section aims to be technology agnostic; we discuss our specific realization, PQClean, which accomplishes many of these goals, in Section 5. Where necessary, we focus on the C programming language as it remains prevalent for cryptographic software.

Our proposal could be implemented either as an offline solution using virtualization, or online using continuous integration. The benefit of the latter is that most testing can be executed in the cloud without requiring setup by or using resources of each submission team.

We propose that such a testing framework be made available together with the call for proposals, or alternatively, not later than 6 months before the submission deadline for software implementations.

The framework should include at least the following features.

Build system. The framework should include a build system that enables a reasonable level of compiler warnings and refuses to compile if any warnings exist. This would help to avoid many obvious mistakes and would save many hours for other researchers to fix the same bugs.

Provide a working example. Along with the testing framework, a working example should be provided to serve as a reference of what is expected from submitters. In the case of NIST PQC standardization, this could have been a KEM and signature scheme based on RSA or elliptic curves.

Automated functional tests. Straightforward functional tests should be implemented. For example, for digital signature schemes, a generated signed message should verify correctly. Failure test cases should also be included, e.g., a signed message should not verify under a different public key.

Verify test vectors. In addition to asking submission teams to submit test vectors, the framework should check if the software satisfies the test vectors. We believe that this is best done by including a hash of the test vectors in the submission, saving space in the case of large parameter sets. (Several round 1 submissions in the NIST PQC standardization project had KAT file archives in excess of 75 MB.)

Provide code building blocks. The framework should provide core functionality that is likely to be used by most schemes. For the NIST PQC competition, this primarily consists of hash functions (e.g., SHA-2, SHA-3), extendible output functions (XOFs) (e.g., SHAKE), the AES function, and functions outputting random bytes. This ensures that performance differences between implementations are not due to different implementations of the same building blocks. Requiring implementations to use the same APIs also eases evaluation and modifications by other teams. Submissions should not be allowed to ship their own version of the same functions, though the competition organizers will need to consider how to deal with requests for specialized versions of these functions (e.g., vectorized implementations).

Test using all major toolchains. The submitted software should support all major toolchains. In the case of the C programming language, at least gcc, clang, and Microsoft's compiler (CL) should be supported. The framework should make sure that the code compiles with all of them. As compilers change over time, it is important to fix a version for each of them.

Test on all major platforms. To ensure that submitted code is indeed platform-independent, the tests should be executed on a variety of platforms, including 32-bit and 64-bit systems, and little-endian and big-endian architectures.

Leverage modern static and dynamic analysis. The framework should enable the static analysis included in the toolchains. Additionally, Valgrind and AddressSanitizer should be used for dynamic software analysis for detecting memory problems.

Enforce namespacing. As implementations from multiple submissions are likely going to be used in the same software (e.g., in a library) their namespaces should be properly separated. The framework should enable and enforce appropriate namespacing and visibility, requiring unique names for public API functions (e.g., `mykem_level1_encaps` rather than `crypto_kem_encaps`) and unique names or limited visibility for internal symbols (e.g., static functions within a compilation unit).

Enforce code style and documentation. To improve the readability of code, submissions should be formatted in the same way. The call for submissions should include the coding guidelines and the framework should check if the code is formatted accordingly. In the same vein, a common syntax for documenting code should be defined. The framework should automatically check if a bare minimum of documentation for each function in the source code exists.

Benchmarking code. The framework should allow basic benchmarks to ensure that all teams run benchmark their code in the same way. The benchmarking results should be reported in the submission document. Advanced benchmarking (multiple platforms, multiple compilers) may be provided by the competition organizer, in which case it should be a public platform with clear submission procedures and transparent results reporting, or can be taken on by third-party projects like SUPERCOP.

Additionally, the framework could include the more advanced features in the following.

Verify that code is constant time. Code intended for use in actual software needs to have runtime independent of any secret data, i.e., avoiding branches depending on secrets, secret-dependent memory accesses, and variable-time instructions depending on secrets. Tools like ct-verif [43] could be used to detect such timing leakage. Alternatively, dynamic checking through Valgrind with uninitialized secret data can be used to catch most of the variable-time code. For some code (e.g., rejection sampling) this approach may result in false positives. In that case, submitters need to be able to mark the finding as a false positive and provide a rationale for why it is not a security issue.

Disallow dynamic memory allocation. Dynamic memory allocations present a problem on smaller bare-metal platforms. Additionally, they are often a source for bugs that would have been prevented by exclusively using stack memory. Thus, we recommend disallowing dynamic memory allocation and enforcing it using a test. In the majority of cases, cryptographic code is only using fixed-sized buffers which makes switching to stack memory straightforward. If variable-sized buffers are needed, the software can usually be re-written to allocate the worst-case size. Admittedly disallowing dynamic memory allocation can cause other problems. Some PQC algorithms have rather large memory usage, and some platforms do have problems with large stack sizes, especially within threads. Typically, 8 MB of stack is the limit on Linux. If such large buffers are required, dynamic memory allocation may be acceptable.

To lower the burden for initial software submissions, some of the requirements could be optional, i.e., failing tests will merely trigger a warning. In subsequent evaluation rounds, more requirements could become mandatory to gradually increase the quality of implementations. In case any of the requirements are not fulfilled in the submission, we recommend publicly disclosing a list of problems with each submission. The submission team should then be able to remedy the issues within a reasonable time frame. For this to work transparently, code should be hosted in a code versioning system (e.g., git) that is accessible to everyone. Note, however, that there need to be clear guidelines on the scope of updates allowed. As specifications are usually frozen during each evaluation round, changes that alter test vectors or algorithmic interoperability should not be allowed while algorithm specifications are meant to be frozen.

5. The PQClean framework

PQClean is not a software library; the schemes and implementations all exist independently and PQClean offers no API other than the scheme's interface. We organize the implementations in PQClean like in the SUPERCOP [1] project: the implementations are organized by type (KEM or signature scheme), then by scheme and specific instantiation (e.g., Kyber512). Each instantiation might have several implementations. PQClean supports C and assembly code; for some schemes we also have ARMv8-A or AVX2-specific code.

5.1. Common files

PQClean makes some common primitives available to each implementation. This includes (incremental) hashing primitives, AES, and random number generators. Anyone extracting implementations from PQClean can re-use these, or provide their own implementation based on our API. For hashing and encryption primitives, we additionally provide initialization and cleanup functions. This allows them to be implemented by heap-based primitives, as for example provided by OpenSSL [49]. PQClean does not attempt to offer the most efficient or any machine-optimized implementations of the primitives.

5.2. Meta information

In each instantiation folder, there exists a `META.yml` file, in which some scheme metadata is tracked. This information includes some scheme-specific information, like authors; instantiation-specific information, like key sizes; and information on each of the implementations present, like version and optionally compatibility information. This machine-readable information can be helpful for any automated tool using the framework, including the internal testing framework, as well as for any projects that generate code that wraps implementations from PQClean.

5.3. Namespacing

PQClean enforces that all exported symbols, like function names and global values, have a predictable and unique name. They are "namespaced" by prefixing with `PQCLEAN_`, then the name of the scheme and parameter set, and the name of the implementation. This ensures that no symbols conflict between, for example, different schemes or different implementations. Without this separation, it would be difficult to build software that uses several primitives or selects implementations based on CPU feature detection.

5.4. Automated testing

Currently, there are 22 automated tests in the PQClean testing harness, against which each scheme is evaluated. The tests range from simply compiling the source code with compiler warnings or checking the existence of license files, to the parsing of source files to exclude certain types of patterns. These tests include the following; symbols indicate which flaws in Table 1 the test can potentially identify:

- ⋆ that the scheme compiles correctly, without compiler warnings;
- ♠ `Makefile` correctness: that all scheme source files are correctly specified as dependencies;
- ♣ functional correctness: that the key generation, KEM, and signature operations function as intended, even on unaligned buffers; we also check if corrupted ciphertexts and signatures fail to verify;
- † that scheme keys, ciphertexts, and signatures match test vectors, including NIST's KAT test vectors;
- ◇ running functional tests with sanitizers (clang's address sanitizer [47], memory sanitizer [50], and undefined behavior sanitizer [48]) and Valgrind [46];
- ± specification of signedness of `char`;
- = existence of certain timing-suspicious boolean operations;
- ✝ `clang-tidy` [39] linting and static analysis;
- © existence of license files;
- • symbol namespacing;
- • no usage of dynamic memory;

TABLE 1. FLAWS FOUND, AND WHICH TESTS MIGHT HAVE DETECTED THEM, IN HOW MANY OF THE 10 KEMS AND 7 SIGNATURE SCHEMES THAT HAVE EVER BEEN INCLUDED IN PQCLEAN.

Flaw	KEMs	Sigs	**Flaw**	KEMs	Sigs		**Test**
Memory safety ◇	3	4	Endianness assumptions †	7	2	∗	Compilation test
Signed integer overflow ⋆, ◇, ✝	3	1	Platform-specific behavior ♣, ±, †, ✝	4	0	♠	`Makefile` checks
Alignment assumptions ⋆, ♣, ◇	4	4	Variable-Length Arrays ⋆	4	1	♣	Functional tests
Other Undefined Behavior ⋆, ♣	1	1	Compiler extensions ⋆	5	2	†	Test vectors
Dead code ⋆, ♠	3	4	Integer sizes ◇, ⋆, †	6	3	◇	Sanitizers
Global state	2	1	Non-constant time =	4	0	±	Signedness of `char`
Licensing unclear ©	3	1				=	Timing-suspicious ops.
						✝	`clang-tidy`
						©	License file

- consistent style and formatting.

The testing framework is based on Pytest [51], which allows us to generate tests for each scheme and implementation flexibly, and which generates convenient output. For tests that compile code, we isolate the source files and compilation targets so that tests can be executed in parallel.

5.5. Testing platform and platform diversity

The automated tests are run on each commit and pull request to PQClean. We also run them periodically on the `master` branch. This ensures that the implementations continue to be validated as compilers and tools get updated.

The testing platform is based on Github Actions [52]. This service provides Linux, Windows, and macOS runners, on which we run our automated tests. We run all tests on Linux and macOS with a recent version of GNU GCC as well as with Clang. Windows tests use Microsoft's CL compiler. Results are publicly visible through Github's user interface.

Although the native architecture of these systems is the 64-bit, little-endian Intel x64 architecture, we also run tests on 32-bit Intel x86, on 32-bit ARMv7 and 64-bit ARMv8, and on big-endian PowerPC. We use user-mode QEMU [53] emulation together with Linux's `binfmt_misc` capability [54] to run our tests within Docker images that emulate these targets.

Due to the large number of tests being run on every implementation, we have split the CI jobs per each implementation, operating system, compiler, and architecture. Otherwise, we quickly exceed the maximum allowed runtime for each job (5 hours). On pull requests, we only run the tests on the affected scheme, if possible. This makes testing times manageable and keeps feedback cycles short.

5.6. Results

We have integrated over 230 implementations of multiple parameter sets of 17 schemes into PQClean over the course of the project. In almost every scheme we identified "unclean" code, ranging from missing casts to memory safety problems and other forms of undefined behavior. In Table 1 we provide a summary of the number of schemes affected by some of the more significant categories of problems. Many of these flaws were detected by our automated testing, as described in Section 5.4, but in the process of integration, we also solved many problems by hand. The symbols in the table correspond to the tests that might have detected the type of flaw.

Many of the flaws are simply detected by enabling compiler warnings. Solving these warnings probably took the most time when integrating schemes into PQClean. Although many of the reported warnings did not immediately mean the code had a security or correctness flaw, we found that enabling all warnings did help find those problems that were flaws, as well as improve the general code quality.

A perhaps surprising issue was the uncertainty around licensing of 4 of the 17 schemes. Although NIST required the submitters of code in submissions to grant "the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process" [17], it is unclear what the scope of "public review and evaluation" is exactly, and NIST did not require any specific open-source license. Several of the included schemes did not include clear licensing information with their implementations. We contacted their authors and found that many had intended to grant permissive licensing, or even CC0 copyright waivers, to their implementations. However, this often resulted in notable delays; one submission team never provided us with a license and we had to abandon including it in PQClean.

6. Beyond "cleaning C"

The various difficulties we discussed in the previous sections motivate the question of whether C is the appropriate programming language for the purposes of a cryptographic standardization project. This is perhaps as much a philosophical discussion as a technical one. We can not answer this question definitively for any future standardization project but feel it is worth highlighting some issues.

6.1. Is C a good fit for specifications?

The C programming language has compilers for almost every system under the sun, which makes it very attractive for experiments (implementation-specific behavior notwithstanding). However, for the purposes of documenting and explaining an implementation, it is perhaps less well suited. More expressive higher-level languages like Python are perhaps better suited for the role of "executable pseudo-code". Rust could perhaps have stood in as a low-level language that simply does not allow most of the problems that our testing system was designed to catch. Additionally, allowing implementations in computational algebra systems like SageMath [55] or Magma [56] would permit expressing the mathematical constructions very directly, not distracting a reader with the details of, for example, polynomial multiplication. For specifications, there have also been efforts such as hacspec [57] that aim to not only generate executable code, but also translate specifications to formal-verification frameworks like F* [58], EasyCrypt [59], or Coq [60]. We believe this pathway has the potential for powerful collaborations with the world of computer-aided high-assurance cryptography [61].

6.2. Other languages in PQClean

Although PQClean initially only collected cleaned-up reference C implementations of schemes, we now also have optimized C and assembly implementations of schemes that use platform-specific features like AVX2 or Neon. These implementations are subjected to the same tests as the reference implementations. It would be possible to extend this project to implementations in other programming languages as well. The cross-implementation testing of test vectors would grant more confidence that each implementation is correct and interoperable, especially if at least one of the implementations has been formally verified against a machine-readable specification in, e.g., hacspec [57].

6.3. Beyond standardization projects

The goal of PQClean could be described as building a repository of highly-tested, high-confidence implementations. We believe cleaned-up implementations are valuable to other projects. The Open Quantum Safe [4] and the pqm4 [3] projects already automatically integrate implementations from PQClean. We argue that there is value in such an approach for more algorithms and cryptographic libraries. Firstly, it allows focusing analysis and testing efforts. It can save developer time and energy to, e.g., implement automated timing side-channel testing centrally instead of in each individual project. Any such efforts would then benefit all the consumers of the implementations. Currently, it seems that often when a vulnerability like a side-channel leak is discovered, it affects many cryptographic libraries and the effort of patching is duplicated many times. A central repository would minimize the maintenance effort required. Thus, we do believe that a well-designed testing framework benefits not only the standardization effort itself, but may reach beyond into the phase of deployment.

7. Conclusions

This paper presented what we believe NIST and other bodies coordinating cryptography standardization competitions should do to improve the software quality of submitted code. Properly implemented, a set of guidelines together with a testing framework could benefit submitters, the community, and the standardization body itself. It will allow everyone to focus on what the competition is about: evaluating the candidate cryptographic schemes.

We believe that many of the recommendations in this paper are uncontroversial and should be implemented in any future competition. For example, providing a working example of what is expected from submitters together with a testing framework would be the bare minimum. The scope of the testing framework may be more controversial and one has to be careful to not raise the bar for submissions too high. Limited resources at standardization bodies may also limit the features of such a framework.

More controversially, we question if C is a suitable programming language for reference implementations, especially if the main goal is clarity of the implementation. While as of now there seems to be no consensus on which alternative should be used, standardization entities should revisit this on a regular basis.

Acknowledgements

We would like to everyone who contributed to the PQ-Clean project, either in submission of schemes, additions to the testing framework or reporting and fixing of bugs. In particular, we thank Joost Rijneveld and John Schanck for their numerous contributions.

References

[1] "eBACS: ECRYPT benchmarking of cryptographic systems," 2018. [Online]. Available: https://bench.cr.yp.to/

[2] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "PQClean." [Online]. Available: https://github.com/PQClean

[3] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "PQM4: Post-quantum crypto library for the ARM Cortex-M4." [Online]. Available: https://github.com/mupq/pqm4

[4] D. Stebila and M. Mosca, "Post-quantum key exchange for the Internet and the Open Quantum Safe project," in *Proc. 23rd Conference on Selected Areas in Cryptography (SAC) 2016*, ser. LNCS, R. Avanzi and H. Heys, Eds., vol. 10532. Springer, October 2017, pp. 1–24. [Online]. Available: https://github.com/open-quantum-safe/

[5] R. J. Anderson, "Why cryptosystems fail," in *ACM CCS 93*, D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, Eds. ACM Press, Nov. 1993, pp. 215–227.

[6] B. Schneier, "Cryptographic design vulnerabilities," *Computer*, vol. 31, no. 9, pp. 29–33, 1998.

[7] P. Gutmann, "Lessons learned in implementing and deploying crypto software," in *USENIX Security 2002*, D. Boneh, Ed. USENIX Association, Aug. 2002, pp. 315–325.

[8] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does cryptographic software fail? A case study and open problems," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, ser. APSys '14. ACM, 2014.

[9] J. Blessing, M. A. Specter, and D. J. Weitzner, "You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries," arXiv:2107.04940, Jul. 2021. [Online]. Available: https://arxiv.org/abs/2107.04940

[10] A. Langley, "Apple's SSL/TLS bug," Feb. 2014. [Online]. Available: https://www.imperialviolet.org/2014/02/22/applebug.html

[11] N. Mouha, M. S. Raunak, D. R. Kuhn, and R. Kacker, "Finding bugs in cryptographic hash function implementations," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 870–884, 2018.

[12] A. Braga and R. Dahab, "A survey on tools and techniques for the programming and verification of secure cryptographic software," in *Proc. XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg) 2015*, 2015, pp. 30–43.

[13] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, “HACL*: A verified modern cryptographic library,” in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 1789–1806.

[14] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, “Jasmin: High-assurance and high-speed cryptography,” in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 1807–1823.

[15] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, “High-assurance cryptography in the Spectre era,” in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1884–1901.

[16] M. Neikes, “TIMECOP: Automated dynamic analysis for timing side-channels,” 2020. [Online]. Available: https://www.post-apocalyptic-crypto.org/timecop/

[17] National Institute of Standards and Technology, “Submission requirements and evaluation criteria for the post-quantum cryptography standardization process,” Dec. 2016. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf

[18] ——, “PQC - API notes,” Dec. 2016. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/example-files/api-notes.pdf

[19] ——, “PQC – known answer tests and test vectors,” Dec. 2016. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/example-files/kat.pdf

[20] ——, “Source code files for KATs,” Dec. 2016. [Online]. Available: https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/example-files/source-code-files-for-kats.zip

[21] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and Y.-K. Liu, “Status report on the first round of the NIST post-quantum cryptography standardization process,” National Institute of Standards and Technology, Tech. Rep. NISTIR 8240, Jan. 2019. [Online]. Available: https://doi.org/10.6028/NIST.IR.8240

[22] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, “Status report on the second round of the NIST post-quantum cryptography standardization process,” National Institute of Standards and Technology, Tech. Rep. NISTIR 8309, Jul. 2020. [Online]. Available: https://doi.org/10.6028/NIST.IR.8309

[23] National Institute of Standards and Technology, “Guidelines for submitting tweaks for third round finalists and candidates,” Jul. 2020. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/guidelines-for-sumbitting-tweaks-third-round.pdf

[24] N. Drucker and S. Gueron, “A toolbox for software optimization of QC-MDPC code-based cryptosystems,” *Journal of Cryptographic Engineering*, vol. 9, no. 4, pp. 341–357, Nov. 2019.

[25] M. J. Kannwischer, J. Rijneveld, and P. Schwabe, “Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on cortex-M4 to speed up NIST PQC candidates,” in *ACNS 19*, ser. LNCS, R. H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung, Eds., vol. 11464. Springer, Heidelberg, Jun. 2019, pp. 281–301.

[26] W.-K. Lee, H. Seo, Z. Zhang, and S. Hwang, “TensorCrypto,” Cryptology ePrint Archive, Report 2021/173, 2021, https://eprint.iacr.org/2021/173.

[27] V. B. Dang, F. Farahmand, M. Andrzejczak, K. Mohajerani, D. T. Nguyen, and K. Gaj, “Implementation and benchmarking of round 2 candidates in the NIST post-quantum cryptography standardization process using hardware and software/hardware co-design approaches,” Cryptology ePrint Archive, Report 2020/795, 2020, https://eprint.iacr.org/2020/795.

[28] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, “Post-quantum authentication in TLS 1.3: A performance study,” in *NDSS 2020*. The Internet Society, Feb. 2020.

[29] E. Crockett, C. Paquin, and D. Stebila, “Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH,” Cryptology ePrint Archive, Report 2019/858, 2019, https://eprint.iacr.org/2019/858.

[30] P. Schwabe, D. Stebila, and T. Wiggers, “Post-quantum TLS without handshake signatures,” in *ACM CCS 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM Press, Nov. 2020, pp. 1461–1480.

[31] A. Hülsing, K.-C. Ning, P. Schwabe, F. Weber, and P. R. Zimmermann, “Post-quantum WireGuard,” in *2021 IEEE Symposium on Security and Privacy"*. IEEE Computer Society Press, 2021 (to appear). [Online]. Available: http://eprint.iacr.org/2020/379

[32] D. J. Bernstein, B. B. Brumley, M.-S. Chen, and N. Tuveri, “OpenSSLNTRU: Faster post-quantum TLS key exchange,” in *Proceedings of the 31st USENIX Security Symposium*. USENIX Association, 2022 (to appear). [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein

[33] A. Langley, “CECPQ1 results,” Blog post, 2016. [Online]. Available: https://www.imperialviolet.org/2016/11/28/cecpq1.html

[34] ——, “CECPQ2,” Blog post, 2018. [Online]. Available: https://www.imperialviolet.org/2018/12/12/cecpq2.html

[35] K. Kwiatkowski and L. Valenta, “The TLS post-quantum experiment,” Post on the Cloudflare blog, 2019. [Online]. Available: https://blog.cloudflare.com/the-tls-post-quantum-experiment/

[36] Infineon, “Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip,” Infineon Press Release, 2017. [Online]. Available: https://www.infineon.com/cms/en/about-infineon/press/press-releases/2017/INFCCS201705-056.html

[37] ——, “Future-proof security solution: Infineon launches worlds first tpm with a pqc-protected firmware update mechanism,” Infineon Press Release, 2022. [Online]. Available: https://www.infineon.com/cms/en/about-infineon/press/market-news/2022/INFCSS202202-051.html

[38] R. Krebbers, “The C standard formalized in Coq,” Ph.D. dissertation, Radboud University Nijmegen, 2015. [Online]. Available: https://robbertkrebbers.nl/thesis.html

[39] The Clang Team, `clang-tidy` – *Extra Clang Tools 15.0.0git documentation*. [Online]. Available: https://clang.llvm.org/extra/clang-tidy/

[40] C. Calcagno and D. Distefano, “Infer: An automatic program verifier for memory safety of C programs,” in *NASA Formal Methods*. Springer, 2011, pp. 459–465. [Online]. Available: https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.421.9629

[41] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “A static analyzer for large safety-critical software,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 196–207. [Online]. Available: https://www.di.ens.fr/~cousot/COUSOTpapers/PLDI03.shtml

[42] Sonarqube. [Online]. Available: https://sonarqube.com

[43] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying constant-time implementations,” in *USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX Association, Aug. 2016, pp. 53–70.

[44] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ““They’re not that hard to mitigate”: What cryptographic library developers think about timing attacks,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022 (to appear). [Online]. Available: https://cryptojedi.org/papers/#usect

[45] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 89–100. [Online]. Available: https://valgrind.org/docs/valgrind2007.pdf

[46] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 2.

[47] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '12. USA: USENIX Association, 2012. [Online]. Available: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[48] The Clang Team, *UndefinedBehaviorSanitizer – Clang 15.0.0git documentation*. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[49] OpenSSL. [Online]. Available: https://openssl.org

[50] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, USA, 2015, pp. 46–55.

[51] (2022) pytest: helps you write better programs. [Online]. Available: https://docs.pytest.org/en/7.0.x/

[52] Github. Github features – Actions. [Online]. Available: https://github.com/features/actions

[53] The QEMU Project Developers. Qemu. [Online]. Available: https://www.qemu.org

[54] R. Günther, "Kernel support for miscellaneous binary formats (`binfmt_misc`)." [Online]. Available: https://www.kernel.org/doc/html/v5.16/admin-guide/binfmt-misc.html

[55] The Sage Developers, *SageMath, the Sage Mathematics Software System*, 2022. [Online]. Available: https://www.sagemath.org

[56] W. Bosma, J. Cannon, and C. Playoust, "The Magma algebra system. I. The user language," *J. Symbolic Comput.*, vol. 24, no. 3-4, pp. 235–265, 1997, computational algebra and number theory (London, 1993). [Online]. Available: http://dx.doi.org/10.1006/jsco.1996.0125

[57] D. Merigoux, F. Kiefer, and K. Bhargavan, "hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust," Inria, Technical Report, Mar. 2021. [Online]. Available: https://hal.inria.fr/hal-03176482

[58] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 256–270. [Online]. Available: https://www.fstar-lang.org/papers/mumon/

[59] Easycrypt. [Online]. Available: https://github.com/EasyCrypt/easycrypt

[60] The Coq Development Team. (2022, Jan.) The Coq proof assistant. [Online]. Available: https://coq.inria.fr

[61] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 777–795.