Finding Bugs and Features Using Cryptographically-Informed Functional Testing

Giacomo Fenzi[1], Jan Gilcher[2], and Fernando Virdia[3]

[1] Computational Security Lab, EPFL, Lausanne, Switzerland
[2] Applied Cryptography Group, ETH Zurich, Zürich, Switzerland
[3] NOVA LINCS, Universidade NOVA de Lisboa, Caparica, Portugal
giacomo.fenzi@epfl.ch
jan.gilcher@inf.ethz.ch
f.virdia@fct.unl.pt

Abstract. In 2018, Mouha *et al.* (IEEE Trans. Reliability, 2018) performed a post-mortem investigation of the correctness of reference implementations submitted to the SHA3 competition run by NIST, finding previously unidentified bugs in a significant portion of them, including two of the five finalists. Their innovative approach allowed them to identify the presence of such bugs in a black-box manner, by searching for counterexamples to expected cryptographic properties of the implementations under test. In this work, we extend their approach to key encapsulation mechanisms (KEMs) and digital signature schemes (DSSs). We perform our tests on multiple versions of the LibOQS collection of post-quantum schemes, to capture implementations at different points of the recent Post-Quantum Cryptography Standardization Process run by NIST. We identify multiple bugs, ranging from software bugs (segmentation faults, memory overflows) to cryptographic bugs, such as ciphertext malleability in KEMs claiming IND-CCA security. We also observe various features of KEMs and DSS that do not contradict any security guarantees, but could appear counter-intuitive.

1 Introduction

As cryptography is adopted in a growing number of applications, it is often the case that implementation correctness of cryptographic libraries becomes a necessary condition for achieving overall system and application security. Implementations of cryptographic algorithms often need to satisfy the conflicting requirements of performing complex mathematical operations while achieving a high level of performance, resulting in generally hard-to-debug code. High-profile vulnerabilities have resulted from incorrect implementations of on-paper secure cryptographic schemes, such as the recent "psychic signatures" vulnerability in Java's ECDSA library (CVE-2022-21449)[4], or the buffer overflow bug in the Keccak reference implementation (CVE-2022-37454)[5].

[4] https://www.cve.org/CVERecord?id=CVE-2022-21449
[5] https://www.cve.org/CVERecord?id=CVE-2022-37454

While a long-term solution to the problem of developing robust implementations of cryptographic algorithms "from the get-go" will likely require more extensive adoption of formal verification tools, today it is still the case that initial reference implementations for cryptographic schemes are often written directly in system programming languages such as C, rather than starting from a formal specification. Often these reference implementations are then used to generate known-answer test vectors used to verify correctness of later implementations and may also be directly deployed as part of products, potentially resulting in the proliferation of bugs (e.g., CA-1999-15)[6].

In this work, we investigate the issue of automatically discovering bugs in implementations of cryptographic primitives, by searching for counterexamples to expected cryptographic properties of the implementations under test. This process can be seen as a form of metamorphic testing (Zhou *et al.*, ISFST 2004) [21], instantiated using fuzzing tools.

In a nutshell, our approach is to run "bit-contribution" and "bit-exclusion" tests on selected inputs to a cryptographic function. We make use of the AFL++ fuzzer [5], running on test programs specifically designed to crash whenever an expected cryptographic property of a function is not satisfied. For example, when testing IND-CCA2-secure key encapsulation mechanisms' decapsulation, we may generate a valid encapsulation c, modify it into $c' \neq c$, and attempt to decapsulate it; if decapsulation succeeds, we crash the program. Together with a custom mutator that generates c' by individually "flipping" every bit in c, AFL++ will then identify implementations that either internally crash, or that do not achieve IND-CCA2 security, flagging either issue. This approach, used on every (function, input) combination provided by the syntax of hash functions, key encapsulation mechanisms (KEMs), and digital signature schemes (DSSs), together with a cryptographically-informed test of their expected output properties and with appropriate de-randomisation, led to our discoveries.

Related work. Our starting point is the work of Mouha *et al.* [12] on hash functions proposed during the SHA3 competition run by the United States National Institute of Standards and Technologies (US NIST). We borrow their techniques and apply them to key encapsulation mechanisms and digital signature schemes, as well as running some of their tests on different hash functions. While the PQClean suite performs similar tests [11] on some KEM and DSS functions, they do so on a smaller selection of algorithms and only testing: the effect of replacing valid inputs with random tapes (for KEMs), whether replacing the public key with a different valid public key still allows signature verification to pass (for DSSs), and whether canary bytes around allocated buffers are touched during normal operation (for KEMs and DSSs). This implies that some of the bugs we encounter would likely not be detected.

Our work is also related to CLFuzz [20], which also focuses on testing cryptographic libraries via fuzzers that are aware of some of the semantics of crypto-

[6] https://vuls.cert.org/confluence/display/historical/CERT+Advisory+CA-1999-15+Buffer+Overflows+in+SSH+daemon+and+RSAREF2+Library

graphic algorithms. Therein, the fuzzing techniques are aware of the correctness semantics of cryptographic primitives, while our work tests for a wider set of semantics that our primitives should uphold. Yang, Arya and Wang [18] combine formal methods with fuzzing in order to discover bugs in complex systems such as 5G implementations. Our approach limits its scope to the underlying cryptographic primitives, rather than aiming to holistically target an entire system.

Our contributions. During the development of the project, we identified various cryptographic bugs, including: an IND-CCA2 bug for some parameter sets in the reference implementation of NTRU (fixed in 2020 after disclosure); a bug breaking output consistency of the reference implementation of the KNOT-384 hash function submitted to the first round of the NIST Lightweight Cryptography Standardization standardization process (independently fixed by the authors, albeit not explicitly disclosed), that would allow influencing output digests by modifying bytes neighboring (but not belonging to) the input buffer; second-preimage bugs in the AVX/SSE implementations of acehash and syconhash submitted to the NIST Lightweight Cryptography Standardization process. We remark that the bugs mentioned exclusively affect the specific implementations tested for each scheme, rather than their mathematical design.

Our tests also automatically detected a few surprising properties, including: the lack of strong unforgeability for the compressed signature format in the Falcon DSS (in version 1.1 of its spec, independently discovered by the Falcon team); that Rainbow and Falcon public keys pk could be modified such that verification using a specific $pk' \neq pk$ would still succeed (as of LibOQS 0.4.0, since resolved), a property of many KEMs using implicit-rejection Fujisaki-Okamoto transform variants where valid ciphertexts decapsulate correctly under modified secret keys; a similar property for the randomized version of the Dilithium DSS, where valid signatures can be generated by modified secret keys in the presence of a faulty randomness source. In contrast with the bugs that we previously mentioned, these properties result from the schemes' definition, and were identified by automatically testing an implementation. While these properties do not invalidate the security notions claimed (IND-CCA2 and EUF-CMA), some may be unexpected by users and could affect the interaction between the cryptographic primitives and the protocols these may be used as components of, similarly to what Jackson *et al.* [10] discover for non-post-quantum signature schemes using formal verification tools.

Finally, as part of our larger scale effort, we found that dealing with more complex primitives and minimising the engineering overhead of adding new tests can be obstacles to employing this methodology. To streamline this process we develop a syntax for the testing procedure, which, once implemented, lets us reuse a majority of the testing code across different primitives (hash functions, KEMs, DSSs), libraries (SUPERCOP for hash functions, LibOQS for KEMs and DSSs), and fuzzers (our final choice being AFL++) at minimal implementation cost. This syntax could easily be adapted to check for other properties that we did not test, such as near-collisions in hash functions, which could also result from potential bugs.

We believe our results further demonstrate the usefulness of the techniques introduced by Mouha *et al.*, which are successful in identifying subtle bugs and properties of implementations and schemes in a black-box manner. We think our formalisation of it simplifies the extension to other primitives and tests, and we released our implementation on `https://gitlab.com/fvirdia/crypto-fun-test`. We think that adoption of similar testing techniques (possibly as an optional "mode") by cryptographic algorithm testing suites such as SUPERCOP or Project Wycheproof [2] could help increase early detection of various bugs at a relatively low engineering cost for the cryptographic community.

Paper roadmap. In Section 2 we lay down preliminary notation and definitions. In Section 3 we recall how metamorphic testing can be used to stress hash function implementations, and explain how to extend this to KEMs and DSSs. In Section 4 we introduce our approach to implementing metamorphic testing on cryptographic schemes, and describe the specific tests we run. In Section 5 we describe the results of our experiments, the bugs we identify as well as some possibly unexpected properties found.

2 Preliminaries

2.1 Notation

We denote by $\mathbb{Z}_{\geq 0}$ the set of non-negative integers. Let 'true' and 'false' be denoted by $\top$ and $\bot$, respectively. Given a statement S, we denote its truth value by $[\![S]\!]$. Given a bit string s, we denote its bitlength as $|s|$. Given two strings s_1 and s_2, we denote $s_1||s_2$ as the concatenation of s_1 and s_2. We write $\oplus$ for the exclusive-or operation on single bits or on bitstrings having the same length. We write $(0^x10^y)_2$ to represent the bitstring of length $x+y+1$ with all bits set to zero except for the bit in position $x+1$, which is set to one. Whenever we write $m \bmod n$ we mean the value m' in $\{0,\dots,n-1\}$ such that $m' = m \bmod n$. We denote integer floor division of x by y as $\lfloor x/y \rfloor$.

In pseudocode, we denote variable assignment using $\leftarrow$, and random sampling from a set using $\leftarrow\!\$$. Whenever we are addressing a buffer x starting from the second byte, rather than the first, we write "$\&x[1]$" rather than "x".

Definition 1 (Stateful pseudorandom generator). *We define a "stateful" pseudorandom generator* PRG *to be a function mapping λ-bit "seeds" $s \in \{0,1\}^\lambda$ into random tapes $r \in \{0,1\}^*$ and new seeds,* $\mathsf{PRG}\colon \{0,1\}^\lambda \to \{0,1\}^\lambda \times \{0,1\}^*$.

We chose this syntax for PRGs to simplify describing our generation of random tapes during tests in Section 4.

Definition 2 (Key Encapsulation Mechanism (KEM)). *Let Π be a triple* $(\mathsf{Gen}, \mathsf{Encaps}, \mathsf{Decaps})$ *of algorithms where* $\mathsf{Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$ *takes a security* *parameter and outputs a public-secret keypair,* $\mathsf{Encaps}(\mathsf{pk}) \to (c, ss)$ *takes a public key* pk *and outputs a ciphertext* c *and a shared secret* ss*, and* $\mathsf{Decaps}(\mathsf{sk}, c) \to ss$ *is a deterministic algorithm that takes a secret key* sk *and a ciphertext* c *and outputs a shared secret* ss *(or a distinguished failure symbol* $\perp$*). We say* Π *is a* key encapsulation mechanism *(KEM), if*

$$1 - \Pr\left[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda), (c, ss) \leftarrow \mathsf{Encaps}(\mathsf{pk}), ss' \leftarrow \mathsf{Decaps}(\mathsf{sk}, c) \colon ss = ss'\right]$$

is negligible.

Definition 3 (Digital Signature Scheme (DSS)). *Let* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *be a triple of algorithms where* $\mathsf{Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$ *takes a security parameter and outputs a public-secret keypair,* $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$ *takes a secret key* sk*, a message* $m \in \{0,1\}^*$ *and outputs a signature* σ*, and* $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \in \{0,1\}$ *is a deterministic algorithm that takes a public key* pk*, a message* $m \in \{0,1\}^*$ *and a signature* σ *and outputs a bit* b*, with* $b = 1$ *meaning "valid" and* $b = 0$ *meaning "invalid". We say* Π *is a* digital signature scheme *(DSS) if*

$$1 - \Pr\left[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda), \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m) \colon \mathsf{Verify}(\mathsf{pk}, m, \sigma) = 1\right]$$

is negligible.

2.2 Security notions

In this section we recall security notions for hash functions, KEMs and DSSs, that will be "stressed" by our testing, meaning that the tests may output counterexample to the notion demonstrating that a specific implementation does not satisfy it. This is not meant as an exhaustive list of security properties.

Definition 4 (Second-preimage resistance). *Let* $H \colon \{0,1\}^m \to \{0,1\}^\ell$ *be a hash function.* H *is said to be* second-preimage resistant *if for any efficient adversary* $\mathcal{A}$ *and random message* $x \leftarrow\!\!\$\ \{0,1\}^m$*,*

$$\Pr[x \leftarrow\!\!\$\ \{0,1\}^m, x' \leftarrow \mathcal{A}(x) \colon H(x) = H(x') \text{ and } x \neq x']$$

is negligible.

Definition 5 (KEM IND-CCA security). *Let* $\Pi = (\mathsf{Gen}, \mathsf{Encaps}, \mathsf{Decaps})$ *be a KEM. We say that* Π *has* indistinguishable encapuslations under chosen ciphertext attacks *(IND-CCA) secure if for any efficient adversary* $\mathcal{A}$

$$\Pr\left[\begin{array}{r} b \leftarrow\!\!\$\ \{0,1\}, (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda), \\ (c^*, ss^*) \leftarrow \mathsf{Encaps}(\mathsf{pk}), b' \leftarrow \mathcal{A}^{O_{\mathsf{sk}}^{\mathsf{Decaps}}(\cdot)}(\mathsf{pk}, c^*, \overline{ss}) \end{array} \colon b = b'\right]$$

is negligible, where $O_{\mathsf{sk}}^{\mathsf{Decaps}}(c)$ *returns* $\mathsf{Decaps}(\mathsf{sk}, c)$ *except if* $c = c^*$*, in which case it returns* $\perp$*, and* $\overline{ss} = ss^*$ *if* $b = 0$ *and is randomly sampled from the shared secret space if* $b = 1$*.*

Definition 6 (sUF-CMA security). *Let $\Pi = (\mathsf{Gen}, \mathsf{Encaps}, \mathsf{Decaps})$ be a DSS. We say that Π is* strongly unforgeable under chosen message attacks *(sUF-CMA) secure if for any efficient adversary $\mathcal{A}$*

$$\Pr\left[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda), (m^*, \sigma^*) \leftarrow \mathcal{A}^{O_{\mathsf{sk}}^{\mathsf{Sign}}(\cdot)}(\mathsf{pk})\colon \mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1\right]$$

is negligible, where $O_{\mathsf{sk}}^{\mathsf{Sign}}(m)$ returns $(m, \mathsf{Sign}(\mathsf{sk}, m))$, and where (m^*, σ^*) was not output by $O_{\mathsf{sk}}^{\mathsf{Sign}}$.

3 Metamorphic testing

By design, the output of cryptographic primitives is often conjectured to be indistinguishable from random. When testing a new implementation P of a cryptographic primitive π, a common strategy is to compare the output of P with that of a reference implementation P_0 deemed correct. This can be done by generating test cases $(t_i)_i$, and checking that $P(t_i) = P_0(t_i)$.

This approach however poses the problem of testing the initial reference implementation P_0. Indeed, given an input t_i, it may be impossible to tell if $P_0(t_i)$ is correct, due to its pseudorandomness. If π is an encryption scheme, for example, one may test whether ciphertexts generated with P_0 correctly decrypt. However this may not be a sufficient criteria if multiple ciphertexts could plausibly decrypt to the same message. If π is a one-way function, such as a hash function, this may be even harder, since no inverse function would be known. This is an instance of the "oracle problem" in software testing [17].

We say that a test input t_i is *successful* if, given a known output $\pi(t_i)$, an implementation under test satisfies $P(t_i) = \pi(t_i)$. We say t_i is *indeterminate* if $P(t_i)$ does not obviously fail (say, by crashing the program). We say t_i is *unsuccessful* if $P(t_i) \neq \pi(t_i)$. Metamorphic testing [4,3] is a testing approach where a series of test cases $(t_i)_i$ that are indeterminate can be transformed into new test cases $(t'_j)_j$ that may be possible to classify as unsuccessful, given the partial knowledge of π.

For example, Mouha *et al.* [12] use metamorphic testing to test hash functions, by creating indeterminate test cases $(t_i)_i$ for $i \in [n]$ by setting $t_0 = (0^n)_2$ be the zero-bitstring of length n, and $t_i = (0^{i-1}10^{n-i})_2$ be the bitstring with zeroes everywhere except at index i, for $i > 0$, and checking whether $P_0(t_i) = P_0(t_j)$ for some $i \neq j$ (among other tests). Whenever this happens, the a likely bug in the implementation is found, and (t_i, t_j) is marked as an unsuccessful test case, since collisions or second pre-images to the hash function π should be hard to find. This simple testing technique (the "Bit-Contribution" test, in [12]'s terminology) allowed them to identify 19 bugs in submissions to the SHA3 standardization competition.

3.1 Metamorphic testing for Hash functions

We now describe the original set of metamorphic tests introduced in [12] for cryptographic hash functions. We will use these as the initial point for our further tests on KEMs and DSSs.

For context, during the SHA3 competition, submitters had to design a hash function able to take inputs of any bitlength in $\mathbb{Z}_{\geq 0}$ (with 0-bit inputs corresponding to the empty string), and output digests of length in $\{224, 256, 384, 512\}$. Due to the possibility of having hash functions have to compute digests over very large bitstrings that could potentially be streamed between servers, in order to facilitate progressive hashing (rather than having to wait until the transmission had completed to start hashing), the reference implementation for the function $\mathsf{Hash}(\cdot)$ would have consisted of one call to $\mathsf{Init}(\cdot)$ which would generate some initial state σ, one or more calls to $\mathsf{Update}(\sigma, \ell)$ which would update the state σ by processing ℓ bits of input, and $\mathsf{Final}(\sigma)$ which would output the resulting digest.

Mouha *et al.* [12] introduce three kinds of metamorphic test: the "Bit-Contribution", "Bit-Exclusion", and "Update" tests.

Bit-Contribution. The *bit-contribution* test defines a series of input bitlengths $(\ell_i)_i \subset \mathbb{Z}_{\geq 0}$, and for each ℓ_i defines an initial test case $t_{i,0} \in \{0,1\}^{\ell_i}$. After mauling this test case by defining $t_{i,j} \coloneqq t_{i,0} \oplus (0^{j-1}10^{\ell_i - j})_2$ for $0 < j \leq \ell_i$, it collects $\mathsf{Hash}(t_{i,j})$ for all i, j as above, and checks for collisions. The rationale of this test is that a collision would potentially be caused by some bit of the input not contributing to the output in the implementation, likely meaning a bug is present in the implementation.[7] Of the 86 reference implementations tested in [12], 19 fail this test.

Bit-Exclusion. The *bit-exclusion* test targets the fact that NIST required the SHA3 competition submitters to define their hash functions over non-multiple-of-8 bitlenghts. However, in practice implementations in most languages work at byte-level, including the reference implementations that were requested to be written in the *C* programming language. This means that, for example, if a function was called on an input of length $\ell = 6 \bmod 8$, the last 2 bits of the input *byte buffer* should be ignored, since they don't actually belong to the input bitstring. The bit-exclusion test is designed to check this property, by defining input test cases $t_{i,0}$ for $\ell_i \in \{\ell \in \mathbb{Z}_{\geq 0} \mid \ell \bmod 8 \neq 0\}$, and then mauling these into further test cases $t_{i,j} \coloneqq t_{i,0} \oplus (0^{\ell_i} 0^{j-1} 10^{8-(\ell_i \bmod 8)-j})_2$, for $0 < j \leq 8 - (\ell_i \bmod 8)$. For each i, it then checks for non-collisions in $\{\mathsf{Hash}(t_{i,j})\}_j$. If given some length ℓ_i, non-collisions are generated, this means that the implementation of Hash is incorrectly over-reading bits beyond the input boundary when producing the final digest. Of the 86 reference implementations tested in [12], 17 fail this test.

[7] While a fundamental weakness of the design could be possible, this does not seem to be the case for any of the tested primitives.

Update. The *update* test makes use of the guarantee on the internal API of Hash, by checking whether $\mathsf{Hash}(x) = \mathsf{Final}(\cdot) \circ \mathsf{Update}(\cdot, \ell_2) \circ \mathsf{Update}(\cdot, \ell_1) \circ \mathsf{Init}(x)$ for various values of x, such that $\ell_1 + \ell_2 = |x|$. Of the 86 reference implementations tested in [12], 32 fail this test.

3.2 Metamorphic testing for KEMs and DSSs

As seen above, the core metamorphic test performed in [12] on hash functions is whether outputs collide or not, depending on whether they are expected to (bit-exclusion and update tests) or not (bit-contribution tests). The reason for only checking for collisions is that hash functions only provide a method Hash, which is expected to be one-way. This means that one cannot check for interactions of the output of Hash with other cryptographic methods.

Extending metamorphic testing to KEMs and DSSs presents us with more functions to use compared to hashes, but also with a few syntactic differences that should be addressed.

Testable functions. In the case of KEMs and DSSs, both primitives provide three methods, $(\mathsf{Gen}, \mathsf{Encaps}, \mathsf{Decaps})$ and $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ respectively, presenting structure when composed with each other. This means that we can use bit-contribution and bit-exclusion ideas to stress various aspects and security properties of such primitives. This also means that more tests will have to be run, when testing each function. Unfortunately, for the purpose of PQC standardisation, NIST did not mandate a specific internal API. Therefore, we don't define any "update" tests, unlike [12].

Randomness. Unlike hash functions, Gen, Encaps and Sign functions use random tapes as one of their inputs. This adds some variability to testing, which can be resolved by fixing a PRG seed used when generating the random tape. However, the use of random tapes also suggest a different test that can be performed on randomised algorithms: providing bad randomness. This may not always be easy to implement, depending on the internal API of the implementation being tested. Luckily, LibOQS does allow defining a custom PRG, such as one that repeatedly outputs a fixed byte. While the setting of having access to a bad randomness source is usually out-of-model in terms of cryptographic vulnerabilities, by performing this test we do identify some implementations that potentially hang if they get an unlucky random tape, which likely is not desired behaviour. By adding a check for low hamming weight outputs (which we do not perform as part of our tests, but did observe by manually inspecting outputs), one can also notice that certain implementations chose to use the output of the PRG function call defined by NIST ("`randombytes`") directly, while others may be further hashing it. For the latter, their output under bad randomness does not appear obviously different from output obtained using the correct PRG.

Fixed-length inputs. Another difference with testing hash functions is that often inputs to KEM and DSS methods are fixed-length. Exceptions are (in principle) random tapes, where however we provide fixed-lenghts PRG seeds, messages for Sign, where we consider fixed messages since anyway these are usually first hashed,[8] and Verify for some schemes where variable-length signatures are possible. In the latter case, we store the signature in a buffer large enough to always contain the largest possible signature σ output by Sign, together with an buffer encoding the bytelength ℓ_σ of the specific signature. We then consider the case of mauling both σ and ℓ_σ.

Format strings. Finally, as we will explain shortly in Section 4, for the purpose of reducing the amount of tests to be individually written, we will collect bit-contribution and bit-exclusion tests into a single test. The testing functionality will be given an input string s, that could contain multiple function inputs concatenated (say, $x = pk||sk||m||sig$ for a signature scheme test) and a "format string" tape, that captures whether one should expect the output of the function being tested on a mauled version of x to equal or differ from the output of the tested on the original s. We remark that this is not imposed by the primitives, and is rather an implementation choice. This helps reduce code duplication when writing the tests.

4 Implementing metamorphic testing

A main focus of our work is to reduce the friction for maintainers of cryptographic libraries in order to incorporate metamorphic testing techniques in said libraries. In order to achieve this goal, we develope a metamorphic framework that is general enough to encompass our test suite. In this section, we formalize this framework, and in Section 4.1 we describe how it will be instantiated to test the corresponding primitives.

Informally, our testing framework follows similar steps to those used in software fuzzing. First, we generate some valid input x with respect to some cryptographic primitive function Π and some public parameters pp. We then call Π on this input to obtain some baseline result y. We will apply some mutations to x in order to obtain a new input x'. On this mauled input we will apply again the tested function and receive a new output y'. Then, we will compare this y and y' to verify that the input mutation has caused the output to change (or not) in the way that we expected. In general, we assume non-malleability, meaning that we expect $\Pi(x) \neq \Pi(x')$.

In formalizing this, we define the a metamorphic test specification as follows.

Definition 7 (metamorphic test specification). *Let Φ = (GenInput, Call, Maul, Match) be a tuple of functions with the following syntax.*

– *GenInput* : $(\Pi, \mathrm{pp}, \mathrm{fmt}) \mapsto (x, y, \mathrm{aux})$*, takes an argument Π that specifies a cryptographic primitive's method implementation and how to execute it, some*

[8] And testing messages of different lengths would realistically only stress the underlying hash function being used by the DSS.

fixed public parameter pp*, and a formatting specifier* fmt *and returns a triple consisting of an input x, an output y, and some auxiliary information* aux*.*
- *Call* : $(\mathit{\Pi}, x, \text{aux}, \text{pp}, \text{fmt}) \mapsto y$*, takes $\mathit{\Pi}$,* pp*,* fmt *as in GenInput, auxiliary information* aux *and an input x, and returns an output y.*
- *Maul* : $(x, \text{fmt}, \sigma) \mapsto (x', \sigma, \text{expected\_result})$*, takes an input x, a format string* fmt*, some state σ and outputs a mutated input x', some updated state σ and a value* expected_result *that specifies the expected change that the mutated input should have on the result of Call.*
- *Match* : $(y, y', \text{expected\_result}) \mapsto b$*, compares y with y' on account of* expected_result *and raises an error on failure.*

Given a metamorphic test specification $\mathit{\Phi}$, we instantiate our testing framework as shown in Figure 1 to obtain a testing function $\mathsf{Test}(\mathit{\Pi}, \text{pp}, \text{fmt}, \text{runs})$.

```
Test(Π, pp, fmt, runs)
sus ← [ ];  σ ← ⊥;
(x, y, aux) ← GenInput(Π, pp, fmt)
assert(y ≠ crash)
// Fuzzer loop
foreach i in {1, . . . , runs}
  x′, σ, expected_result ← Maul(x, fmt, σ)
  try
    y′ ← Call(Π, x′, aux, pp, fmt)
    Match(y, y′, expected_result)
  catch
    sus.append((x, fmt, σ))
return sus
```

```
GenInput(Π, pp, fmt)
x ← // an initial valid input to Π
aux ← // auxiliary data for evaluating Call
y ← Call(Π, x, aux, pp, fmt)
return (x, y, aux)

Call(Π, x, aux, pp, fmt)
return // output of a call the Π API on x

Maul(x, fmt, σ)
return // a format-aware mutated input

Match(y, y′, expected_result)
// checks whether the implementation behaved
// as expected and crashes on failure
```

Figure 1: Generic test framework.

The advantage of defining a test specification, and strictly adhering to the defined interface during its implementation (for example, by defining custom generic types `aux_t` for auxiliary information aux, or `pp_t` for public parameters pp) is that the code implementing functions that do not have a cryptographic primitive's method implementation $\mathit{\Pi}$ as input (such as the Match and Maul functions, or as the Test function main loop) can be reused "as is" for different functions $\mathit{\Pi}$ being tested. Furthermore, we observe that in practice often some of the structure and code in $\mathit{\Pi}$-dependent functions such as Call and GenInput can also be reused between different tests.

Given a test specification $\mathit{\Phi}$, the Test function that we use, described in Figure 1, only checks for single (y, y') pairs, rather than collecting sets of multiple

$\{y_0, y_1, y_2, \dots\}$ for collisions, as done in [12]. While manually implementing such a test should be possible while reusing the code from Φ, the reason we define the Test over pairs is that it lends itself to a trivial adaptation to fuzzing libraries. In particular, we will use Φ to define a custom AFL++ mutator function that will allow us to use the parallelisation utilities provided by AFL++, while running our tests. Indeed, our Maul function is essentially a bit-flip deterministic mutator, similar to the one provided by the fuzzing library, the main difference being that we run a subset of all deterministic mutations considered by AFL++ to avoid an explosion of equivalent mutations being discovered. In order for the fuzzer to pick up an unexpected output from Π, whenever the Match function observes an unexpected result, we purposely crash it.

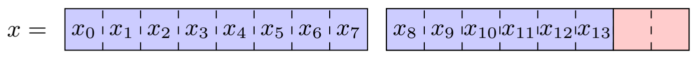
While defining a testing specification helps us normalise terminology and in principle implement more general testing, our main contribution is defining specific *cryptographically-informed* programs to fuzz, that not only are supposed to not crash due to software bugs, but that also may capture a security property, such that their crashing implies failure of the underlying implementation to uphold a cryptographic property.

Types. We implement our test code in the C programming language, since the reference implementations in LibOQS and SUPERCOP are similarly written in C. To keep our test code in line with the test specification Φ, we define some abstract types for inputs x (`in_t`), outputs y (`out_t`), public parameters pp (`pp_t`), auxiliary information aux (`aux_t`), formatting specifiers fmt (`fmt_t`) and expected results (`exp_res_t`).

- The method implementation Π and the public parameters `pp_t` will be specific to the cryptographic library being tested. In LibOQS' case, Π is implicitly imported with the library (`#include <oqs/oqs.h>`), while `pp_t` only contains an integer "algorithm identifier" entry which is used by the LibOQS interface to select the algorithm being used.
- `in_t` is a struct pointing to a byte buffer and storing its byte length.
- `out_t` is a struct pointing to a byte buffer, storing its length, and also storing an integer return value that the method being tested may return (e.g., `0` on correct decapsulation).
- `aux_t` is a list of byte buffers, each stored together with its byte lenght.
- `fmt_t` is a list of (ℓ, lbl) tuples, each containing a *bit* length ℓ together with an associated "label" lbl.
- `exp_res_t` is a boolean storing whether outputs (y, y') should match or not.

The case of (ℓ, lbl) tuples contained in `fmt_t` deserves a more detailed explanation. In particular, consider the case covered in bit-exclusion tests in [12]. Suppose we are testing a hash function implementation that should be defined for bit strings with length $\ell \neq 0 \bmod 8$. Say we have an input string x of bit length $\ell = 14$. Then x will be stored in a `in_t` struct pointing to a byte buffer of bytelength 2, where the last two bits are unused.

If we were testing bit-contribution and bit-exclusion independently, the first test would flip the first 14 bits and check that resulting hashes differ, while the


$x =$
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
x_8 x_9 x_{10} x_{11} x_{12} x_{13}

second test would flip the last 2 bits and check that hashes do not differ. Our Maul function performs both tests as once by making use of the format specifier fmt. In particular, we would set fmt $\leftarrow$ [(14, `DIFF`), (2, `EQ`)], and store which bit should first be flipped as part of state σ. Say, $\sigma \leftarrow 0$, where we address bits from 0 to 15. Given the initial input `in_t` x, Maul would flip x's σ^{th} bit, and set the expected result value based on whether to expect the flipped bit to cause different hashes ($\sigma < 14$) or the same hashes ($\sigma \geq 14$). In practice, whether this condition is satisfied or not (and hence wether we detected a bug or not) will be checked by Match. Utility functions for computing labels and data lengths from format strings are given in Figure 2.

In order to detect memory over-reading, we tend to define format strings that include a one-byte buffer before and after the main buffer to be used by the test program. For example, in the case of the above example, we may pass fmt $\leftarrow$ [(8, `EQ`), (14, `DIFF`), (2, `EQ`), (8, `EQ`)], and have the test program specifically address the input and output buffers starting from the second byte, located at "&x[1]", rather than from the first, located at "&x".

```
BufBitlen(fmt)
bitlen ← 0
foreach $(\ell, \text{lbl}) \in$ fmt
    bitlen ← bitlen + $\ell$
return bitlen

BufBytelen(fmt)
len ← $\lfloor(\text{BufBitlen(fmt)} + 7)/8\rfloor$
return len

GetLabel($i$, fmt)
cnt ← −1
idx ← −1
while cnt < $i$
    idx ← idx + 1
    $(\ell, \text{lbl})$ ← fmt[idx]
    cnt ← cnt + $\ell$
return lbl
```

Figure 2: Format parsing utilities.

4.1 Instantiating specifications

We now look at how to concretely instantiate a metamorphic test specification. We start by looking at hash functions, which present a simpler case. We then follow by outlining the tests we perform on KEMs and DSSs.

Hash Functions. The security properties often expected from cryptographic hash functions are preimage, second preimage, and collision resistance. Many protocols model them as random oracles, meaning that there is commonly also the expectation that the output of the hash function would look uniformly distributed.

The Test function defined in Figure 1 essentially generates pairs $(x, y \leftarrow H(x))$ and $(x', y' \leftarrow H(x'))$, and proceeds to check for validity of these pairs using a Match function. While it is unclear how to test for for preimage and collision resistance using this framework, second preimage resistance is to some extent testable by having Match implement an equality check between y and y', as essentially previously done in [12]. This results in Test 1.

Near-second preimages and uniform outputs may also be testable, by using some statistical test on y and y' inside Match, however we leave this for future work.

```
Test: $\mathsf{Hash}(\mathsf{Maul}(x))$
Format: $[(8, \texttt{EQ}), (\ell, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $x \leftarrow (1^8)_2 || (0^\ell)_2 || (1^8)_2$
    $\mathrm{aux} \leftarrow \bot$
    return $x, \mathsf{Call}(), \mathrm{aux}$
Call
    $y \leftarrow \mathsf{Hash}(\&x[1])$
    return $y$
```

Test 1: Testing Hash, mauling the input x.

```
Test: $\mathsf{Gen}(; \mathsf{Maul}(r))$
Format: $[(8, \texttt{DIFF})]$
GenInput
    $x \leftarrow (0^8)_2$
    $\mathrm{aux} \leftarrow \bot$
    return $x, \mathsf{Call}(), \mathrm{aux}$
Call
    $(\mathsf{sk}, \mathsf{pk}, rv) \leftarrow \mathsf{Gen}(; x)$
    $y \leftarrow (\mathsf{sk}||\mathsf{pk}, rv)$
    return $y$
```

Test 2: Testing Gen, mauling random tape r.

KEMs. Due to the richer API compared to hash functions, more tests can be designed for KEMs. We will be trying to capture bugs in the three functions Gen, Encaps and Decaps, by testing the behaviour of the output when modifying each individual input.

As Gen and Encaps take in input a random tape, our tests will have to deal with it. We consider two scenarios. In the first scenario, we are mauling an input that is not the random tape, say pk in Encaps. In this case, we fix a seed for the random number generator (RNG). In the second scenario, we are mauling specifically the random tape. Since mauling the seed to the RNG would likely not have any effect other than testing on different randomness, we instead leverage the flexibility of the LibOQS interface to RNGs, and define a bad RNG, that outputs a fixed byte, given in input. While no security should be expected by a KEM (or DSS) without access to good randomness, we notice however that some implementations hang in this scenario.

We note that, while the cryptographic interface of KEM functions does not mention a "return value", these functions do technically return an integer value as C functions as implemented in LibOQS. We include this in the output of each function, and check it as part of Match.

Testing Gen. The key generation algorithm Gen takes as input a random tape and outputs a secret-public key pair (sk, pk). Here we only test the function under bad randomness, Test 2, looking for collisions in output. Some implementations hang on bad random tapes; this is likely due to key generation trying to sample random elements with some given mathematical property, and failing to find one.

Testing Encaps. The encapsulation function Encaps takes in input a public key pk and a random tape, and outputs a shared secret ss and an encapsulation c of ss. We run two experiments where pk is mauled. In the first one, Test 3, we test whether the shared secret generated by Encaps on a valid public key pk and on a mauled version pk′ differ. In the second experiment, Test 4, we test whether the shared secret generated by Encaps on a mauled public key still decapsulates correctly. In the experiment where we maul the random tape, Test 5, we check that outputs of Encaps change as a result.

```
Test: $\mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r)$
Format: $[(8, \texttt{EQ}), (|\mathsf{pk}|, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
  $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
  $(\mathsf{pk}, \mathsf{sk}, rv) \leftarrow \mathsf{Gen}(; r)$
  $x \leftarrow (1^8)_2 || \mathsf{pk} || (1^8)_2$
  $\mathrm{aux} \leftarrow \mathsf{sk}$
  $\textbf{return } x, \mathsf{Call}(), \mathrm{aux}$
Call
  $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"call"})$
  $(ss, c, rv) \leftarrow \mathsf{Encaps}(\&x[1]; r)$
  $y \leftarrow (ss || c, rv)$
  $\textbf{return } y$
```

Test 3: Testing Encaps, mauling public key pk.

```
Test: $\mathsf{Decaps}(\mathsf{sk}, \mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r))$
Format: $[(8, \texttt{EQ}), (|\mathsf{pk}|, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
  $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
  $(\mathsf{pk}, \mathsf{sk}, rv) \leftarrow \mathsf{Gen}(; r)$
  $x \leftarrow (1^8)_2 || \mathsf{pk} || (1^8)_2$
  $\mathrm{aux} \leftarrow \mathsf{sk}$
  $\textbf{return } x, \mathsf{Call}(), \mathrm{aux}$
Call
  $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"call"})$
  $(ss_e, c) \leftarrow \mathsf{Encaps}(\&x[1]; r)$
  $(ss_f, rv) \leftarrow \mathsf{Decaps}(c, \mathrm{aux})$
  $eq \leftarrow [\![ ss_e = ss_f ]\!]$
  $y \leftarrow (eq, rv)$
  $\textbf{return } y$
```

Test 4: Testing Encaps, mauling public key pk, then decapsulating.

Testing Decaps. The decapsulation function Decaps produces a shared secret ss from a secret key sk and encapsulation c. In this case no random tape is

```
Test: $\mathsf{Encaps}(\mathsf{pk}; \mathsf{Maul}(r))$
Format: $[(8, \mathtt{DIFF})]$
GenInput
    $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $(\mathsf{pk}, \mathsf{sk}, rv) \leftarrow \mathsf{Gen}(; r)$
    $x \leftarrow (0^8)_2$
    $\mathrm{aux} \leftarrow \mathsf{pk}$
    $\textbf{return } x, \mathsf{Call}(), \mathrm{aux}$
Call
    $(ss, c, rv) \leftarrow \mathsf{Encaps}(\mathrm{aux}; x)$
    $y \leftarrow (ss || c, rv)$
    $\textbf{return } y$
```

Test 5: Testing Encaps, mauling random tape r.

present, and the mauling is simply performed over the secret key sk, Test 6, or the ciphertext c, Test 7, with Match checking whether the resulting shared secrets differ from the original. In particular, failing to pass the test mauling c highlights a failure to provide IND-CCA security (Definition 5).

DSSs. The API of digital signatures is similarly rich to that of KEMs, hence we follow a similar approach in writing and running tests. While we test the functionality of Verify, we note that in the older versions of LibOQS the interface did use instead a "Open" function (`crypto_sign_open(...)`) required by NIST. We have decided to omit also testing Open, and following to the UF-CMA syntax instead.

Testing Gen. This is done identically to what is done for KEMs.

Testing Sign. The Sign function maps a secret key sk, message m, and random tape r, to a signature σ of length ℓ_σ (and a return value, in the case of LibOQS). Not all signature schemes tested have constant-size signatures. To address these cases, LibOQS provides an upper bound on the signature length, and outputs a buffer containing the signature σ and an integer containing the signature length ℓ_σ. We consider both σ and ℓ_σ as part of the signature (as both are output by Sign and are required by Verify), and store and check them accordingly.

We expect the output of our tests in Tests 8 to 10 to result in differing signatures. We however notice that the Sign algorithm must not necessarily be a deterministic one. Therefore, we expect collisions for deterministic schemes whenever Sign is being tested using differing random tapes.

```
Test: $\mathsf{Decaps}(\mathsf{Maul}(\mathsf{sk}), c)$
Format: $[(8, \texttt{EQ}), (|\mathsf{sk}|, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $(s, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(; r)$
    $(\_, r') \leftarrow \mathsf{PRG}(s)$
    $(ss_e, c) \leftarrow \mathsf{Encaps}(\mathsf{pk}; r')$
    $x \leftarrow (1^8)_2||\mathsf{sk}||(1^8)_2$
    $\mathrm{aux} \leftarrow (\mathsf{pk}, c)$
    $\textbf{return } x, \mathsf{Call}(), \mathrm{aux}$
Call
    $(ss_f, rv) \leftarrow \mathsf{Decaps}(\&x[1], \mathrm{aux})$
    $y \leftarrow (ss_f, rv)$
    $\textbf{return } y$
```

Test 6: Testing Decaps, mauling secret key sk.

```
Test: $\mathsf{Decaps}(\mathsf{sk}, \mathsf{Maul}(c))$
Format: $[(8, \texttt{EQ}), (|c|, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $(s, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(; r)$
    $(\_, r') \leftarrow \mathsf{PRG}(s)$
    $(ss_e, c) \leftarrow \mathsf{Encaps}(\mathsf{pk}; r')$
    $x \leftarrow (1^8)_2||c||(1^8)_2$
    $\mathrm{aux} \leftarrow (\mathsf{pk}, \mathsf{sk})$
    $\textbf{return } x, \mathsf{Call}(), \mathrm{aux}$
Call
    $(ss_f, rv) \leftarrow \mathsf{Decaps}(\mathrm{aux}, \&x[1])$
    $y \leftarrow (ss_f, rv)$
    $\textbf{return } y$
```

Test 7: Testing Decaps, mauling ciphertext c.

```
Test: $\mathsf{Sign}(\mathsf{Maul}(\mathsf{sk}), m; r)$
Format: $[(8, \texttt{EQ}), (|\mathsf{sk}|, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $(\mathsf{pk}, \mathsf{sk}, rv) \leftarrow \mathsf{Gen}(; r)$
    $m \leftarrow (0^{256})_2$
    $\mathrm{aux} \leftarrow (\mathsf{pk}, m, \perp)$
    $x \leftarrow (1^8)_2||\mathsf{sk}||(1^8)_2$
    $\textbf{return } x, \mathsf{Call}(), \mathrm{aux}$
Call
    $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"call"})$
    $(\ell_\sigma, \sigma, rv) \leftarrow \mathsf{Sign}(\&x[1], \mathrm{aux}.m; r)$
    $y = (\ell_\sigma||\sigma, rv)$
    $\textbf{return } y$
```

Test 8: Testing Sign, mauling secret key sk.

```
Test: $\mathsf{Sign}(\mathsf{sk}, \mathsf{Maul}(m); r)$
Format: $[(8, \texttt{EQ}), (256, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $(\mathsf{pk}, \mathsf{sk}, rv) \leftarrow \mathsf{Gen}(; r)$
    $\mathrm{aux} \leftarrow (\mathsf{pk}, \mathsf{sk})$
    $m \leftarrow (0^{256})_2$
    $x \leftarrow (1^8)_2||m||(1^8)_2$
    $\textbf{return } x, \mathsf{Call}(), \mathrm{aux}$
Call
    $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"call"})$
    $(\ell_\sigma, \sigma, rv) \leftarrow \mathsf{Sign}(\mathrm{aux}.\mathsf{sk}, \&x[1]; r)$
    $y = (\ell_\sigma||\sigma, rv)$
    $\textbf{return } y$
```

Test 9: Testing Sign, mauling message m.

```
Test: $\mathsf{Sign}(\mathsf{sk}, m; \mathsf{Maul}(r))$
Format: $[(8, \texttt{DIFF})]$
GenInput
    $(\_, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $(\mathsf{pk}, \mathsf{sk}, rv) \leftarrow \mathsf{Gen}(; r)$
    $x = \$ \leftarrow (0^8)_2$
    $m \leftarrow \mathsf{00}$ // 1 byte
    $\mathrm{aux} \leftarrow (\mathsf{sk}, m)$
    return $x, \mathsf{Call}(), \mathrm{aux}$
Call
    $(\ell_\sigma, \sigma, rv) \leftarrow \mathsf{Sign}(\mathrm{aux}.\mathsf{sk}, \mathrm{aux}.m; x)$
    $y = (\ell_\sigma || \sigma, rv)$
    return $y$
```

Test 10: Testing Sign, mauling random tape r.

```
Test: $\mathsf{Verify}(\mathsf{Maul}(\mathsf{pk}), m, \sigma)$
Format: $[(8, \texttt{EQ}), (|\mathsf{pk}|, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $(s, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $m \leftarrow \mathsf{42}^{32}$ // 32 bytes
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(r)$
    $(\_, r') \leftarrow \mathsf{PRG}(s)$
    $(\ell_\sigma, \sigma, rv) \leftarrow \mathsf{Sign}(\mathsf{sk}, m; r')$
    $x \leftarrow (1^8)_2 || \mathsf{pk} || (1^8)_2$
    $\mathrm{aux} \leftarrow (\mathsf{sk}, m, \sigma, \ell_\sigma)$
    return $x, \mathsf{Call}(), \mathrm{aux}$
Call
    $rv \leftarrow \mathsf{Verify}(\&x[1], \mathrm{aux}.m, \mathrm{aux}.\sigma)$
    $y \leftarrow rv$
    return $y$
```

Test 11: Testing Verify, mauling public key pk.

```
Test: $\mathsf{Verify}(\mathsf{pk}, \mathsf{Maul}(m), \sigma)$
Format: $[(8, \texttt{EQ}), (256, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $(s, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $m \leftarrow (0^{256})_2$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(r)$
    $(\_, r') \leftarrow \mathsf{PRG}(s)$
    $(\ell_\sigma, \sigma, rv) \leftarrow \mathsf{Sign}(\mathsf{sk}, m; r')$
    $x \leftarrow (1^8)_2 || m || (1^8)_2$
    $\mathrm{aux} \leftarrow (\mathsf{pk}, \mathsf{sk}, \sigma, \ell_\sigma)$
    return $x, \mathsf{Call}(), \mathrm{aux}$
Call
    $rv \leftarrow \mathsf{Verify}(\mathrm{aux}.\mathsf{pk}, \&x[1], \mathrm{aux}.\sigma)$
    $y \leftarrow rv$
    return $y$
```

Test 12: Testing Verify, mauling message m.

```
Test: $\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Maul}(\sigma))$
Format: $[(|\ell_\sigma|, \texttt{DIFF}), (8, \texttt{EQ}), (|\sigma|, \texttt{DIFF}), (8, \texttt{EQ})]$
GenInput
    $(s, r) \leftarrow \mathsf{PRG}(\texttt{"geninput"})$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(; r)$
    $m \leftarrow \mathsf{42}^{32}$ // 32 bytes
    $(\_, r') \leftarrow \mathsf{PRG}(s)$
    $(\ell_\sigma, \sigma, rv) \leftarrow \mathsf{Sign}(\mathsf{sk}, m; r')$
    $\mathrm{aux} \leftarrow (\mathsf{pk}, \mathsf{sk}, m)$
    $x \leftarrow (\ell_\sigma || (1^8)_2 || \sigma || (1^8)_2)$
    return $x, \mathsf{Call}(), \mathrm{aux}$
Call
    $rv \leftarrow \mathsf{Verify}(\mathrm{aux}.\mathsf{pk}, \mathrm{aux}.m, \&x[\lfloor \frac{\ell_\sigma + 7}{8} \rfloor + 1])$
    $y \leftarrow rv$
    return $y$
```

Test 13: Testing Verify, mauling signature σ.

Testing Verify. The Verify function maps a public key pk, message m and signature (and signature length) (σ, ℓ_σ), into a return value indicating success or failure of verification. Checks are performed on the output return value rv accepting or rejecting the signature. Tests check whether mauling of the inputs results in a similarly valid signature to the original, see Tests 11 to 13. In the specific case where the signature is being mauled, Test 13, the new signature passing verification implies a lack of strong unforgeability (Definition 6) of the implementation.

5 Our Experiments

5.1 Experimental setup

Testing loop. The starting point of our implementation is the AFL++ fuzzing framework[9] [5]. The fuzzing framework is responsible for handling crashes of the program under observation and collecting information required to reproduce such crashes. While AFL++ already provided most of the functionality that was needed out-of-the-box, there were a few modifications that were required from our part to make the framework more closely match our requirements. First, the default AFL++ has a hard 1 MB limit on the size of input files, which we had to increase to 10 MB to make room for storing large public keys as part of the crashing inputs. Second, the default deterministic mutator triggers the same bug multiple times due to trying many redundant Maul combinations such as flipping every bit and every pair of neighboring bits. We write a custom mutator to only flip single bits for the sake of runtime. Third, some schemes produce too many crashing examples. Since each recorded crash may contain megabytes of auxiliary information (such as large public keys), we limit AFL++ to only collect up to 10 unique crashes (down from the default 10^4), to avoid storing terabytes of data.

Cryptographic implementations. The cryptographic implementations that we chose to test were collected from from the Open Quantum Safe project (LibOQS) [16] and the SUPERCOP project [1]. For LibOQS, we tested the 0.8.0 release, the 0.4.0 release, and the November 2018 `nist-branch` snapshot. These three snapshots include the majority of all reference implementations submitted to NIST as part of their post-quantum standardisation process [14]. For hash functions, we tested the 20240107 release of SUPERCOP.

5.2 Results

During our testing, we identify many instances of implementations deviating from our expected results. Before delving into details, we clarify how we count the number of instances of an unexpected result. Say we have a scheme "XYZ" that presents two different parameterisations: "XYZ-128" and "XYZ-256", and that both parameter sets have a reference implementation in different versions of LibOQS. We count test results in different versions of LibOQS independently.

[9] https://github.com/AFLplusplus/AFLplusplus

We also count test results in different parametrisations independently. However, we do not count multiple test results in a specific parametrisation and LibOQS version multiple times. For example, if the implementations of “XYZ-128/256” in LibOQS 0.4.0 and 0.8.0 share a same line of code that triggers a bug in both version of LibOQS using two different inputs, we would count this 4 times: once per parameter set and per LibOQS version. We use this approach since we count total numbers automatically, instead of manually inspecting the reason for each specific unexpected result.

In our testing, we identify 265 malleabilities, where changing one function input does not result in a change in function output. We report in Table 2 the total number for each test. While not all malleabilities imply a security or software bug, some do. We also detect 38 hangs, 50 segmentation faults, 3 heap overflows, and 2 stack overflows caused by mauling one of the intended inputs. These could be seen as software vulnerabilities, depending on how exposed the API for the scheme is. We proceed to mention some examples below. We provide a full list of results and code for reproducing the tests at `https://gitlab.com/fvirdia/crypto-fun-test`.

Table 1: Wall time required to run our tests. The machine used has two Intel(R) Xeon(R) Gold 6138 CPUs at 2.00GHz with 20 cores each and 376GiB of RAM.

Library	Used cores	Wall time
SUPERCOP `20240107`	31	$6h\ 58m$
LibOQS `nist-branch` 11-2018	31	$8h\ 59m$
LibOQS 0.4.0	31	$12h\ 43m$
LibOQS 0.8.0	31	$14h\ 23m$

Software bugs. An immediately useful category of results identified by our test are software bugs, by which we mean segmentation faults, stack and heap overflows, memory over-reads.

In particular, we find a memory over-read vulnerability in the optimised implementation of the KNOT-384 [19] hash function submitted to the first round of the NIST lightweight cryptography standardisation process [13], which has since been fixed by the authors albeit not disclosed by them. This bug would result in messages with a multiple-of-6 bytelength to be hashed to different values depending on the first byte following the message in memory. The issue was caused by incorrect masking in the conversion between least-significant-bit-first and most-significant-bit-first integer notation. This bug was fixed by the authors in a later round submission without disclosure of the bug. The version in SUPERCOP was not updated.

We also include hangs in this classification, although often this is plausibly not the result of a software bug. Indeed, they may be expected behaviour. For example, if key generation requires using the randomness source to sample a

Table 2: Summary of the kind of malleabilities and crashes observed during our tests.

Test	Bit contribution failure		Bit exclusion failure
	Malleabilities	Crashes/hangs	Non-malleabilities
SUPERCOP 20240107			
$\mathsf{Hash}(\mathsf{Maul}(m))$, Test 1	3	0	3
LibOQS 0.8.0			
$\mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r)$, Test 3	10	0	0
$\mathsf{Decaps}(\mathsf{sk}, \mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r))$, Test 4	10	0	0
$\mathsf{Encaps}(\mathsf{pk}; \mathsf{Maul}(r))$, Test 5	1	10 (hang)	0
$\mathsf{Decaps}(\mathsf{Maul}(\mathsf{sk}), c)$, Test 6	23	0	0
$\mathsf{Sign}(\mathsf{Maul}(\mathsf{sk}), m; r)$, Test 8	3	0	0
$\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Maul}(\sigma))$, Test 13	2	0	0
LibOQS 0.4.0			
$\mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r)$, Test 3	23	8 (segfault)	0
$\mathsf{Decaps}(\mathsf{sk}, \mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r))$, Test 4	31	8 (segfault)	0
$\mathsf{Encaps}(\mathsf{pk}; \mathsf{Maul}(r))$, Test 5	1	10 (hang)	0
$\mathsf{Decaps}(\mathsf{Maul}(\mathsf{sk}), c)$, Test 6	51	0	0
$\mathsf{Decaps}(\mathsf{sk}, \mathsf{Maul}(c))$, Test 7	10	0	0
$\mathsf{Sign}(\mathsf{Maul}(\mathsf{sk}), m; r)$, Test 8	10	3 (heap buffer overflow) + 1 (hang)	0
$\mathsf{Sign}(\mathsf{sk}, \mathsf{Maul}(m); r)$, Test 9	0	1 (hang)	0
$\mathsf{Sign}(\mathsf{sk}, m; \mathsf{Maul}(r))$, Test 10	22	1 (hang)	0
$\mathsf{Verify}(\mathsf{Maul}(\mathsf{pk}), m, \sigma)$, Test 11	7	1 (hang)	0
$\mathsf{Verify}(\mathsf{pk}, \mathsf{Maul}(m), \sigma)$, Test 12	0	1 (hang)	0
$\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Maul}(\sigma))$, Test 13	9	1 (hang)	0
LibOQS `nist-branch` 11-2018			
$\mathsf{KEM\ Gen}(; \mathsf{Maul}(r))$, Test 2	3	0	0
$\mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r)$, Test 3	4	6 (segfault) + 1 (returns $\perp$)	0
$\mathsf{Decaps}(\mathsf{sk}, \mathsf{Encaps}(\mathsf{Maul}(\mathsf{pk}); r))$, Test 4	9	7 (segfault) + 1 (hang)	0
$\mathsf{Decaps}(\mathsf{Maul}(\mathsf{sk}), c)$, Test 6	24	12 (segfault) + 7 (hang)	0
$\mathsf{Decaps}(\mathsf{sk}, \mathsf{Maul}(c))$, Test 7	3	9 (segfault) + 2 (stack overflow) + 4 (hang)	0
$\mathsf{Sign}(\mathsf{sk}, m; \mathsf{Maul}(r))$, Test 10	6	0	0

non-zero integer and we assume the source is tampered such that only zeroes are output, these may likely lead to key generation hanging. To avoid this kind of scenario, it may be advisable to not use directly a randomness source, and instead post-processing it with a pseudorandom generator.

Cryptographic weaknesses. Three of our tests identify implementations not satisfying well-established security notions.

First, we observe three hash function implementations failing to achieve second-preimage resistance (Definition 4): acehash256v1 (SSE2 implementation), syconhash256v1 (AVX implementation), syconhash256v1 (SSE implementation).

We further identify some KEM implementations not satisfying IND-CCA security (Definition 5). Some of these are purposely designed not to provide such security, such as the ephemeral ThreeBears and SIKE variants in LibOQS 0.4.0, and the BIKE variant in LibOQS `nist-branch` 11-2018. Some crash or hang on decapsulation of mauled ciphertext, such as Big Quake, LedaKEM and Lima in LibOQS `nist-branch` 11-2018, technically providing a a form of ciphertext-validity oracle. Finally, NTRU variants in LibOQS 0.4.0 (HRSS-701, HPS-2048-677, HPS-2048-509) perform decapsulation of the encapsulated shared secret, hence failing to provide IND-CCA security. This bug, caused by the implementation not checking zero-padding of non-byte-aligned encapsulations, was confirmed and fixed after private disclosure. [10]

In the case of signatures, we identify some schemes not providing strong unforgeability (Definition 6), such as various Picnic and Falcon instances in LibOQS 0.4.0, with Falcon still lacking strong unforgeability in LibOQS 0.8.0. Private communication with the Falcon team confirmed that this had been independently discovered and addressed in version 1.2 of the specification. Shortly after disclosure, the version of Falcon available via LibOQS was updated. [11] While strong UF-CMA is a standard security notion, the NIST competition did not require proposed schemes to satisfy it, and none of these schemes claims it. However, in the case of Falcon, the signature scheme is built using the GPV paradigm, which guarantees strong unforgeability [8, Prop 6.1]. As the Falcon team did not mention this not being the case in their design document, users could expect this property to hold by default. The loss of strong unforgeability was due to the encoding scheme being used for integer coefficients in their signatures, which did not guarantee unique encodings.

Potentially unexpected properties. Most of our tests stress malleabilities allowed by standard security notions. However, some of these may be counterintuitive at first thought. Among these, we highlight the malleability of keys.

Intuitively, one may think that a secret key should be uniquely identified to lead to correct decapsulation. However, This is not true for a large quantity of schemes. For example, this is the case for many schemes using the Fujisaki-Okamoto transform [6,7] to compile a passively secure encryption scheme into an

[10] `https://github.com/jschanck/ntru/commit/6bb52396fed494001228ca579f4c1d91ef558171`
[11] `https://github.com/open-quantum-safe/liboqs/issues/1315`

IND-CCA KEM. Implicit-rejection variants of the transform append a random string to the secret key from the passively secure scheme to be used to derive incorrect decapsulations whenever a ciphertext has been mauled [9]. This random string is not used if a ciphertext was not altered before decapsulation. Consequently, modifying this tag (and hence the secret key of the KEM) still allows correct decapsulation. This phenomenon is behind all implementations claiming IND-CCA security in LibOQS 0.8.0 and 0.4.0 that our tests detect as returning an unexpected result in Test 6.

We also observe some cases of malleability of DSS secret and public keys. Unlike for KEMs, malleability of secret keys does not seem to be caused by a specific transform, but rather due to having the same secret key format among randomised and de-randomised variants of the scheme. For example, Dilithium's secret key has the format $\mathsf{sk} = (\dots, K, \dots)$ where $K \in \{0,1\}^{256}$, where K is used in the deterministic variant of the scheme for generating message-dependent pseudorandomness within Sign. The randomised variant of the scheme ignores K and samples such signing randomness freshly. As in our tests we fix the seed of the random number generator (which provides this "fresh" signing randomness to the randomised variant), ignoring K results in different secret keys producing in the same signature. In practice, unlike for KEMs where decapsulation is inherently deterministic, this case would only be observable in deployment only if the randomness source for the signing process was faulty.

In the case of public keys, we notice that being able to maul a public key pk such that, given a (pk, m, σ) tuple that passes verification, $(\mathsf{Maul}(\mathsf{pk}), m, \sigma)$ also does, is reminiscent of breaks of conservative exclusive ownership (CEO) [15, Def. 1]. Unlike in the case of successful CEO adversaries, we are however not able to learn a valid secret key for $\mathsf{Maul}(\mathsf{pk})$, excluding the possibility of impersonation attacks [10].

6 Conclusions

Metamorphic testing had been previously proposed as a source of stress tests that could be useful when developing and evaluating hash function implementations [12]. In this work we argue that metamorphic testing approaches can be used to test implementations of more complex primitives, and demonstrate this by discovering or rediscovering various bugs in various implementations of KEMs and DSSs adopted by the LibOQS project at different points in time. Our implementation also suggests that this kind of tests could relatively easily be integrated with pre-existing libraries such as SUPERCOP and LibOQS due to the standard API used by these collections, allowing implementers to quickly be made aware of bugs and other unexpected properties their implementations may present.

Acknowledgements. We thank Greg Zaverucha for many useful discussions on early versions of this work. We also thank Martin R. Albrecht for providing servers for running our experiments.

References

1. D. J. Bernstein and T. Lange. (editors). eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to, accessed on 29 January 2024.
2. D. Bleichenbacher, T. Duong, E. Kasper, Q. Nguyen, and C. Lee. Project wycheproof. https://github.com/google/wycheproof.
3. T. Chen, T. Tse, and Z. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.
4. T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*, 2020.
5. A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.
6. E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, Aug. 1999.
7. E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, Jan. 2013.
8. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In R. E. Ladner and C. Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008.
9. D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Y. Kalai and L. Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, Nov. 2017.
10. D. Jackson, C. Cremers, K. Cohn-Gordon, and R. Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 2165–2180. ACM Press, Nov. 2019.
11. M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers. Improving software quality in cryptography standardization projects. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 19–30, 2022.
12. N. Mouha, M. S. Raunak, D. R. Kuhn, and R. Kacker. Finding bugs in cryptographic hash function implementations. *IEEE Transactions on Reliability*, 67(3):870–884, 2018.
13. National Institute of Standards and Technologies. Announcing request for nominations for lightweight cryptographic algorithms; notice. In *83 Federal Register 43656*, pages 43656—43657, August 2019. Available at https://www.federalregister.gov/d/2018-18433.
14. NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.
15. T. Pornin and J. P. Stern. Digital signatures do not guarantee exclusive ownership. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005. Proceedings 3*, pages 138–150. Springer, 2005.
16. D. Stebila and M. Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In R. Avanzi and H. Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 14–37, Cham, 2017. Springer International Publishing.

17. E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 11 1982.
18. J. Yang, S. Arya, and Y. Wang. Formal-guided fuzz testing: Targeting security assurance from specification to implementation for 5g and beyond. *arXiv preprint arXiv:2307.11247*, 2023.
19. W. Zhang, T. Ding, B. Yang, Z. Bao, Z. Xiang, F. Ji, and X. Zhao. KNOT: Algorithm specifications and supporting document, 2019. `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/knot-spec-round.pdf`.
20. Y. Zhou, F. Ma, Y. Chen, M. Ren, and Y. Jiang. Clfuzz: Vulnerability detection of cryptographic algorithm implementation via semantic-aware fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 33(2), dec 2023.
21. Z. Q. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, pages 346–351. Software Engineers Association Xian, China, 2004.