



Open Quantum Safe liboqs

Security Assessment

April 3, 2025

Prepared for:

Open Quantum Safe

Prepared by: **Filipe Casal, Scott Arciszewski, and Will Song**

Table of Contents

| | |
|--|-----------|
| Table of Contents | 1 |
| Project Summary | 2 |
| Executive Summary | 3 |
| Project Goals | 5 |
| Project Targets | 6 |
| Automated Testing | 7 |
| Summary of Findings | 8 |
| Detailed Findings | 9 |
| 1. Undefined arguments in CI workflow commands | 9 |
| 2. Missing OQS_OPENSSL_GUARD call | 11 |
| 3. Memset used to zero out memory instead of OQS_MEM_CLEANS | 13 |
| A. Vulnerability Categories | 14 |
| B. Code Quality Issues | 17 |
| C. Constant-Time Code Analysis | 18 |
| D. Automated Analysis Tool Configuration | 21 |
| E. Differential Fuzzing Implementations of Cryptographic Code | 23 |
| F. libFuzzer Differential Fuzzing Harness | 25 |
| G. Extending the Differential Fuzzing Setup | 30 |
| About Trail of Bits | 32 |
| Notices and Remarks | 33 |

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Will Song, Consultant
will.song@trailofbits.com

Scott Arciszewski, Consultant
scott.arciszewski@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

| Date | Event |
|-----------------|--|
| July 11, 2024 | Pre-project kickoff call |
| July 23, 2024 | Status update meeting #1 |
| August 20, 2024 | Delivery of report draft |
| August 20, 2024 | Report readout meeting |
| April 3, 2025 | Delivery of final comprehensive report |

Executive Summary

Engagement Overview

Trail of Bits reviewed the security of liboqs, an Open Quantum Safe library that provides an API for quantum-resistant key encapsulation mechanisms and signature algorithms.

A team of three consultants conducted the review from July 15 to August 20, 2024, for a total of five engineer-weeks of effort. Our testing efforts focused on reviewing the CI/CD pipeline, reviewing the wrapper code and utility functions developed by the liboqs maintainers, and exploring different testing techniques to be integrated into the codebase. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. We did not aim to review the implementation of each algorithm, as these implementations originate from third parties, such as the reference implementations.

Observations and Impact

During the review part of the engagement, we identified three minor issues related to the CI pipeline, missing error checks, and the risk that memory with secret data may not always be cleansed. During the review, we wrote custom CodeQL queries targeting the codebase, which we include in [appendix D](#). In addition to the code review, we explored the constant-time analysis testing used in the codebase and how to test different implementations using differential fuzzing.

For constant-time analysis, we explored using different tools to identify the Clangover vulnerability. We collected our findings in [appendix C](#).

To test different implementations of the same algorithm, we used fuzz testing to gather corpus files to use in cross-testing all binaries. Our methodology is described in [appendix E](#) and [appendix G](#).

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Open Quantum Safe take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Integrate additional compiler configurations for constant-time testing.** Adding a few other common compiler configurations to the testing environment can help detect potential constant-time issues (which could be introduced by the compiler).

- **Integrate a simple differential fuzzing campaign.** [Appendix E](#) describes a methodology for identifying discrepancies between different implementations of the same algorithm and how to circumvent corpus-related issues when fuzzing cryptographic code. We recommend integrating this approach into the testing infrastructure and extending it to all supported algorithms.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

| <i>Severity</i> | <i>Count</i> |
|-----------------|--------------|
| High | 0 |
| Medium | 0 |
| Low | 1 |
| Informational | 2 |
| Undetermined | 0 |

CATEGORY BREAKDOWN

| <i>Category</i> | <i>Count</i> |
|-----------------|--------------|
| Data Exposure | 1 |
| Data Validation | 1 |
| Testing | 1 |

Project Goals

The engagement was scoped to provide a security assessment of the Open Quantum Safe liboqs library. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the current CI/CD infrastructure mature and secure? Does the project integrate simple-to-use static analysis tools and techniques such as CodeQL, Semgrep, fuzzing, and constant-time analysis?
- Are the wrapper code and utility functions implemented by liboqs secure?
- How are algorithms tested for constant time in liboqs? Is the current approach robust? Can the current approach be easily extended to other tools?
- How does the project test different implementations of the same algorithm? Can differential fuzzing be used to compare reference implementations against an AVX2 implementation?

Project Targets

The engagement involved a review and testing of the following target.

liboqs

| | |
|------------|---|
| Repository | https://github.com/open-quantum-safe/liboqs |
| Version | 4cc88845e8bb66dca20cd61ff88c2162f38c3230 |
| Type | C, ASM |

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

| Tool | Description | Policy |
|----------------------------|---|---|
| Semgrep | An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time | All public Semgrep rules and Trail of Bits' private ruleset |
| actionlint | A static checker for GitHub Actions workflow files | Default configuration |
| poutine | A security scanner that detects misconfigurations and vulnerabilities in the build pipelines of a repository | Default configuration |
| CodeQL | A code analysis engine developed by GitHub to automate security checks | Appendix D |

Areas of Focus

Our automated testing and verification focused on the following:

- Identification of general code quality issues and unidiomatic code patterns
- Identification of issues in the CI deployment of the project

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

| ID | Title | Type | Severity |
|----|---|-----------------|---------------|
| 1 | Undefined arguments in CI workflow commands | Testing | Informational |
| 2 | Missing OQS_OPENSSL_GUARD call | Data Validation | Low |
| 3 | Memset used to zero out memory instead of OQS_MEM_CLEANSE | Data Exposure | Informational |

Detailed Findings

1. Undefined arguments in CI workflow commands

Severity: Informational

Difficulty: N/A

Type: Testing

Finding ID: TOB-OQS-1

Target: .github/workflows/{unix.yml, weekly.yml}

Description

Two CI workflow commands use parameters missing from the workflow job matrix. In the `weekly.yml` workflow, `SKIP_ALGS` is missing from the matrix (figure 1.1), while in the `unix.yml` workflow, `PYTEST_ARGS` is missing from the matrix (figure 1.2).

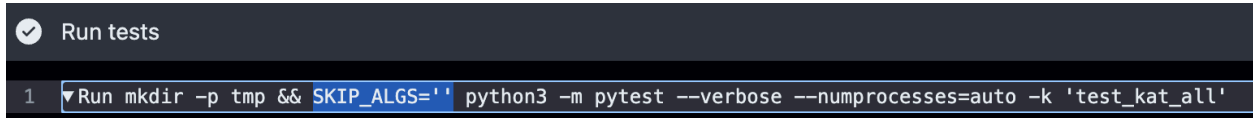
```
nistkat-x64:
  runs-on: ubuntu-latest
  strategy:
    fail-fast: false
  matrix:
    include:
      - name: generic
        container: openquantumsafe/ci-ubuntu-focal-x86_64:latest
        CMAKE_ARGS: -DOQS_DIST_BUILD=OFF -DOQS_OPT_TARGET=generic
        PYTEST_ARGS: --numprocesses=auto -k 'test_kat_all'
      - name: extensions
        container: openquantumsafe/ci-ubuntu-focal-x86_64:latest
        CMAKE_ARGS: -DOQS_DIST_BUILD=OFF -DOQS_OPT_TARGET=haswell
        PYTEST_ARGS: --numprocesses=auto -k 'test_kat_all'
  container:
    image: ${{ matrix.container }}
  steps:
    - name: Checkout code
      uses: actions/checkout@ee0669bd1cc54295c223e0bb666b733df41de1c5 # pin@v2
    - name: Configure
      run: mkdir build && cd build && cmake -GNinja ${{ matrix.CMAKE_ARGS }} .. &&
cmake -LA ..
    - name: Build
      run: ninja
      working-directory: build
    - name: Run tests
      timeout-minutes: 360
      run: mkdir -p tmp && SKIP_ALGS='${{ matrix.SKIP_ALGS }}' python3 -m pytest
--verbose ${{ matrix.PYTEST_ARGS }}
```

Figure 1.1: `.github/workflows/weekly.yml#41-67`

```
run: mkdir -p tmp && python3 -m pytest --verbose
--ignore=tests/test_code_conventions.py --ignore=tests/test_kat_all.py ${matrix.PYTEST_ARGS }
```

Figure 1.2: [.github/workflows/unix.yml#L273-L273](#)

At execution time, undefined parameters such as these will instantiate with an empty string when used in workflow commands, as can be seen in figure 1.3:



```
1 Run mkdir -p tmp && SKIP_ALGS='' python3 -m pytest --verbose --numprocesses=auto -k 'test_kat_all'
```

Figure 1.3: GitHub action execution of the `weekly.yml` workflow showing that the `SKIP_ALGS` parameter is an empty string

Recommendations

Short term, either define the parameters in the job matrix or remove their use from the command.

Long term, add tools such as [actionlint](#) and [poutine](#) to the CI pipeline to detect this and similar classes of issues.

2. Missing OQS_OPENSSL_GUARD call

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-OQS-2

Target: src/common/sha2/sha2_oss1.c, src/common/rand/rand_nist.c

Description

The `do_hash` function calls the `EVP_DigestInit_ex`, `EVP_DigestUpdate_ex`, and `EVP_DigestFinal_ex` functions without checking their return values. In principle, due to these missing checks, if one of these functions errors out and returns, a runtime error could occur.

```
static void do_hash(uint8_t *output, const uint8_t *input, size_t inplen, const
EVP_MD *md) {
    EVP_MD_CTX *mdctx;
    unsigned int outlen;
    mdctx = OSSL_FUNC(EVP_MD_CTX_new)();
    OQS_EXIT_IF_NULLPTR(mdctx, "OpenSSL");
    OSSL_FUNC(EVP_DigestInit_ex)(mdctx, md, NULL);
    OSSL_FUNC(EVP_DigestUpdate)(mdctx, input, inplen);
    OSSL_FUNC(EVP_DigestFinal_ex)(mdctx, output, &outlen);
    OSSL_FUNC(EVP_MD_CTX_free)(mdctx);
}
```

Figure 2.1: *src/common/sha2/sha2_oss1.c#16-25*

Figure 2.2 shows code that is not missing the error check but does not use the `OQS_OPENSSL_GUARD` macro; instead, it manually checks the return value.

```
/* Create and initialise the context */
if (!(ctx = OSSL_FUNC(EVP_CIPHER_CTX_new)())) {
    handleErrors();
}

if (1 != OSSL_FUNC(EVP_EncryptInit_ex)(ctx, oqs_aes_256_ecb(), NULL, key, NULL)) {
    handleErrors();
}

if (1 != OSSL_FUNC(EVP_EncryptUpdate)(ctx, buffer, &len, ctr, 16)) {
    handleErrors();
}
```

Figure 2.2: *src/common/rand/rand_nist.c#L59-L70*

We searched for variants of this issue using the following CodeQL query:

```

import cpp

from FunctionCall call, Function f
where
  f = call.getTarget() and
  // function name starts with EVP
  f.getName().matches("EVP%") and
  // and the function does not return a pointer or void
  not f.getType() instanceof PointerType and
  not f.getType() instanceof VoidType and
  // and the function is not guarded by the OQS_OPENSSL_GUARD macro
  not exists(MacroAccess m |
    m.getLocation().subsumes(call.getLocation()) and
    m.getMacroName() = "OQS_OPENSSL_GUARD"
  )
select call

```

Figure 2.3: Custom CodeQL query to identify variants of this finding

Exploit Scenario

An attacker triggers a call to the `do_hash` function. Due to the missing error checks, the call to `EVP_DigestFinal_ex` causes a runtime error, and the application crashes.

Recommendations

Short term, guard the calls to the functions in the `do_hash` function with the `OQS_OPENSSL_GUARD` macro. Consider replacing the manual check in the `rand_nist.c` file with a call to the `OQS_OPENSSL_GUARD` macro.

Long term, add custom CodeQL queries to the CI pipeline to prevent errors of this kind from being reintroduced into the codebase.

3. Memset used to zero out memory instead of OQS_MEM_CLEANSE

Severity: Informational

Difficulty: N/A

Type: Data Exposure

Finding ID: TOB-OQS-3

Target: src/common/rand/rand_nist.c,
src/sig_stfl/xmss/external/xmss_core_fast.c,
src/sig_stfl/lms/external/hss_sign.c

Description

The liboqs library provides the OQS_MEM_CLEANSE API, which enables memory to be zeroed out in a way that it is not removed by the compiler (e.g., by using the `memset_s` function if it is available during compilation).

However, there are still several explicit uses of `memset` throughout the codebase, potentially leaving room for the compiler to optimize that function call and to leave secret data exposed in memory. The following snippets show a few examples of those calls:

```
memset(DRBG_ctx.Key, 0x00, 32);  
memset(DRBG_ctx.V, 0x00, 16);  
AES256_CTR_DRBG_Update(seed_material, DRBG_ctx.Key, DRBG_ctx.V);  
DRBG_ctx.reseed_counter = 1;
```

Figure 3.1: *src/common/rand/rand_nist.c#L90-L93*

```
if (idx >= ((1ULL << params->full_height) - 1)) {  
    // Delete secret key here. We only do this in memory, production code  
    // has to make sure that this happens on disk.  
    memset(sk, 0xFF, params->index_bytes);  
    memset(sk + params->index_bytes, 0, (size_t)(params->sk_bytes -  
params->index_bytes));
```

Figure 3.2: *src/sig_stfl/xmss/external/xmss_core_fast.c#L952-L956*

```
if (idx >= ((1ULL << params->full_height) - 1)) {  
    // Delete secret key here. We only do this in memory, production code  
    // has to make sure that this happens on disk.  
    memset(sk, 0xFF, params->index_bytes);  
    memset(sk + params->index_bytes, 0, (size_t)(params->sk_bytes -  
params->index_bytes));
```

Figure 3.3: *src/sig_stfl/xmss/external/xmss_core_fast.c#L679-L683*

```
/* On failure, make sure that we don't return anything that might be */
```

```
/* misconstrued as a real signature */  
memset( signature, 0, signature_buf_len );
```

Figure 3.4: [src/sig_stfl/lms/external/hss_sign.c#L715-L717](#)

Recommendations

Short term, use the `OQS_MEM_CLEANSSE` function to guarantee that calls to `memset` are not removed by the compiler.

Long term, add CodeQL queries that check for explicit calls to `memset` in the codebase. Furthermore, consider adding binary validation for `memset` calls to ensure that the compiler does not remove the fallback implementation of `OQS_MEM_CLEANSSE`.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

| Vulnerability Categories | |
|--------------------------|---|
| Category | Description |
| Access Controls | Insufficient authorization or assessment of rights |
| Auditing and Logging | Insufficient auditing of actions or logging of problems |
| Authentication | Improper identification of users |
| Configuration | Misconfigured servers, devices, or software components |
| Cryptography | A breach of system confidentiality or integrity |
| Data Exposure | Exposure of sensitive information |
| Data Validation | Improper reliance on the structure or values of data |
| Denial of Service | A system failure with an availability impact |
| Error Reporting | Insecure or insufficient reporting of error conditions |
| Patching | Use of an outdated software package or library |
| Session Management | Improper identification of authenticated users |
| Testing | Insufficient test methodology or test coverage |
| Timing | Race conditions or other order-of-operations flaws |
| Undefined Behavior | Undefined behavior triggered within the system |

| Severity Levels | |
|-----------------|--|
| Severity | Description |
| Informational | The issue does not pose an immediate risk but is relevant to security best practices. |
| Undetermined | The extent of the risk was not determined during this engagement. |
| Low | The risk is small or is not one the client has indicated is important. |
| Medium | User information is at risk; exploitation could pose reputational, legal, or moderate financial risks. |
| High | The flaw could affect numerous users and have serious reputational, legal, or financial implications. |

| Difficulty Levels | |
|-------------------|---|
| Difficulty | Description |
| Undetermined | The difficulty of exploitation was not determined during this engagement. |
| Low | The flaw is well known; public tools for its exploitation exist or can be scripted. |
| Medium | An attacker must write an exploit or will need in-depth knowledge of the system. |
| High | An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue. |

B. Code Quality Issues

We identified the following code quality issues through manual and automated code review.

- **Stale documentation link:** The link in the code comment in figure B.1 points to a file that does not exist anymore. Add the commit hash so that the permalink will always correctly reference the intended state.

```
#if defined(OQS_DIST_X86_64_BUILD)
/* set_available_cpu_extensions_x86_64() has been written using:
 * https://github.com/google/cpu\_features/blob/master/src/cpuinfo\_x86.c
 */
#include "x86_64_helpers.h"
static void set_available_cpu_extensions(void) {
```

Figure B.1: *src/common/common.c#40–45*

- **Code comment on the wrong line of code:** The code comment highlighted in red in figure B.2 should document the line highlighted in yellow.

```
static void set_available_cpu_extensions(void) {
    /* mark that this function has been called */
    cpu_ext_data[OQS_CPU_EXT_ARM_AES] = 1;
    cpu_ext_data[OQS_CPU_EXT_ARM_SHA2] = 1;
    cpu_ext_data[OQS_CPU_EXT_ARM_SHA3] =
macos_feature_detection("hw.optional.armv8_2_sha3");
    cpu_ext_data[OQS_CPU_EXT_ARM_NEON] =
macos_feature_detection("hw.optional.neon");
    cpu_ext_data[OQS_CPU_EXT_INIT] = 1;
}
```

Figure B.2: *src/common/common.c#107–114*

C. Constant-Time Code Analysis

We set out to review the constant-time analysis tool used to test the algorithms used by the liboqs library, contribute potential improvements to it, and survey other existing tools that could complement the constant-time analysis. A [recent study](#) evaluated the usability and effectiveness of constant-time tools. The corresponding [list of constant-time tools](#) provides a valuable resource for constant-time research. We reviewed this work and chose a subset of tools to experiment with on the liboqs codebase.

Recently, two vulnerabilities affecting the Kyber reference implementation were uncovered by researchers:

- **KyberSlash**: A division operation, which is often a variable time instruction, takes secret data as input.
- **Clangover**: Certain compiler flags cause the emitted assembly to contain a branch depending on a secret value.

The constant-time tool evaluation included determining whether the tool is capable of detecting these two vulnerabilities.

Valgrind's Memcheck Tool

Currently, liboqs uses Valgrind's **Memcheck** tool to identify code branching and memory accesses that depend on secret data in the style of **ctgrind** and **TIMECOP**. The codebase has test harnesses for key-encapsulation mechanisms and signatures that check for the algorithms' correctness. Using these harnesses, and marking all secret data as uninitialized using Valgrind's `VALGRIND_MAKE_MEM_UNDEFINED` macro, Memcheck will report an error if it detects the use of uninitialized memory during the harness execution.

The Valgrind approach is well established and provides a suitable way to detect variable-time code that could enable key extraction, such as in the case of the Clangover vulnerability, described above. However, this approach will not detect instructions running in variable time, such as in the case of the KyberSlash vulnerability. The Valgrind approach is also not guaranteed to find all locations where the code branches on secret data, as it does not explore all code paths in the program. This limitation can be mitigated by running the program under inputs that explore all code paths or by running the program in a code-coverage-guided manner.

Clang's MemorySanitizer

An alternative to Memcheck is **Clang's MemorySanitizer** (MSan) to perform a similar constant-time code analysis. The approach is the same: MSan provides an API (`__msan_allocated_memory`) to mark memory as uninitialized, for which MSan will emit errors when this memory is used on branching instructions or memory accesses. Despite

its apparent advantages versus Memcheck (no Valgrind dependency, much lower execution overhead, support for architectures that Valgrind does not support), we were not able to use MSan to successfully detect the Clangover vulnerability. Additionally, given that the generated binaries are already instrumented, disassembling and analyzing the produced binary becomes a much more complex task.

This contrasts with the Valgrind approach, where the binary is translated at runtime to Valgrind's intermediate representation to be executed under its engine. This means that although Valgrind has an additional translation/execution layer above the compiled binary, it will run on the "real-world" binary, whereas in the MSan case, the analysis runs on a different and heavily instrumented binary (where different compilation configurations could even cause a variable-time piece of code to disappear).

dudect

The **dudect** tool takes a completely different approach to both Memcheck and MSan. Dudect measures the time it takes to run the binary for two classes of inputs and then uses statistical tests to determine if the two timing distributions are statistically different. For setup, Dudect requires writing a test harness that defines the two classes of inputs to test (e.g., a fixed input versus randomly generated inputs) and specifying the function to test. Given that the tool effectively measures the time it takes to run a function under multiple inputs, this approach can, in principle, detect all classes of variable-time code (that are executed). However, our experiments using dudect to detect the KyberSlash vulnerability were inconsistent: the same test successfully detected the timing difference on one machine, but we could not replicate this result on two other different machines. Since dudect measures the execution time of a given function, noise introduced by the microarchitecture, the operating system, or the load of the system executing the test can make it difficult to generalize or reproduce results.

Binsec

The **Binsec** constant-time analysis module performs symbolic analysis at the binary level and checks that no secret data affects the execution's control flow (i.e., executions will always execute the same path and perform the same memory accesses independently of the secret data). The tool does not detect variable-cycle instructions and would not be able to detect the KyberSlash vulnerability. In practice, it requires writing a configuration script that sets up the analysis engine and, for larger functions, takes a considerably larger amount of time to run when compared to the Memcheck approach.

Its advantage over Memcheck or MSan is that it explores all program paths, and not only those that are executed. However, unless the analysis is split into small functions, the cost of running the analysis with Binsec would become prohibitive when analyzing all the binaries compiled under a combination of compiler configurations.

haybale-pitchfork

The **haybale-pitchfork tool** is a symbolic execution tool that works on LLVM bitcode files for analysis and at the binary level. As with the other described tools, with the exception of **dudect**, it is capable only of detecting branching dependent on secret data and memory accesses from secret indices. Given that it works at the LLVM bitcode level, to analyze a binary, one needs to compile the source code to Clang-specific bitcode (*.bc files). This makes it unsuitable to analyze binaries generated by different compilers (although **translation engines** from machine code to LLVM bitcode exist).

To use **haybale-pitchfork**, a harness needs to be written in Rust that specifies the inputs to the function under test and whether these inputs should be considered public or secret with respect to the constant-time analysis.

D. Automated Analysis Tool Configuration

As part of this assessment, we used the tools described below to perform automated testing of the codebase.

D.1. Semgrep

We used the static analyzer **Semgrep** to search for weaknesses in the source code repository.

```
git clone https://github.com/0xdea/semgrep-rules

semgrep --metrics=off --sarif --config p/r2c-security-audit
semgrep --metrics=off --sarif --config p/trailofbits
semgrep --metrics=off --sarif --config semgrep-rules
```

Figure D.1: A subset of the used Semgrep rulesets

D.2. CodeQL

We analyzed the codebase with all C/C++ CodeQL rules, Trail of Bits' private CodeQL rules, and custom rules specifically targeting the codebase. Figure D.2 shows how we built the CodeQL database for the liboqs project. Note that running `ninja` from within the `build/` folder does not create a complete CodeQL database.

```
mkdir build
cmake -GNinja -B build/
codeql database create codeql.db --language=cpp --command="ninja -C build/"
--overwrite
```

Figure D.2: Commands used to build the CodeQL database

Finding uses of `OQS_STATUS` in a unary logical operation

This query checks for the use of the `OQS_STATUS` type in a unary logical operation. This is an antipattern of the codebase, as `OQS_STATUS` should always be used to compare with the `OQS_SUCCESS` or `OQS_ERROR` values.

```
/**
 * OQS_STATUS is used in a unary logical operation.
 *
 * @name oqs-status-used-in-unary-logical-operation
 * @kind problem
 * @problem.severity warning
 * @id cpp/oqs-status-used-in-unary-logical-operation
 */

import cpp
```

```

from FunctionCall call
where
  call.getTarget().getType().getName() = "OQS_STATUS" and
  exists(UnaryLogicalOperation op | op.getAChild() = call)
select call, "OQS_STATUS used in a unary logical operation"

```

Figure D.3: unary-operation-on-oqsstatus.q1

Finding unguarded uses of OpenSSL's EVP API

This query checks for the use of OpenSSL EVP functions that return an integer type and are not guarded by the OQS_OPENSSL_GUARD macro. Functions returning an integer usually indicate they are returning an error value, which should be checked.

```

/**
 * @name opensslcalls-guarded
 * @kind problem
 * @problem.severity warning
 * @id cpp/opensslcalls-guarded
 */

import cpp

from FunctionCall call, Function f
where
  f = call.getTarget() and
  // function name starts with EVP
  f.getName().matches("EVP%") and
  // and the function returns an integer type
  f.getType() instanceof IntegralType and
  // and the function is not guarded by the OQS_OPENSSL_GUARD macro
  not exists(MacroAccess m |
    m.getLocation().subsumes(call.getLocation()) and
    m.getMacroName() = "OQS_OPENSSL_GUARD"
  )
select call, "The call to " + f.getName() + " is not guarded by OQS_OPENSSL_GUARD"

```

Figure D.4: codeql-custom-queries-cpp/opensslcalls-guarded.q1#1-22

E. Differential Fuzzing Implementations of Cryptographic Code

The liboqs library contains implementations of key encapsulation mechanisms and digital signature algorithms. For each algorithm, the library can provide different implementations. For example, the ML-KEM-512 algorithm is implemented both in an architecture-agnostic way and with an AVX2-targeted implementation for x86-64.

This poses the question, Are these implementations equivalent? One way to try and answer this question is to perform differential fuzzing on the two implementations: provide the same input to two different programs and check the output (different output indicates a potential issue).

Besides detecting errors in different implementations of the same algorithm, differential fuzzing can also identify implementation errors in a single implementation. By differentially fuzzing the same implementation built for different targets, say 32-bit and 64-bit targets, issues related to `int` and `size_t` conversions can be identified.

Methodology

The approach is simple:

1. Write a deterministic fuzz harness that tests the functionality. We adapted the KEM tests in the library to be fed input entropy from the fuzzer.
2. Build the harness for different targets. We focused on comparing the non-AVX2 implementation with the AVX2 implementation.
3. Run separate fuzzing campaigns on each binary and gather different corpus sets, meaningful input values that provide comprehensive code coverage for each different target.
4. Run each target for each corpus (its own, and others) and compare the results to check whether all target binaries return the same value for the same corpus set.

One difficulty particular to cryptographic code is that most functions are straight-line programs, which will cause a generic code-coverage-guided fuzzer to fail to identify new corpus inputs. To gather coverage metrics, we added a function that branches on each bit of an input buffer and used this function on the generated secret key buffer. This causes the fuzzer to collect values every time it sees a new branching behavior in that function.

Results

We applied the described methodology using libFuzzer to test the ML-KEM-512 standard and AVX2 implementations, gathering corpus files until the rate of new corpus file addition

slowed down (which totaled 80 and 78 corpus files, respectively). Once gathered, we ran the binaries under test against the corpus files and hashed the resulting output for each binary–corpus pair. For each corpus set, the different targets resulted in the same output, so no issues with the implementations were identified.

Future Work

Given that we applied this technique only to the ML-KEM-512 AVX2 and non-AVX2 x86-64 implementations, the first step in continuing this work would be to apply the same methodology to the x86-32 implementations and to all other KEM and signature algorithms.

Given the difficulty of acquiring meaningful code-coverage inputs for cryptographic code, other approaches to obtaining corpus files should be studied and compared.

Lastly, this methodology could also be applied to test the other supported architectures, although this would require different machines for each architecture. Additionally, we believe that if any issues were identified when comparing an ARM with an x86 binary generated from the same code, they would more likely be compiler issues than issues in the cryptographic code itself.

F. libFuzzer Differential Fuzzing Harness

Figure F.1 shows the simple fuzzing harness we wrote for differentially fuzzing the ML-KEM-512 implementations. Figure F.2 shows the implementation of the compute function and the code-coverage-generating function, `branch_on_all_bits`.

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

void compute(char *buf, size_t buf_len);

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    compute((char *)data, size); // Invoke the PUT; in our case that is the compute
    // function
    return 0; // Return 0, which means that the test case was
    // processed correctly
}
```

Figure F.1: tests/harness.c

```
#include <assert.h>
#include <errno.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include <oqs/oqs.h>
#include <oqs/rand_nist.h>
#include <oqs/sha3.h>

#include "test_helpers.h"

#include "system_info.c"

// branch whether bit i == 1
void branch_on_all_bits(const uint8_t *data, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (data[i] & (1 << j))
            {
                puts("bit is 1");
            }
            else
            {
            }
        }
    }
}
```

```

        puts("bit is 0");
    }
}

static OQS_STATUS kem_kat(const char *method_name, uint8_t *fuzzer_input)
{
    bool all = true;
    uint8_t entropy_input[48];
    memcpy(entropy_input, fuzzer_input, 48);
    uint8_t seed[48];
    FILE *fh = NULL;
    OQS_KEM *kem = NULL;
    uint8_t *public_key = NULL;
    uint8_t *secret_key = NULL;
    uint8_t *ciphertext = NULL;
    uint8_t *shared_secret_e = NULL;
    uint8_t *shared_secret_d = NULL;
    OQS_STATUS rc, ret = OQS_ERROR;
    int rv;
    size_t max_count;
    OQS_KAT_PRNG *prng;

    prng = OQS_KAT_PRNG_new(method_name);
    if (prng == NULL)
    {
        goto err;
    }

    kem = OQS_KEM_new(method_name);
    if (kem == NULL)
    {
        printf("[kem_kat] %s was not enabled at compile-time.\n", method_name);
        goto algo_not_enabled;
    }

    OQS_KAT_PRNG_seed(prng, entropy_input, NULL);

    fh = stdout;

    public_key = malloc(kem->length_public_key);
    secret_key = malloc(kem->length_secret_key);
    ciphertext = malloc(kem->length_ciphertext);
    shared_secret_e = malloc(kem->length_shared_secret);
    shared_secret_d = malloc(kem->length_shared_secret);
    if ((public_key == NULL) || (secret_key == NULL) || (ciphertext == NULL) ||
        (shared_secret_e == NULL) || (shared_secret_d == NULL))
    {
        fprintf(stderr, "[kat_kem] %s ERROR: malloc failed!\n", method_name);
        goto err;
    }
}

```

```

max_count = all ? prng->max_kats : 1;

for (size_t count = 0; count < max_count; ++count)
{
    fprintf(fh, "count = %zu\n", count);
    OQS_randombytes(seed, 48);
    OQS_fprintBstr(fh, "seed = ", seed, 48);

    OQS_KAT_PRNG_save_state(prng);
    OQS_KAT_PRNG_seed(prng, seed, NULL);

    rc = OQS_KEM_keypair(kem, public_key, secret_key);
    if (rc != OQS_SUCCESS)
    {
        fprintf(stderr, "[kat_kem] %s ERROR: OQS_KEM_keypair failed!\n",
method_name);
        goto err;
    }
    OQS_fprintBstr(fh, "pk = ", public_key, kem->length_public_key);
    OQS_fprintBstr(fh, "sk = ", secret_key, kem->length_secret_key);

    branch_on_all_bits(secret_key, kem->length_secret_key);

    rc = OQS_KEM_encaps(kem, ciphertext, shared_secret_e, public_key);
    if (rc != OQS_SUCCESS)
    {
        fprintf(stderr, "[kat_kem] %s ERROR: OQS_KEM_encaps failed!\n",
method_name);
        goto err;
    }
    OQS_fprintBstr(fh, "ct = ", ciphertext, kem->length_ciphertext);
    OQS_fprintBstr(fh, "ss = ", shared_secret_e,
kem->length_shared_secret);

    // The NIST program generates KAT response files with a trailing
newline.
    if (count != max_count - 1)
    {
        fprintf(fh, "\n");
    }

    rc = OQS_KEM_decaps(kem, shared_secret_d, ciphertext, secret_key);
    if (rc != OQS_SUCCESS)
    {
        fprintf(stderr, "[kat_kem] %s ERROR: OQS_KEM_decaps failed!\n",
method_name);
        goto err;
    }

    rv = memcmp(shared_secret_e, shared_secret_d,
kem->length_shared_secret);
    if (rv != 0)

```

```

        {
            fprintf(stderr, "[kat_kem] %s ERROR: shared secrets are not
equal\n", method_name);
            OQS_print_hex_string("shared_secret_e", shared_secret_e,
kem->length_shared_secret);
            OQS_print_hex_string("shared_secret_d", shared_secret_d,
kem->length_shared_secret);
            goto err;
        }

        OQS_KAT_PRNG_restore_state(prng);
    }

    ret = OQS_SUCCESS;
    goto cleanup;

err:
    ret = OQS_ERROR;
    goto cleanup;

algo_not_enabled:
    ret = OQS_SUCCESS;

cleanup:
    if (kem != NULL)
    {
        OQS_MEM_secure_free(secret_key, kem->length_secret_key);
        OQS_MEM_secure_free(shared_secret_e, kem->length_shared_secret);
        OQS_MEM_secure_free(shared_secret_d, kem->length_shared_secret);
    }
    OQS_MEM_insecure_free(public_key);
    OQS_MEM_insecure_free(ciphertext);
    OQS_KEM_free(kem);
    OQS_KAT_PRNG_free(prng);
    return ret;
}

void compute(uint8_t *buf, size_t buf_len)
{
    if (buf_len != 48)
    {
        return;
    }
    OQS_init();

    OQS_STATUS rc = kem_kat("ML-KEM-512", buf);
    if (rc != OQS_SUCCESS)
    {
        puts("failed");
    }
    puts("succeeded");
}

```

Figure F.2: tests/test_kem_fuzz.c

Figure F.3 shows the cmake instructions to build the fuzzing harness.

```
add_executable(fuzz_kem test_kem_fuzz.c harness.c test_helpers.c)
target_link_libraries(fuzz_kem PRIVATE ${TEST_DEPS} -fsanitize=fuzzer)
target_compile_definitions(fuzz_kem PRIVATE NO_MAIN=1)
target_compile_options(fuzz_kem PRIVATE -g -O2 -fsanitize=fuzzer)
```

Figure F.3: tests/CMakeLists.txt#79–82

G. Extending the Differential Fuzzing Setup

We extended the proof-of-concept methodology described in [appendix E](#) for differentially fuzzing the two implementations of ML-KEM-512 in two ways:

- We extended the differential fuzzing campaign to test all KEMs supported by liboqs, comparing each algorithm's AVX2 and generic implementations. We also differentially compared the libjade versions of Kyber-512 and Kyber-768 against the generic implementation.
- We extended the fuzzing campaign to differentially fuzz the two implementations of all signature schemes supported by liboqs.

This work resulted in three scripts (one for the KEMs, one to compare the libjade Kyber implementations, and one for the signature schemes) that can be added to the CI pipeline to regularly test the equivalence of implementations of the same algorithm. The script will do the following for each supported algorithm:

1. Build the generic implementation of the algorithm and run a fuzzing campaign for a fixed amount of time, gathering corpus files in CORPUS_A.
2. Build the AVX2 (or libjade) implementation of the algorithm and run a fuzzing campaign for a fixed amount of time, gathering corpus files in CORPUS_B.
3. Run both binaries on the corpus files of CORPUS_A and CORPUS_B and verify that their results match.

In addition to these correctness checks, while the fuzzing campaigns run, libFuzzer will gather any slow-running or crash-causing input seeds in a separate folder so that these results can be later analyzed.

Results

The three differential fuzzing campaigns we ran did not identify differences in the output of two implementations of the same algorithm. This indicates that these implementations do indeed correctly implement the same algorithm as expected.

However, during these campaigns, we did find seeds that cause the key-generation algorithm of certain algorithms to take an extraordinary amount of time to terminate.

In particular, the fuzzer identified input seeds that, when provided to the KAT PRNG, will cause the key generation of KEM_classic_mceliece_348864, KEM_classic_mceliece_460896, KEM_classic_mceliece_6688128, KEM_classic_mceliece_6960119, and KEM_classic_mceliece_8192128 to take

upwards of 40 seconds on the generic implementation, versus 2 seconds on the AVX2 implementation.

Future Work

In this work, we completed differential fuzzing campaigns for all KEMs and signature schemes supported by liboqs, comparing their generic build on x86-64 and the AVX2-specific implementations.

Future extensions of this work follow the lines described in [appendix E](#):

- Differentially compare builds for the tier 1-supported architectures, starting the initiative by focusing on architectures where the `size_t` type is 32 bits wide.
- Experiment with different corpus-generation metrics.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at <https://github.com/trailofbits/publications>, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries and government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow [@trailofbits](#) on X and explore our public repositories at <https://github.com/trailofbits>. To engage us directly, visit our "Contact" page at <https://www.trailofbits.com/contact> or email us at info@trailofbits.com.

Trail of Bits, Inc.

228 Park Ave S #80688

New York, NY 10003

<https://www.trailofbits.com>

info@trailofbits.com

Notices and Remarks

Copyright and Distribution

© 2025 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

Trail of Bits considers this report public information; it is licensed to Open Quantum Safe under the terms of the project statement of work and has been made public at Open Quantum Safe's request. Material within this report may not be reproduced or distributed in part or in whole without Trail of Bits' express written permission.

The sole canonical source for Trail of Bits publications is the [Trail of Bits Publications page](#). Reports accessed through sources other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.