KAT-Seeded Fuzzing of Stateful Hash-Based Signature Verification in liboqs

Vishnu Ajith[*†], Muhammed Sihan Haroon[†], Muhammad Ibrahim[†]
[*]Research & Development Engineer, Quentangle, United Kingdom
[†]Lecturer in Computing, QA Higher Education, United Kingdom
vishnu@quentangle.com

***Abstract*—Stateful hash-based signature schemes such as XMSS and LMS are increasingly important in post-quantum cryptographic deployments, yet their verification paths remain difficult to fuzz effectively because key generation is substantially more expensive than verification. This paper presents a structured libFuzzer methodology for testing stateful hash-based signature verification in `liboqs`. The proposed approach uses pre-computed Known Answer Test (KAT) vectors to initialise valid public key, signature, and message buffers once during harness setup, enabling high-throughput mutation of verification inputs without repeated key generation.**

Using this methodology, we implemented and upstreamed two fuzz harnesses targeting XMSS/XMSSMT and LMS/HSS-LMS verification paths in `liboqs`. A ten-minute AddressSanitizer-enabled fuzzing campaign identified an OID-confusion heap-buffer-overflow in `xmssmt_core_sign_open`, assigned CVE-2026-46344. The flaw occurs when attacker-controlled public-key OID bytes derive a parameter set whose expected signature length exceeds the caller-allocated buffer, resulting in an out-of-bounds read during signature parsing. The issue was reproduced using a five-byte minimal crashing input and a standalone C reproducer, disclosed through the Open Quantum Safe coordinated disclosure process, and fixed in `liboqs` 0.16.0.

We further identify a mutation-distribution bias affecting short-input LMS fuzzing and resolve it using a round-robin field-rotation strategy. The resulting framework provides a reproducible and upstream-integrated methodology for adversarial testing of stateful post-quantum signature verification implementations.

***Index Terms*—post-quantum cryptography, stateful hash-based signatures, XMSS, LMS, libFuzzer, fuzz testing, heap-buffer-overflow, CVE-2026-46344, liboqs, NIST SP 800-208, coordinated disclosure, OID confusion**

I. Introduction

The transition to post-quantum cryptography has elevated stateful hash-based signatures (HBS) to a prominent deployment role. XMSS and LMS are the two families standardised by NIST SP 800-208 [1] and are actively used in Cisco IOS XR firmware signing [22], AWS KMS customer-managed keys, and the UK QAssure consortium's post-quantum migration tooling. Their security rests entirely on the collision-resistance of an underlying hash function - a conservative security assumption in post-quantum cryptography - which has driven their adoption in contexts where long-term cryptographic conservatism is prioritised.

CVE-2026-46344 (GHSA-2wxh-55qf-c7wg, CVSS 3.1 Moderate 5.3), distinct from CVE-2026-44518. Fuzz harnesses merged as PRs #2400 and #2412: https://github.com/open-quantum-safe/liboqs. No external funding.

Against this deployment backdrop, the verification paths of liboqs's XMSS and LMS implementations had received no structured adversarial coverage prior to this work. The existing test suite exercised correctness through Known Answer Test (KAT) vectors but, by construction, only with structurally valid inputs. Malformed OIDs, truncated signatures, and cross-parameter-set mismatches had never been probed. The following line, present in `xmss_commons.c` at the time of our investigation, illustrates the gap:

```
// Unused since smlen is a constant
(void) smlen;                /* xmss_commons.c:182 */
```

The comment is locally correct: the internal liboqs call path always derives `smlen` from `params.sig_bytes`, making it a constant from the inner function's perspective. It is not correct when an attacker controls `pk[0..3]`, because those bytes determine `params.sig_bytes` through a separate OID parse inside `xmss_sign_open` - a parse that occurs *after* the wrapper-layer length check introduced by the prior fix for CVE-2026-44518. When the OID-derived `sig_bytes` exceeds the allocated buffer, the discarded `smlen` is the only value that could prevent an out-of-bounds read.

A. Contributions

This paper makes four concrete, independently verifiable contributions:

1) **CVE-2026-46344:** A heap-buffer-overflow in the XMSS verification path of liboqs 0.15.0, confirmed with a five-byte minimal reproducer and a 40-line standalone C program, disclosed via GHSA-2wxh-55qf-c7wg, and fixed in liboqs 0.16.0 (Section VI).
2) **KAT-seeded harness design pattern:** A reusable methodology for fuzzing stateful HBS verify paths that eliminates the keygen cost barrier entirely (Section IV).
3) **Two upstream-merged fuzz harnesses:** Covering XMSS-SHA2_10_256, XMSSMT-SHA2_20/2_256, LMS-SHA256-H5-W2, and HSS LMS-SHA256-H5-W8-H5-W8, with 93 CI checks passed across all supported platforms (PRs #2400 [5] and #2412 [6], Sections IV–V).
4) **Round-robin field-mutation correction:** Characterisation and elimination of a field-mutation bias that emerges under short-input corpus conditions in sequential three-field mutation (Section IV).

II. Background

A. Stateful Hash-Based Signatures

Hash-based signature schemes derive their security from a single primitive: a collision-resistant hash function. Both XMSS [2] and LMS [3] construct a Merkle tree whose leaves are one-time signature (OTS) public keys: Winternitz OTS+ for XMSS, Leighton-Micali OTS for LMS. The tree root serves as the long-term public key. Each signing operation consumes a leaf; once consumed, that leaf must never be reused (*statefulness*). XMSSMT and HSS extend signing capacity by layering multiple trees.

Verification is stateless: given (pk, sm, m), the algorithm re-derives the authentication path and checks the result against the tree root. The *parameter set* governs all buffer dimensions, including `sig_bytes`, the expected signature length. This is the value that becomes attacker-controlled when `pk[0..3]` is tampered with.

B. liboqs Architecture

`liboqs` [4] provides a unified C API for post-quantum KEMs and signature schemes, referenced by oqs-provider (OpenSSL 3), OQS-BoringSSL, OQS-OpenSSH, and language bindings for C++, Go, Java, Python, and Rust. Algorithm dispatch for stateful signatures uses function pointers in an `OQS_SIG_STFL` struct; the public entry point is `OQS_SIG_STFL_verify`, which routes to a per-variant implementation compiled from `src/sig_stfl/xmss/` or `src/sig_stfl/lms/`.

C. libFuzzer and AddressSanitizer

libFuzzer [9] is an in-process, coverage-guided mutation engine distributed with LLVM. It calls `LLVMFuzzerTestOneInput` repeatedly, mutating inputs according to observed edge coverage. Paired with AddressSanitizer (ASan) [10], it detects heap-buffer-overflow, use-after-free, and related memory-safety violations with low false-positive rates. For coverage-guided fuzzing, higher execution throughput improves the rate of corpus exploration; therefore, reducing per-iteration cost is important when testing cryptographic verification routines [11].

III. Adversarial Model and Scope

We target memory-safety vulnerabilities in the verification paths of stateful HBS implementations in liboqs, reachable via the public `OQS_SIG_STFL_verify` API. The attacker is modelled as controlling one or more of (`pk`, `sm`, `m`) independently.

a) Reachability in practice: In firmware update chains, the vendor's public key is often bundled with signed metadata on writable storage: an attacker with filesystem write access can modify `pk` without touching `sm`. TUF root files similarly bundle public keys alongside signed targets. Certificate stores on constrained IoT devices store keys on flash that may be rewritten independently of signatures. In each scenario, the verifier receives an attacker-influenced `pk` with a correctly-sized but semantically mismatched `sm`.

TABLE I
Key Generation vs. Verification Cost

Parameter set	Keygen (ms)	Verify (μs)
XMSS-SHA2_10_256	∼800	∼1.2
XMSSMT-SHA2_20/2_256	∼2,400	∼2.8
LMS-SHA256-H5-W2	∼15	∼0.8
LMS-SHA256-H5-W8-H5-W8	∼45	∼1.5

Apple M-series

arm64, clang 22, `-O1`.

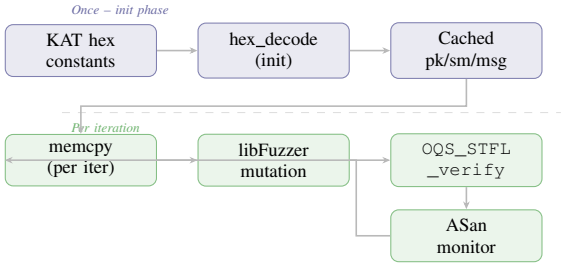

Fig. 1. KAT-seeded harness execution model. Keygen runs once during initialisation; per-iteration cost is a `memcpy`, mutation, and one call to `OQS_SIG_STFL_verify`.

b) Out of scope: Keygen and sign paths; timing and power side-channels; formal correctness proofs; cryptanalytic attacks on the underlying hash function.

IV. Harness Design

A. The Keygen Cost Problem

The central obstacle to fuzzing stateful HBS verification is the cost asymmetry between key generation and verification. Table I shows representative timings on the evaluation platform (Apple M-series arm64, clang 22, `-O1`). Naive per-iteration keygen would reduce execution throughput substantially, limiting practical corpus exploration during fuzzing.

B. KAT-Seeded Design Pattern

Our solution executes keygen exactly once during `LLVMFuzzerInitialize`, using pre-computed KAT (Known Answer Test) count-0 vectors embedded as hex-encoded string constants in the harness source. These are decoded once and cached for the lifetime of the session. Each iteration copies the cached buffers, applies libFuzzer's byte-level mutations, and calls `OQS_SIG_STFL_verify`. Fig. 1 illustrates the design.

a) Why KAT seeding outperforms random initialisation: Random or zero-filled inputs activate early-exit guards (invalid length, unknown OID, malformed index) and never reach the cryptographic core. KAT seeding provides a structurally valid baseline from which mutations explore the neighbourhood of legitimate inputs, reaching deeply nested code paths that random bytes cannot. The OID mutation that triggered CVE-2026-46344 exemplifies this: it started from a valid XMSS-SHA2_10_256 KAT vector and mutated only bytes 1–4 of the cached `pk` buffer.

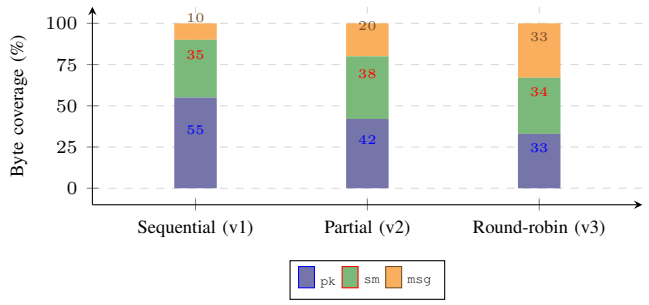

Fig. 2. Modelled field-byte coverage across three LMS harness design iterations under a short-input corpus. Sequential fill (v1) biases 55% toward `pk`. Round-robin (v3) achieves near-uniform coverage from the first iteration.

b) Throughput: On arm64 with ASan enabled we measure 1,667 execs/s for XMSS and 1,701 execs/s for LMS during ten-minute seeded fuzzing campaigns.

C. Input Layout

The fuzz input byte stream is interpreted as follows:

- **Byte 0:** Algorithm selector ($\mathrm{mod}\, N_{\mathrm{algs}}$), choosing which cached parameter-set buffers to use.
- **Byte 1 (LMS only):** Field-order selector (mod3; see below).
- **Remaining bytes:** Mutation payload applied to `pk`, `sm`, and `msg`.

A single binary covers multiple parameter sets; libFuzzer's corpus generation distributes inputs across selector values naturally.

D. Three-Field Mutation Rationale

We mutate all three verification inputs - `pk`, `sm`, and `m` - rather than the signature alone. This reflects the threat model of Section III: `pk` may be attacker-controlled independently of `sm`. Signature-only mutation would not have reached the OID mismatch path that produced CVE-2026-46344.

E. Round-Robin Field Rotation (LMS)

During upstream review, maintainer Michael Baentsch identified a coverage bias in sequential fill: when libFuzzer generates short inputs early in a session, sequential fill exhausts the mutation budget on `pk`, leaving `sm` and `m` at their KAT baseline. Fig. 2 quantifies the effect.

The fix is a three-case round-robin: each field occupies the first (most mutation-heavy) position in exactly one of the three cases, guaranteeing balanced access regardless of input length. Listing 1 shows the implementation.

```
size_t field_selector = fuzz_data[0] % 3;
uint8_t *fp[3]; size_t fl[3];
switch (field_selector) {
case 1: /* sm -> msg -> pk */
    fp[0]=mutated_sm;  fl[0]=sm_len;
    fp[1]=mutated_msg; fl[1]=msg_len;
    fp[2]=mutated_pk;  fl[2]=pk_len; break;
case 2: /* msg -> pk -> sm */
    fp[0]=mutated_msg; fl[0]=msg_len;
    fp[1]=mutated_pk;  fl[1]=pk_len;
    fp[2]=mutated_sm;  fl[2]=sm_len; break;
default: /* pk -> sm -> msg */
    fp[0]=mutated_pk;  fl[0]=pk_len;
    fp[1]=mutated_sm;  fl[1]=sm_len;
    fp[2]=mutated_msg; fl[2]=msg_len; break;
}
```

Listing 1. Round-robin field rotation in `fuzz_test_sig_stfl_lms.c`. A selector byte (mod 3) controls which field receives the most heavily mutated portion of the fuzz input.

V. Implementation and Upstream Integration

A. XMSS Harness (PR #2400)

`fuzz_test_sig_stfl_xmss.c` (269 lines) targets XMSS-SHA2_10_256 and XMSSMT-SHA2_20/2_256. KAT count-0 vectors are embedded as hex string constants (pk: 68 bytes; sm: 2,500 bytes XMSS / ∼5,605 bytes XMSSMT; msg: 33 bytes) and decoded during `LLVMFuzzerInitialize`.

A platform-specific issue arose during development on macOS: Homebrew LLVM compiled `LLVMFuzzerInitialize` as a hidden symbol, preventing libFuzzer from calling it at startup. The harness ran with its `initialised` flag permanently `false`, silently discarding all work. Detection required a diagnostic `fprintf` and a 5-second smoke run to confirm the `DIAG` line appeared. Annotating both entry points with `OQS_API` (`__attribute__((visibility("default")))`) resolved the issue; this fix will be submitted as a separate upstream PR.

B. LMS Harness (PR #2412)

`fuzz_test_sig_stfl_lms.c` (298 lines) targets LMS-SHA256-H5-W2 and HSS LMS-SHA256-H5-W8-H5-W8. Development required three commits: initial sequential fill, a partial rotation fix, and the correct round-robin design (Section IV-E). Maintainer Michael Baentsch merged the final version with the comment: *"Thanks very much @Vishnu2707 for the PR and the latest improvement! Good resolution for #2398 and good contribution."*

C. Build System Integration

Both harnesses are registered as CMake targets under the `OQS_BUILD_FUZZ_TESTS` guard with a compiler-identity check, consistent with the existing `fuzz_test_sig` harness. Enabling stateful algorithm compilation requires the `OQS_HAZARDOUS_EXPERIMENTAL_ENABLE_SIG_STFL_KEY_SIG` flag.

D. CI Coverage

Both PRs passed 93 CI checks spanning Linux (x86_64 and arm64, multiple configurations), macOS (14 and 15), Windows (x86 and arm64, 2022 and 2025 runners), Android (four ABI targets), iOS (OS64, TVOS), and Zephyr (v3.7.0, v4.3.0). Coverage reports showed a -0.004 pp delta, expected for new fuzz infrastructure not exercised by KAT tests.

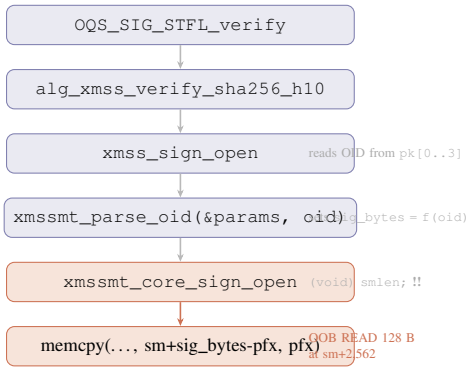

Fig. 3. Call chain from `OQS_SIG_STFL_verify` to the out-of-bounds read (shaded red). `pk[0..3]` drives `xmssmt_parse_oid` to set `sig_bytes` = 2,690 for XMSS-SHA2_16_256, while the caller's buffer holds only 2,500 bytes. `(void) smlen` at line 182 suppresses the only check that would catch the mismatch.

VI. Vulnerability: CVE-2026-46344

A. Discovery

After resolving the `LLVMFuzzerInitialize` visibility issue and seeding the corpus with KAT vectors, the XMSS harness was executed for ten minutes under ASan on liboqs at commit `2297d7a5`. It crashed during the seeded fuzzing campaign. The minimal crashing input is five bytes: `00 00 00 00 02`. Byte 0 selects XMSS-SHA2_10_256 and bytes 1–4 overwrite `pk[0..3]` - precisely the OID field.

B. Root Cause and Call Chain

Fig. 3 traces the execution path from the public API to the crash site.

The mechanism is as follows. (1) `xmss_sign_open` reads `pk[0..3]` and calls `xmss_parse_oid`, setting `params.sig_bytes` from the OID-identified parameter set. OID `0x00000002` resolves to XMSS-SHA2_16_256 (`full_height`=16), for which `sig_bytes` = 2,690. (2) The caller's `sm` was allocated for XMSS-SHA2_10_256 (`sig_bytes` = 2,500). (3) At `xmss_commons.c:182`, the caller-supplied `smlen` is discarded: `(void) smlen`. (4) The `memcpy` at line 194 computes offset $2{,}690 - 128 = 2{,}562$ into the 2,500-byte buffer, reading 128 bytes past the end. ASan reports the allocation boundary at 64 bytes after the region end due to alignment rounding.

a) Distinction from CVE-2026-44518: The prior fix (commit `ef70dea`) added a wrapper-layer check:

```
if (signature_len !=
    OQS_SIG_STFL_alg_xmss##v##_length_signature)
    return OQS_ERROR;
```

This rejects callers that pass `signature_len` shorter than the declared algorithm's compile-time constant. It does not validate that `pk[0..3]` encodes the *same* parameter set. When the caller passes a correctly-sized buffer (2,500 bytes, matching XMSS-SHA2_10_256) but a pk whose OID encodes XMSS-SHA2_16_256, the wrapper check passes (2,500 = 2,500), and the OID-confusion read proceeds. We confirmed this experimentally: the reproducer crashes against liboqs main at commit `2c8352c4`, which includes `ef70dea`.

C. ASan Report

Listing 2 reproduces the key lines from the ASan output on liboqs 0.15.0 (commit `97f6b86b`).

```
ERROR: AddressSanitizer: heap-buffer-overflow
READ of size 128 at 0x61e000000a84 thread T0
  #1 xmssmt_core_sign_open  xmss_commons.c:194
  #2 xmss_sign_open         xmss.c:127
  #3 OQS_SIG_STFL_alg_xmss_verify_xmss_sha256_h10
             sig_stfl_xmss_functions.c:79
0x61e000000a84 located 64 bytes after
  2500-byte region [0x61e000000080, 0x61e000000a44)
SUMMARY: heap-buffer-overflow xmss_commons.c:194
```

Listing 2. ASan output (key lines). Confirming heap-buffer-overflow at `xmss_commons.c:194` on liboqs 0.15.0.

D. Standalone Reproducer

The bug is reproducible without any fuzzer dependency. Listing 3 is the complete reproducer, confirmed on liboqs 0.15.0 (commit `97f6b86b`) and on `main` (commit `2c8352c4`, which includes the incomplete prior fix).

```
#include <stdlib.h>
#include <oqs/oqs.h>

int main(void) {
    OQS_init();
    OQS_SIG_STFL *sig =
        OQS_SIG_STFL_new("XMSS-SHA2_10_256");
    if (!sig) return 1;

    uint8_t *pk  = calloc(1, sig->length_public_key);
    uint8_t *sm  = calloc(1, sig->length_signature);
    uint8_t *msg = calloc(1, 33);

    /* OID 0x02: XMSS-SHA2_16_256, sig_bytes=2690 > 2500
       */
    pk[0]=0x00; pk[1]=0x00;
    pk[2]=0x00; pk[3]=0x02;

    OQS_SIG_STFL_verify(sig, msg, 33,
                        sm, sig->length_signature, pk);
    free(pk); free(sm); free(msg);
    OQS_SIG_STFL_free(sig);
    return 0;
}
```

Listing 3. Standalone C reproducer. Compile with `-fsanitize=address -g -O1 -loqs`. Minimal crashing input: OID bytes `00 00 00 02`.

E. Fix and Coordinated Disclosure

The vulnerability was disclosed to the OQS Vulnerability Management Team on 13 May 2026 (GHSA-2wxh-55qf-c7wg) under the OQS coordinated disclosure process [8]. The report included the standalone reproducer, the full ASan output, the root-cause analysis distinguishing the vector from CVE-2026-44518, and two proposed remediations.

Douglas Stebila prepared a fix (commit `58625ccf`) that adds an OID consistency check at the wrapper layer - decoding `pk[0..3]` as a big-endian OID and returning `OQS_ERROR` if it does not equal the variant's compile-time OID constant:

```
uint32_t pk_oid =
    ((uint32_t)public_key[0] << 24) |
    ((uint32_t)public_key[1] << 16) |
    ((uint32_t)public_key[2] <<  8) |
    ((uint32_t)public_key[3]);
```

TABLE II
EXECUTION SUMMARY ACROSS HARNESS VERSIONS

Harness	Stage	Iters (M)	Findings
XMSS	Random init	22.9	0
XMSS	KAT seed	18.5	0
XMSS	Visibility fix	0.10	**CVE-2026-46344**
LMS	Sequential fill (v1)	30.4	0
LMS	Partial rotation (v2)	30.4	0
LMS	Round-robin (v3)	33.0	0

Apple M-series arm64, clang 22, `-fsanitize=fuzzer,address`.

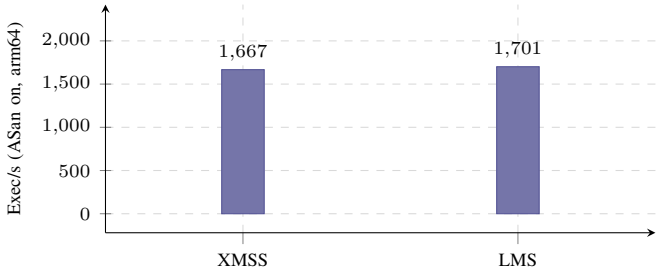

Fig. 4. Execution throughput for XMSS and LMS harnesses during ten-minute seeded fuzzing campaigns with ASan enabled on Apple M-series arm64 using clang 22. Both harnesses exceed 1,600 execs/s under the tested configuration.

```
if (pk_oid !=
    (uint32_t)OQS_SIG_STFL_alg_xmss##v##_oid)
    return OQS_ERROR;
```

We verified that the reproducer returns `OQS_ERROR` cleanly against this patch with no ASan crash. The fix ships in liboqs 0.16.0. CVE-2026-46344 was assigned by GitHub/MITRE; CVSS 3.1: `AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L` (5.3 Moderate); CWE-125.

VII. EVALUATION

A. Execution Summary

Table II records iteration counts across the development history of both harnesses, sourced from PR review threads and local smoke runs on the evaluation platform.

B. Patch Verification

After receiving the fix branch (`fix-xmss-oid-mismatch`, commit `58625ccf`), we rebuilt liboqs with `-fsanitize=address` and reran the standalone reproducer. `OQS_SIG_STFL_verify` returned `OQS_ERROR` cleanly with no ASan report, confirming that the OID consistency check at the wrapper layer blocks the mismatch before any memory access.

C. LMS Robustness

All three LMS harness versions executed over 30 million iterations each with zero memory-safety findings. In accordance with the evaluation guidance of Klees et al. [11], we do not interpret this as a correctness guarantee; it constitutes a robustness assurance for the exercised code paths.

D. Fuzzing Campaign Configuration

The XMSS and LMS harnesses were evaluated using LLVM libFuzzer with AddressSanitizer enabled on Apple M-series arm64 hardware using clang 22. Each campaign was executed for ten minutes using seeded KAT corpora and default coverage-guided mutation settings. The XMSS harness executed at approximately 1,667 exec/s, while the LMS harness executed at approximately 1,701 exec/s. During the XMSS campaign, the OID-confusion crash corresponding to CVE-2026-46344 was reproduced consistently from the generated corpus.

VIII. DISCUSSION

a) The smlen elision pattern warrants a broader audit: The `(void) smlen` construct may appear in other variants of the XMSS and XMSSMT verification kernel. Each occurrence represents a latent assumption: that the caller's buffer matches the OID-inferred parameter set. This assumption fails whenever `pk` is attacker-controlled. A systematic audit of all callers of `xmss_core_sign_open` and `xmssmt_core_sign_open` is warranted.

b) Defence-in-depth for OID-parameterised APIs: The CVE-2026-44518 fix and the CVE-2026-46344 fix each address one layer of the same structural problem: a caller-supplied buffer is indexed using a parameter derived from attacker-supplied data at a *different* layer. A complete defence requires validating the OID-derived parameter set against the declared algorithm at every layer that independently re-parses the OID. The architecture of `xmss_sign_open` - which re-reads `pk[0..3]` and calls `xmss_parse_oid` independently of the wrapper - is the structural root cause; the individual fixes are necessary mitigations within it.

c) Generalising the KAT-seeded pattern: The design pattern applies directly to any scheme where keygen is expensive relative to verify and KAT vectors are available. SLH-DSA (FIPS 205), HSS with deeper trees, and ML-DSA are immediate candidates. The round-robin field-rotation methodology applies to any multi-input verify function.

d) macOS toolchain visibility: The `LLVMFuzzerInitialize` symbol-visibility issue is not liboqs-specific: it affects any libFuzzer harness compiled with Homebrew LLVM on macOS. Verifying that the initialiser fires before committing to a long campaign is a universally applicable pre-condition check.

IX. RELATED WORK

a) Hash-based signature security: Buchmann et al. [17] establish the theoretical foundations of XMSS; Bernstein and Lange [18] survey the post-quantum landscape. Genet et al. [19] and Castelnovi et al. [20] demonstrate fault-injection attacks on SPHINCS, motivating adversarial scrutiny of HBS verification paths more broadly.

b) Cryptographic library fuzzing: Somorovsky [14] systematically fuzzed TLS libraries, finding implementation flaws across OpenSSL and peers. Walz and Sikora [15] applied differential fuzzing to TLS state machines. The OpenSSL project maintains an upstream fuzz corpus [16] as a practitioner

reference. This work extends that tradition to PQC stateful signatures, a category not covered by any prior published fuzzing study of which we are aware.

c) Fuzzing methodology: Bohme et al. [12] provide theoretical grounding for coverage-guided grey-box fuzzing; Klees et al. [11] establish empirical evaluation standards that inform our null-finding framing. Aschermann et al. [13] introduce input-to-state correspondence as a complementary technique for improving coverage of parsing-heavy paths.

d) OQS security assessments: Trail of Bits conducted a security review of liboqs in 2025 [21]. That review did not examine the stateful HBS verification paths, which are the subject of the present work.

X. Conclusion

This work demonstrates that stateful post-quantum signature verification paths can be fuzzed effectively despite the high cost of stateful key generation. By combining KAT-seeded initialisation with structured multi-field mutation, the proposed methodology enables deep adversarial coverage of XMSS and LMS verification logic within `liboqs`. The proximate cause is the unsafe assumption that the caller-supplied signature length is redundant after the XMSS parameter set is derived internally. A related fix had already been shipped for a shorter-buffer attack vector (CVE-2026-44518), but the OID-confusion vector remained open because the inner function re-parses the OID independently of the wrapper layer. Our fuzz harness surfaced this gap during the seeded fuzzing campaign.

The KAT-seeded harness design pattern resolves the keygen cost barrier that has historically made stateful HBS a difficult fuzzing target. Three-field mutation with round-robin field rotation provides balanced adversarial coverage regardless of corpus size. Both harnesses are merged into the liboqs upstream; CVE-2026-46344 is assigned, and the fix ships in liboqs 0.16.0.

a) Future work: Complete line/branch/function coverage measurements with KAT-only baseline; structured 24-hour Linux x86_64 evaluation; ablation study of the round-robin rotation; systematic audit of the LMS verification path for analogous `smlen`-style assumptions; extension of the harness pattern to SLH-DSA (FIPS 205) and ML-DSA.

Acknowledgment

The author thanks Douglas Stebila (`dstebila`) for accepting the report, preparing the fix in `58625ccf`, and requesting the CVE; Norman Ashley (`ashman-p`) for reviewing both the fix and PRs #2400 and #2412; and Michael Baentsch (`baentsch`) for identifying the field-mutation bias that motivated the round-robin rotation design.

References

[1] D. Cooper, D. Apon, Q. Dang, M. Davidson, M. Dworkin, and C. Miller, "Recommendation for stateful hash-based signature schemes," NIST Special Publication 800-208, Oct. 2020.

[2] A. Hulsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: eXtended Merkle signature scheme," IETF RFC 8391, May 2018.

[3] D. McGrew, M. Curcio, and S. Fluhrer, "Leighton-Micali hash-based signatures," IETF RFC 8554, Apr. 2019.

[4] D. Stebila and M. Mosca, "Post-quantum key exchange for the Internet and the Open Quantum Safe project," in *Proc. SAC 2016*, LNCS 10532, pp. 14–37, Springer, 2017.

[5] V. Ajith, "tests: add fuzz harness for XMSS stateful signature verification," *liboqs* PR #2400, merged 16 Apr. 2026. [Online]. Available: https://github.com/open-quantum-safe/liboqs/pull/2400

[6] V. Ajith, "tests: add fuzz harness for LMS stateful signature verification," *liboqs* PR #2412, merged 30 Apr. 2026. [Online]. Available: https://github.com/open-quantum-safe/liboqs/pull/2412

[7] Open Quantum Safe Project, "liboqs security policy," https://github.com/open-quantum-safe/liboqs/blob/main/SECURITY.md, 2026.

[8] Open Quantum Safe Project, "OQS coordinated vulnerability disclosure process," https://github.com/open-quantum-safe/tsc/blob/main/security/response-process.md, 2025.

[9] K. Serebryany, "libFuzzer – a library for coverage-guided fuzz testing," [Online]. Available: https://llvm.org/docs/LibFuzzer.html, 2015.

[10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX ATC*, pp. 309–318, 2012.

[11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM CCS*, pp. 2123–2138, 2018.

[12] M. Bohme, V. T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as a Markov chain," in *Proc. ACM CCS*, pp. 1032–1043, 2016.

[13] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proc. NDSS*, 2019.

[14] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proc. ACM CCS*, pp. 1492–1504, 2016.

[15] A. Walz and A. Sikora, "Exploiting dissent: Towards fuzzing-based differential black-box testing of TLS implementations," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 3, pp. 578–590, 2020.

[16] OpenSSL Project, "OpenSSL fuzz corpus," https://github.com/openssl/openssl/tree/master/fuzz, 2026.

[17] J. Buchmann, E. Dahmen, and A. Hulsing, "XMSS – A practical forward secure signature scheme based on minimal security assumptions," in *Proc. PQCrypto 2011*, LNCS 7071, pp. 117–129, Springer, 2011.

[18] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, pp. 188–194, 2017.

[19] A. Genet, M. J. Kannwischer, H. Pessl, and A. Sigl, "Practical fault injection attacks on SPHINCS," in *Proc. CHES 2018*, TCHES vol. 2018:3, pp. 143–168, 2018.

[20] L. Castelnovi, A. Martinelli, and T. Prest, "Grafting trees: A fault attack against the SPHINCS framework," in *Proc. PQCrypto 2018*, LNCS 10786, pp. 165–184, Springer, 2018.

[21] Trail of Bits, "Security review of liboqs," Apr. 2025. [Online]. Available: https://github.com/trailofbits/publications/blob/master/reviews/2025-04-quantum-open-safe-liboqs-securityreview.pdf

[22] Cisco Systems, "IOS XR software image signing," Cisco Technical Documentation, 2024.