# Hard-to-Find Bugs in a Post-Quantum Age

Matteo Steinbach, Peter B. Rønne, and Johann Großschädl

DCS and SnT, University of Luxembourg,
6, avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
matteo.steinbach.pro@gmail.com
{peter.roenne,johann.groszschaedl}@uni.lu

**Abstract.** The transition to Post-Quantum Cryptography (PQC) replaces familiar primitives with theoretically secure but implementationally immature algorithms built on structured lattices and polynomial rings. This shift introduces subtle implementation defects—Hard-to-Find Bugs (HFBs)—that can cause catastrophic failures while evading conventional testing. We show that the HFB profile of PQC differs fundamentally from classical cryptography: carry propagation bugs, common in classical schemes, are nearly absent, while timing side-channels in polynomial arithmetic (e.g., KyberSlash) and precision divergences in floating-point operations in Falcon dominate. To address this new attack surface, we present a systematic taxonomy of PQC-specific HFBs and `wycheproof-pqc`, an open-source framework extension that uses targeted Known Answer Tests to target elusive bugs and the documentation of 15+ vulnerabilities in major open-source PQC implementations, providing a caution for securing the next generation of cryptography.

**Keywords:** Public Key Cryptography · Hard-to-Find Bugs · Post Quantum Cryptography · Software Testing · Known Answer Tests

## 1 Introduction

Cryptographic software is foundational to digital security, ensuring confidentiality, authenticity, and integrity in modern communication systems. The security of these systems, however, relies not only on sound theoretical constructions but also on the correctness and robustness of their implementations. Subtle and elusive defects, termed *Hard-to-Find Bugs (HFBs)*, can undermine cryptographic guarantees by causing rare but catastrophic failures such as incorrect verification, exploitable side-channel leakage, or even private key exposure. This paper extends earlier work [1], where we formally defined HFBs, highlighted their significance, and classified over 50, with a particular focus on methods for testing and verifying cryptographic software. In this paper, we extend that foundation by concentrating on the unique challenges of identifying, analyzing, and mitigating HFBs in implementations of standardized and pre-standardized post-quantum cryptography (PQC). Specifically, we examine lattice-based schemes

[1] Previous paper on classic HFBs available here: https://github.com/mattc-try/wycheproof-c/README.md

(ML-KEM, ML-DSA, and FN-DSA), hash-based schemes (SLH-DSA), and code-based schemes (HQC no FIPS standard name yet).

*Contributions* This paper makes the following contributions:

1. **Taxonomy of PQC-specific HFBs:** We identify and systematize vulnerabilities into classes of defined HFBs.
2. **Analysis of specific failures:** We reproduce and document concrete failures, such as Falcon's acceptance of mutated signatures due to floating-point inconsistencies, and highlight their security implications.
3. **Tailored Known Answer Tests (KATs):** We extend the Wycheproof methodology to PQC, generating adversarial test vectors that expose elusive bugs.
4. **Mitigation strategies:** We propose countermeasures for each class of vulnerability, and good secure implementations practices.
5. **Practical artifact:** We release *wycheproof-pqc*, an open-source repository for testing PQC implementations.

*Paper Structure* Section 2 formalizes our threat model and definitions. Section 3 situates our work in the context of prior testing frameworks and PQC implementation studies. Section 4 presents where PQC could potentially fail creating HFBs. Section 5 reports experimental results and reproduced failures.

*Research Questions* This work addresses the following questions:

1. **RQ1:** How do the mathematical and algorithmic properties of PQC schemes lead to new categories of hard-to-find bugs compared to classical cryptography?
2. **RQ2:** What systematic methodologies can effectively detect, classify, and reproduce these PQC-specific HFBs?
3. **RQ3:** How can KATs and adversarial vectors be leveraged to close gaps left by conventional conformance and fuzzing tests, thereby ensuring robust PQC implementations?

## 2 Definition and Threat

**Definition 1 (Hard-to-Find Bug).** *An HFB is an implementation defect that:*

1. *manifests only under rare inputs, execution orders, hardware/compiler settings, or environmental conditions;*
2. *can violate cryptographic guarantees (e.g., cause over-acceptance, leak secret bits, or corrupt ciphertexts); and*
3. *is unlikely to be found by standard conformance tests or lightweight fuzzing.*

*A more detailed definition is provided in Appendix A.*

*Security Risks of HFBs* The elusive nature of HFBs makes them valuable to adversaries. Exploitation often requires carefully crafted inputs or sequences that trigger rare conditions, yet once triggered, an HFB can compromise core cryptographic guarantees. For example, a rounding divergence in floating-point Gaussian samplers may lead to biased signatures, while malformed but accepted encodings may bypass integrity checks. Importantly, HFBs are not only accidental: they could also be *maliciously introduced*. A subtle branch misprediction leak or deliberately weakened parser check, invisible under standard tests, could remain undetected for years before being exploited.

*Threat* Unlike classical cryptography, which has benefited from decades of refinement and accumulated expertise, PQC is comparatively immature. Its algorithms rely on novel mathematical structures—such as polynomial rings, structured lattices, and large linear codes—that differ fundamentally from classical number-theoretic primitives. This novelty, combined with a lack of specialized testing tools and limited implementation experience, creates fertile ground for subtle HFBs. These bugs are particularly interesting as they may remain dormant for long periods, evading conventional validation while still being exploitable.

*Deployment Pressures* The risk is exacerbated by the accelerated standardization roadmaps for PQC, notably NIST2025 [23] and European comission [2] for pqc which target deprecation by 2030. Engineers—often without deep cryptographic backgrounds—are required to implement and deploy new primitives rapidly across software and hardware platforms. This combination of urgency, novelty, and immaturity increases the probability that HFBs will persist in widely deployed libraries. Moreover, the geopolitical landscape underscores the risk: algorithms standardized and certified in one jurisdiction may contain defects that are practically undetectable, manifesting only under astronomically rare conditions (e.g., once in $2^{64}$ executions). Such difficulty in detection motivates some countries to develop independent standards and implementations, relying on their own experts and resources to reduce exposure to potential hidden defects.

## 3 Related Work

The landscape of cryptographic implementation testing and vulnerability analysis has evolved significantly, yet substantial gaps remain in addressing PQC unique challenges. This section critically examines existing approaches across three key domains: systematic cryptographic testing frameworks, side-channel detection methodologies, and post-quantum implementation security initiatives.

*Systematic Cryptographic Testing Frameworks* Google's Project Wycheproof provides adversarial test vectors for classical algorithms (RSA, ECDSA, ECDH, AEAD, etc.), uncovering 40+ vulnerabilities across libraries [7]. Its scope, however, is limited to classical primitives and attacks (e.g., invalid curves, biased

nonces) and does not address PQC-specific issues such as NTT implementations, Gaussian sampling, or non-canonical encodings.

During standardization, NIST offered only basic KATs for functional verification [4], omitting adversarial and misuse testing [16]. As a result, several flawed implementations persisted through multiple evaluation rounds.

*Automated Side-Channel Detection* CipherH is the state-of-the-art in automated side-channel detection, combining taint tracking with symbolic execution to identify ciphertext-dependent leaks [11]. Applied to RSA and ECDSA in major libraries, it revealed 236 vulnerabilities within 28 CPU hours, showing better scalability than traditional static analysis.

However, CipherH targets classical cryptographic patterns and does not capture PQC-specific behaviors such as NTT reductions, discrete Gaussian sampling loops, or floating-point rounding effects. Addressing these would require new taint rules and constraint models.

Other tools, such as HACL* and Jasmin, offer formal verification but require full re-implementation in domain-specific languages, limiting their use for existing C/C++ PQC and highly optimized code [9].

*Post-Quantum Implementation Security Initiatives* The EU's PQCRYPTO project (2015–2018) advanced PQC through over 130 publications, 22 NIST submissions, and reference implementations including the `pqm4` framework [18]. While it noted timing and sampler leakage issues, its implementation security analysis was fragmented and lacked systematic taxonomies for PQC-specific HFBs [31].

The community-driven PQClean project has become the main effort to improve implementation quality [16]. Continuous integration testing across 17 NIST schemes exposed widespread flaws such as memory errors and API violations. However, PQClean mainly addresses general software robustness rather than subtle PQC-specific HFBs like floating-point precision drift, polynomial timing inconsistencies, or sampling deviations.

NIST's PQC evaluation emphasized theoretical cryptanalysis and only documented implementation attacks reactively (e.g., KyberSlash, template, and fault injection) rather than pursuing systematic HFB detection [4].

*Critical Gaps and Research Positioning* Mathematical Foundation Analysis: Neither Wycheproof nor CipherH provides systematic analysis of how PQC's underlying mathematical structures (polynomial rings, lattice geometry, discrete distributions) can create implementation vulnerabilities distinct from classical cryptography.

PQC-Specific Vulnerability Taxonomies: While PQClean has improved general software quality, no existing work provides comprehensive taxonomies specifically for PQC HFBs that exploit the mathematics-implementation interface.

Our research directly addresses these limitations by providing the first systematic framework for identifying, classifying, and detecting hard-to-find bugs that arise specifically from the mathematical properties and implementation constraints of PQC systems.

## 4 Mathematical Foundations and PQC-Specific HFB Manifestations

The transition from classical to PQC represents a fundamental shift in mathematical foundations, with profound implications for implementation security and HFB manifestation patterns. Classical cryptosystems—RSA, ECDSA, and related schemes—rely on number-theoretic problems involving large-integer arithmetic in cyclic groups and finite fields. In contrast, post-quantum cryptosystems derive security from diverse mathematical structures including lattice problems, coding theory, multivariate systems, and isogeny-based constructions.

This mathematical divergence creates distinct HFB profiles. As established in our previous work, HFBs in classical cryptography most frequently involve carry propagation during multi-precision arithmetic operations. However, carry propagation is much less frequent in PQC software, compared to more than 30% in classical systems. This radical reduction stems from fundamental mathematical differences in operand structure and arithmetic domains.

*The Absence of Carry Propagation in PQC:* Carry propagation bugs arise from the sequential processing of multi-precision integers in classical cryptography. Consider RSA modular exponentiation: computing $c = m^e \mod n$ requires handling integers of size $|n| = 2048$ bits or larger, implemented using multi-limb representations where carries must propagate across limb boundaries:

Classical: $\mathbb{Z}_n$ with $n \approx 2^{2048} \Rightarrow$ multi-limb arithmetic $\Rightarrow$ carry propagation

PQC: $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q < 2^{16} \Rightarrow$ single-limb coefficients

$\Rightarrow$no cross-coefficient carries

In lattice-based cryptography, operations occur in polynomial rings $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ where the modulus $q$ is typically small ($q = 3329$ for Kyber, $q = 8380417$ for Dilithium, [24,25]). Coefficients remain bounded within $[0, q-1]$, eliminating the need for carry propagation between polynomial coefficients. Polynomial multiplication in $R_q$ involves coefficient-wise operations modulo $q$, fundamentally avoiding the multi-precision arithmetic that characterizes classical cryptosystems.

Hash-based schemes such as XMSS and SPHINCS+ [1,26] avoid carry propagation because their security relies on cryptographic hash functions

$$H : \{0,1\}^n \to \{0,1\}^m$$

implemented as fixed-size Boolean circuits. Operations like XOR, AND, and modular addition over a fixed word size $w \in \{32, 64\}$ are inherently bounded, preventing variable carry chains. Similarly, code-based schemes such as McEliece [19, 21] operate over finite fields $\mathbb{F}_q$, where addition and multiplication are algebraic operations (e.g., XOR, AND, or polynomial arithmetic) that are carry-free by construction. In both cases, the arithmetic structure guarantees deterministic, bounded operations without cross-term carry propagation.

### 4.1 Lattice-Based Cryptography: Where Mathematical Complexity Creates HFBs

Lattice-based schemes, including NIST standards ML-KEM (Kyber), ML-DSA (Dilithium), and FN-DSA (Falcon), form the core of PQC [3]. Their security relies on structured lattices implemented via polynomial arithmetic in quotient rings, creating a distinct attack surface for HFBs, primarily timing side-channels, floating-point inconsistencies, and logical flaws.

**4.1.1 Timing Side-Channels in Polynomial Arithmetic** Constant-time execution is critical for cryptographic code, yet the complex arithmetic in lattice schemes creates subtle opportunities for timing leaks.

*(a) Conditional Reductions in NTT Operations.* Lattice-based schemes operate in polynomial rings of the form $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where polynomial multiplication is accelerated by the Number Theoretic Transform (NTT). The core NTT "butterfly" operation usually contains a conditional modular reduction:

$$(u, v) = (a + \omega b \pmod q, a - \omega b \pmod q).$$

A naive implementation of this reduction introduces a data-dependent branch, creating a vulnerability that leaks information about secret coefficients.

```
int32_t t = a + omega * b;
if (t >= q) t -= q; // TIMING_GAP: Secret-dependent branch
```

**Listing 1.1.** Vulnerable conditional reduction in NTT.

The standard mitigation replaces this branch with constant-time arithmetic using bitwise masking.

```
// Constant-time modular reduction
int32_t t = a + omega * b;
t -= q;
int32_t mask = t >> 31; // Arithmetic shift: -1 if t < 0, 0
    otherwise
t += (q & mask);        // Add q back if original t was < q
```

**Listing 1.2.** Constant-time modular reduction mitigation.

To verify the mitigation, one can measure the runtime variance of the NTT kernel across a set of crafted inputs. Implementations where timing differs significantly between inputs that trigger the reduction branch and those that do not indicate remaining vulnerabilities.

*(b) Variable-Time Division in Coefficient Compression.* To reduce the size of public keys and ciphertexts, Kyber compresses polynomial coefficients. This step can inadvertently introduce a timing leak if it compiles to a variable-latency integer division instruction (`idiv` on x86). The KyberSlash [6] attacks demonstrated this vulnerability allows for full private key recovery with only $O(n)$ decapsulation queries.

```
// Compiles to a variable-time idiv instruction
uint16_t compressed = (coefficient * 16) / KYBER_Q;
```

**Listing 1.3.** Vulnerable use of integer division for compression.

This is mitigated by replacing division with a constant-time algorithm like Barrett reduction given that it is implemented correctly.

```
// Precomputed constant: (1 << 28) / KYBER_Q
uint32_t t = (uint32_t)coefficient * BARRETT_MULTIPLIER;
uint16_t compressed = t >> 28;
```

**Listing 1.4.** Constant-time Barrett reduction for compression.

Testing this mitigation can be done using specially crafted ciphertexts to maximize timing variance in the division operation can be detected this way also through KATs. The test fails if timing differences remain after rewriting the code to be division-free.

**4.1.2 Precision Divergence in Gaussian Sampling** Discrete Gaussian sampling is a fundamental building block of lattice-based signatures. While most NIST lattice standards implement noise generation using integer-only samplers, Falcon is unique in relying on floating-point arithmetic. In Falcon, signatures require sampling from discrete Gaussian distributions over $\mathbb{Z}$, defined as:

$$D_{\mathbb{Z},\sigma}(x) = \frac{\exp(-x^2/2\sigma^2)}{\sum_{y\in\mathbb{Z}} \exp(-y^2/2\sigma^2)}, \quad x \in \mathbb{Z}.$$

To achieve efficient sampling, Falcon employs fast Fourier transforms (FFT) and double-precision floating-point approximations of the exponential function.

*Cross-Platform Precision Gaps.* Floating-point units (FPUs) are not fully standardized across hardware architectures. For instance, differences between x86 (which supports fused multiply-add, FMA) and ARM architectures can lead to slight variations in floating-point computations. Nguyen et al. (2023) report that up to 7% of exponential evaluations may diverge between platforms [27]. Consequently, identical key–message pairs may yield distinct yet valid Falcon signatures depending on the underlying hardware.

These precision discrepancies can have serious security implications. Potii et al. (2024) demonstrated that collecting approximately 300,000 divergent signatures is sufficient to recover the secret trapdoor with a success rate of 70–76% [29]. Similarly, the PQClean project identified inconsistencies in Falcon implementations on ARM platforms, where mutated signatures were incorrectly accepted, further illustrating the practical impact of these cross-platform precision gaps [30].

*Mantissa Leakage.* Beyond rounding divergences, Falcon's reliance on FFT-based sampling introduces side-channel risks. Each complex coefficient

$$\widehat{a}_j = u_j + iv_j, \quad u_j, v_j \in \mathbb{R},$$

is stored in IEEE-754 double-precision format with a 52-bit mantissa. On AVX-512 platforms, vectorized loads leak the Hamming weights of mantissas through electromagnetic (EM) emissions:

$$E(t) \propto \sum_j \left[ \mathrm{HW}(m_{u,j}) + \mathrm{HW}(m_{v,j}) \right],$$

where $\mathrm{HW}(\cdot)$ denotes the Hamming weight. Correlation Power Analysis (CPA) on such traces enables recovery of FFT coefficients, which after inverse FFT and lattice reduction reveal Falcon's trapdoor polynomials $(f, g)$ [13, 17].

**4.1.3 Algebraic Correctness and Plaintext-Checking Oracles** The algebraic structure of the Ring Learning with Errors (RLWE) problem can be exploited if an implementation's logic reveals information about decryption correctness. During decapsulation, a received ciphertext is used to compute a message, and correctness requires that the coefficients of the resulting error term are small (i.e., $|e_i| < q/2$). A vulnerable implementation might perform an early exit after detecting an invalid coefficient, leaking timing information that an attacker can use as a plaintext-checking oracle.

```
int is_valid = 1;
for (int i = 0; i < N; ++i) {
    if (abs(coeffs[i]) > (Q / 2)) {
        is_valid = 0;
        break; // TIMING ORACLE: Early exit
    }
}
```

**Listing 1.5.** Vulnerable early-exit logic in verification.

Such an oracle can be used to recover a Kyber-512 key with just a few thousand queries. The vulnerability is mitigated by ensuring all checks run in constant time, removing any secret-dependent early exits. [6]

```
uint16_t mask = 0;
for (int i = 0; i < N; ++i) {
    // Bitwise OR accumulates failure flags without branching
    mask |= (abs(coeffs[i]) > (Q / 2));
}
// Final check is based on the accumulated mask
int is_valid = (mask == 0);
```

**Listing 1.6.** Constant-time verification logic.

Testing for this vulnerability can be done by providing pairs of ciphertexts: one valid and one intentionally failing the norm check on the first coefficient. The test fails if the runtime difference exceeds a small, fixed threshold, indicating a potential early-exit vulnerability. [22], [34].

### 4.2 Code-Based Cryptography: linear algebraic core, concrete HFBs

Code-based cryptography derives its security from the hardness of the syndrome decoding problem for linear codes. Formally, given a parity-check matrix $H \in \mathbb{F}_q^{m \times n}$ and a syndrome $s \in \mathbb{F}_q^m$, the decoding problem asks for an error vector $e \in \mathbb{F}_q^n$ with $\mathrm{wt}(e) \leq t$ such that $He^T = s^T$. This problem is NP-complete [5] in the binary case and naturally extends to higher fields $\mathbb{F}_q$, a foundation that underpins schemes such as Classic McEliece and HQC [28]. The canonical McEliece-style encryption $c = mG + e$ is compact linear algebra over $\mathbb{F}_q$ and, at the algorithmic level, admits implementations without secret-dependent control flow.

However, algebraic simplicity does not guarantee implementation safety: many low-level building blocks (finite-field arithmetic, samplers, decoders, polynomial kernels) are engineering artifacts whose concrete implementations can introduce HFBs. Important implementation hotspots that have produced exploitable leakage include:

- Finite-field and polynomial arithmetic: implementations that rely on table lookups, variable-latency instructions, or non-constant modular reduction can leak through timing or memory-access side channels.
- Constant-weight (weight-$t$) samplers: naive samplers implemented with variable rejection loops, data-dependent shuffles, or table draws reveal weight patterns unless the sampler is written to enforce fixed iteration counts or is masked.
- Decoder behaviour: iterative decoders (bit-flipping, belief propagation, or hybrid concatenated decoders) whose iteration counts, branch behaviour, or memory accesses depend on secret errors produce reaction channels that an adversary can amplify (chosen-ciphertext strategies) to obtain oracles leaking secret information.
- Structured code surfaces: use of quasi-cyclic or other structured codes reduces key sizes but introduces distinguishers and weak-key regions; these structures broaden the HFB surface by enabling structure-specific attacks and reaction amplification.

HQC provides concrete, instructive examples: timing/division oracles from non-constant modular reductions, chosen-ciphertext amplification to steer decoder failures, pre-reencryption side-channel oracles on the RM/RS path, and single-trace SASCA results against RS syndrome kernels and polynomial multiplies. These attacks demonstrate how finite-field / polynomial kernels and decoders — though algebraically simple — create HFBs unless carefully hardened. [33]

The algebraic simplicity of code-based cryptography makes many building blocks easier to reason about and harden, but does not obviate HFBs. HQC and other real-world studies show that constant-time behaviour, sampler correctness, decoder failure handling, and protection against reaction/fault oracles must be engineered and verified end-to-end. Empirical case studies and KAT-style tests are valuable complements to formal proofs for achieving robust implementations.

Both were actively encouraged by the NIST in the course of their post-quantum standardization projects.

### 4.3 Hash-based signatures: Minimal HFB Surface

Hash-based signature schemes (for example, SPHINCS+, XMSS, and LMS) build security solely from standard hash properties (preimage resistance, second-preimage resistance, and—where required—collision resistance), using one-time/one-time-like chains and Merkle trees. This reliance on only hash assumptions yields a minimal algebraic attack surface among the post-quantum families and avoids number-theoretic or lattice assumptions entirely.

Merkle authentication paths are produced by iteratively hashing concatenated sibling nodes up to the root using a cryptographic hash function $H$; verification recomputes the path and checks equality with the public-key root node. Formally, given leaf $L$ and authentication siblings $S_1, \ldots, S_h$ the root is computed by $R = H\big(\cdots H(H(L\|S_1)\|S_2)\cdots\big)$, and the verifier accepts if the recomputed root equals the public key root.

Winternitz one-time signatures (W-OTS+) are vector-valued chains. Key generation samples $l$ secret seeds $(x_1, \ldots, x_l)$ and computes chain endpoints $\mathsf{pk}_i = F^{w-1}(x_i)$, where $F$ is an iterated hash/compression and $w$ is the Winternitz parameter. The message digest (plus checksum) is encoded in base-$w$ to select which chain positions are revealed; a verifier advances each revealed chain element to its endpoint and authenticates the aggregated endpoints via the Merkle path [14].

SPHINCS+ is a stateless, hypertree-based construction combining many FORS trees, W-OTS+ chains, and layered Merkle trees; it was standardized (as SLH-DSA) by NIST and offers a conservative, stateless alternative to stateful schemes. XMSS and LMS (and their hierarchical variants) are stateful and are profiled in NIST SP 800-208: they require application environments that can enforce strict private-state management such as HSMs or dedicated firmware signing workflows [1, 12].

*Minimal attack surface — and remaining pitfalls.* Because security reduces to well-studied hash properties, the mathematical attack surface is small compared with algebraic constructions. Nevertheless, implementations must still address practical HFBs and side-channel/fault vectors:

- State management (XMSS/LMS). Stateful schemes are vulnerable to catastrophic key reuse if state updates are lost or mis-synchronized; secure deployment requires atomic state updates and audited backup/restore procedures.
- Side channels and constant-time. SPHINCS+ is stateless but still requires constant-time kernels, robust parsing, and secure randomness/seed handling to avoid timing or microarchitectural leakage.
- Fault injection. Faults that induce inconsistent intermediate W-OTS+ values, Merkle node corruptions, or truncated recomputation can enable forgeries or key recovery; such attacks have been demonstrated against SPHINCS-like designs and adapted to XMSS.

### 4.4 Comparison Summary Across PQC Families

The manifestations of HFBs differs sharply across PQC families. These differences can be traced directly to the mathematical foundations and implementation requirements of each scheme type. Table 1 provides a structured comparison.

**Table 1.** Comparison of HFB Characteristics Across PQC Families

| **Scheme Family** | **Dominant Mathematics** | **Primary HFB Types** | **Underlying Causes / Remarks** |
|---|---|---|---|
| Lattice-based (Kyber, Dilithium, Falcon) | Polynomial rings, NTT, discrete Gaussian sampling | Timing side-channels (NTT reductions, coefficient compression); Precision divergences (floating-point sampling, Falcon); Oracle leaks from early-exit logic. | Complex arithmetic with secret-dependent branches; reliance on floating-point units; immaturity of implementations |
| Code-based (HQC, Classic McEliece) | Linear algebra over finite fields ($\mathbb{F}_q$) | Memory-safety bugs (buffer overflows, API misuse); Rare fault-injection vulnerabilities. | Finite-field operations are inherently constant-time; HFBs mainly stem from conventional programming errors rather than mathematics |
| Hash-based (SPHINCS+, XMSS, LMS) | Cryptographic hash functions, Merkle trees, hash chains | State-management errors (key reuse in stateful variants); Fault injection in tree structures. | Bit-oriented, deterministic computations with minimal mathematical complexity; HFBs largely software-engineering or physical-attack induced |

In summary, lattice-based schemes exhibit the richest and most diverse HFB profiles, with multiple categories of subtle mathematical and platform-dependent vulnerabilities. Code-based schemes show far fewer cryptographic-specific HFBs, with their risks being dominated by classical implementation errors. Hash-based schemes present the smallest mathematical HFB surface, with residual issues restricted to state management and specialized fault attacks.

More information on mitigating these HFBs possibly present in current and future implementations are described in the Appendix B.

## 5 Implementation stats and results

To detect HFBs we provide wycheproof-pqc[2], an extension of the Wycheproof testing framework for PQC. We developed a C-based test harness and a structured methodology for generating KATs, malformed vectors, and adversarial inputs targeting PQC schemes, directly adressing the risks put forward by this paper.

### 5.1 Research and vector generations

The main contribution is the set of vectors targetting ML-KEM, ML-DSA, FN-DSA, SLH-DSA and HQC specifically;

These vectors are targeted at low occurence HFBs and include prior existing HFBs found from open source implementations of PQC.

The researched bugs that were identified around 15 examples are documented methodically and were added to the HFB collection started while working on the first paper. All validated failures were documented with the structured format used throughout the study (Specification, Defect, Impact, Code Snippet) so they can be tracked and reproduced by implementers.

Test generation combined conformance testing with adversarial input crafting:

- Baseline KATs: We integrated official NIST conformance vectors.
- Bug Reproduction: We reproduced known issues from bug research and bugs in open source PQC libraries such as PQClean, LibOQS / liboqs integrations, TLS stacks that include PQC etc.
- Input Mutation: We systematically crafted edge-case vectors by:
  - Bit-flipping signature bytes.
  - Introducing non-canonical encodings and redundant padding.
  - Creating signatures with single-byte offsets.

### 5.2 Capabilities

Each KAT encodes one of the HFB classes established in Section 4. For each vector the tests perform:

1. functional verification (expected true/false);
2. negative/over-acceptance checks (vectors that only a lax parser should accept);
3. timing variability probes (micro-benchmarks around candidate branches compiled with realistic optimization flags);
4. cross-platform comparison (same vector executed on x86/ARM builds to detect precision/FPU differences).

[2] wycheproof-pqc repository, with the technical contribution of the paper is available here: https://github.com/mattc-try/wycheproof-pqc/README.md

## 6 Conclusion

The transition to PQC introduces a sophisticated and distinct class of implementation defects, fundamentally diverging from classical schemes where common carry propagation bugs are nearly absent. The new critical threats are dominated by *timing side-channels* in polynomial arithmetic (such as conditional NTT reductions or variable-time division) and precision divergences in floating-point Gaussian sampling (as documented in FALCON), particularly within lattice-based constructions (ML-KEM, ML-DSA, FN-DSA) which exhibit the richest HFB profile.

To counter these evasive, PQC-specific HFBs, we presented a systematic taxonomy and released `wycheproof-pqc`, an open-source extension of the Wycheproof test library. This practical artifact provides tailored *(KATs)* that specifically incorporate adversarial vectors, timing variability probes, and cross-platform comparisons to expose these defects, leading to the documentation of 15 vulnerabilities and establishing PQC-specific KATs as an effective, low-effort defense mechanism crucial for securing the accelerating PQC deployment roadmap.

### 6.1 Limitations

- The library and testing results come from HFBs found so far, and thus are not as complete as the work made on classical HFBs, this library helps with defense in depth were regular testing fails.
- Some discovered issues require physical side-channel or fault injection to fully validate exploitability; our KATs detect the software precursors but do not substitute for full hardware-level validation.
- This should not be the only way to test cryptographic implementations and should be used along with other methods, as conjectured in the classical paper contribution formal verification would be the best when possible, regular testing, HFBs for KATs and fuzzing when too complicated or costly.

## References

1. Recommendation for stateful hash-based signature schemes. Technical Report NIST SP 800-208, National Institute of Standards and Technology, 2020.
2. Commission recommendation (eu) 2024/1101 on a coordinated implementation roadmap for the transition to post-quantum cryptography. Technical Report 2024/1101, European Commission, April 2024.
3. The mathematical foundation of post-quantum cryptography. *Research*, 2024.
4. Gorjan Alagic et al. Status report on the third round of the nist pqc standardization process. Technical report, National Institute of Standards and Technology, 2022.
5. Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information theory*, 24(3):384–386, 2003.
6. Daniel J. Bernstein et al. Kyberslash: Exploiting secret-dependent division timings in kyber implementations. 2024. Detailed analysis of timing vulnerabilities in Kyber's NTT implementation.

7. Daniel Bleichenbacher and Thai Duong. Project wycheproof, 2016. Google Security Blog, December 2016.
8. Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.
9. Alexandre Braga and Ricardo Dahab. A survey on tools and techniques for the programming and verification of secure cryptographic software. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, pages 30–43. SBC, 2015.
10. Pietro Bressana, Noa Zilberman, and Robert Soulé. Finding hard-to-find data plane bugs with a pta. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 218–231, 2020.
11. Sen Deng, Mengyuan Wu, Huixin Dai, et al. Cipherh: Automated detection of ciphertext side-channel vulnerabilities in cryptographic software. In *32nd USENIX Security Symposium*, pages 289–306, 2023.
12. Scott Fluhrer and David A. McGrew. Leighton-micali hash-based signatures. RFC 8554, Internet Engineering Task Force, April 2019.
13. Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 141–164, 2022.
14. Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, Internet Engineering Task Force, May 2018.
15. Marc Joye, Michael Tunstall, et al. *Fault analysis in cryptography*, volume 147. Springer, 2012.
16. Matthias J Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 19–30. IEEE, 2022.
17. Emre Karabulut and Aydin Aysu. Falcon down: Breaking falcon post-quantum signature scheme through side-channel attacks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 691–696. IEEE, 2021.
18. Tanja Lange et al. Pqcrypto: Post-quantum cryptography for long-term security, 2018. EU Horizon 2020 Project ICT-645622.
19. Rudolf Lidl and Harald Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1997.
20. Aleksandra V Markelova. Vulnerability of RSA algorithm. In *CEUR Workshop Proceedings*, volume 2081, pages 74–78, 2017.
21. Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN Progress Report*, 42-44:114–116, 1978.
22. Puja Mondal, Suparna Kundu, Sarani Bhattacharya, Angshuman Karmakar, and Ingrid Verbauwhede. A practical key-recovery attack on lwe-based key-encapsulation mechanism schemes using rowhammer. In *International Conference on Applied Cryptography and Network Security*, pages 271–300. Springer, 2024.
23. Dustin Moody. Nist pqc: The road ahead. Presentation (March 2025), March 2025. NIST PQC process status, transition guidance highlights, and timelines.
24. National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical Report FIPS 204, U.S. Department of Commerce, 2024.

25. National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical Report FIPS 203, U.S. Department of Commerce, 2024.
26. National Institute of Standards and Technology. Stateless hash-based digital signature standard. Technical Report FIPS 205, U.S. Department of Commerce, 2024.
27. D. Nguyen and K. Gaj. Floating-point inconsistencies in discrete gaussian sampling. *IEEE Transactions on Computers*, 2023. Analysis of platform-dependent floating-point behavior in Gaussian sampling.
28. Raphael Overbeck and Nicolas Sendrier. Code-based cryptography. In *Post-quantum cryptography*, pages 95–145. Springer, 2009.
29. Oleksandr Potii, Olena Kachko, Serhii Kandii, and Yevhenii Kaptol. Determining the effect of a floating point on the falcon digital signature algorithm security. *Eastern-European Journal of Enterprise Technologies*, 2024. Key recovery attack using floating-point discrepancies in Falcon.
30. PQClean Development Team. Pqclean issue #522: Falcon signature inconsistency across arm platforms, 2023. Documentation of floating-point precision issues in Falcon implementations.
31. PQCRYPTO Consortium. Pqcrypto project deliverables, 2018.
32. Jai Vijayan. Attacker social-engineered backdoor code into XZ Utils. https://www.darkreading.com/attacks-breaches/attacker-social-engineered-backdoor-code-into-xz-utils, 2024.
33. Guillaume Wafo-Tapa, Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Etienne Marcatel. A practicable timing attack against hqc and its countermeasure. *Cryptology ePrint Archive*, 2019.
34. Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. *IEEE Transactions on Computers*, 71(9):2163–2176, 2021.

# A Hard-To-Find Bugs Definition

HFBs in cryptographic software are subtle errors that can compromise system security. These bugs are challenging to detect due to their low probability of occurrence and the specific conditions needed to trigger them—conditions that standard testing methods often fail to cover. Despite their rarity, HFBs can lead to serious vulnerabilities, such as incorrect cryptographic operations, side-channel vulnerabilities and private key leakage.

## A.1 Characterization of HFBs

They are characterized by the following:

1. Low Probability of Occurrence: Only under rare and specific conditions, such as unique input combinations, specific sequences of operations, or particular hardware or timing factors.
2. High Complexity and Subtlety: Often result from complex system interactions, subtle hardware behaviors, or intricate algorithmic details, like improper handling of edge cases.

3. Difficulty in Detection and Reproduction: Conventional testing methods struggle to identify these bugs, and without knowing the trick they are hard to reproduce.
4. Cross-Disciplinary Nature: Addressing HFBs often requires expertise across multiple areas, including software, hardware, and cryptographic theory, due to their involvement in various layers of the technology stack.

HFBs are not confined to cryptographic software; they represent a widespread challenge across the technology sector. For example, Bressana et al. (2020) discuss the difficulty of detecting subtle data plane bugs in network hardware using their Portable Test Architecture (PTA), which revealed hidden issues such as performance degradation under specific traffic conditions [10]. Although their research focuses on network devices, the challenges they highlight closely mirror those encountered in cryptography: both fields are plagued by bugs that only manifest under rare conditions, evading traditional testing methods. In cryptographic software—particularly in public key cryptography (PKC), where low-level implementation errors in modular and polynomial arithmetic, along with other subtle bugs, can lead to catastrophic security breaches—the need for a comprehensive, rigorous verification framework becomes even more pressing. This paper aims to develop such a framework to systematically identify and mitigate these elusive vulnerabilities.

### A.2 Impact and Exploitability

**A.2.1 Impact** The impact of bugs is assessed by the severity of the vulnerabilities they introduce, enabling developers to prioritize fixes based on potential damage. Key factors for evaluating impact include: (i) Exposure of sensitive data, (ii) System integrity compromise, (iii) Exploitability by attackers, and (iv) Long-term implications for system security. This classification is crucial for understanding cryptographic risks and underscores the need for expert knowledge in developing secure systems.

HFBs pose a significant security risk due to their potential exploitability by attackers. Their elusive nature makes them prime targets for adversaries who can craft precise inputs or operation sequences to exploit them. This can lead to severe consequences, such as cryptographic failures, unauthorized data access, or even private key leakage, compromising the integrity and confidentiality of secure systems.

**A.2.3 Exploitability** Given their low probability of occurrence under normal conditions, such bugs can remain undetected for extended periods. A notable example is the ROCA vulnerability (CVE-2017-15361), which affected RSA key generation in Infineon chips used by OpenSSL and other cryptographic libraries. The flaw in the key generation algorithm allowed attackers to factorize the public key and recover the private key. Introduced around 2012, it remained undetected until its discovery in October 2017 [20].

Moreover, there is an increasing risk of maliciously introduced HFBs, particularly in environments with frequent and large-scale code contributions, such as open-source projects. This risk is especially significant for malicious nation-state actors, who may seek to embed hidden backdoors in software widely regarded as secure, such as cryptographic libraries, to enable long-term espionage or control over critical systems. Attackers could intentionally introduce subtle HFBs during major commits or feature additions, which may evade detection during standard reviews. A recent example is the 2024 XZ Utils incident, where a malicious actor used social engineering to gain trust and insert a subtle backdoor into the software, potentially allowing unauthorized code execution on Debian machines [32]. These flaws are often masked within complex or large code changes, especially in low-level languages like C, C++, and Assembly, which are commonly used in cryptography implementations for performance reasons. Assembly, in particular, is still prevalent for critical optimizations, adding to the challenge of detecting such bugs.

### A.3 Hardware-Induced Errors

While our focus is on software HFBs, cryptographic failures can also stem from hardware-induced errors, which are often indistinguishable from software faults. Transient faults—such as bit flips in RAM or registers caused by cosmic radiation or electrical interference—can introduce security risks exploitable via fault injection attacks, as explored in comprehensive analyses of natural and induced faults [15]. Techniques like voltage glitching and electromagnetic fault injection can deliberately induce errors, potentially leaking sensitive data, such as private keys through faulty signature generation, as demonstrated in theoretical models of random hardware faults [8].

Though mitigating hardware-induced errors is a crucial research area, it is beyond the scope of this study. Addressing these risks requires error detection mechanisms, hardware-level protections, and resilient cryptographic implementations.

## B Countermeasures for Hard-to-Find Bugs in PQC

Mitigating HFBs in Post-Quantum Cryptography implementations requires a defense-in-depth strategy that transcends simple conformance testing. The countermeasures detailed in this section address vulnerabilities at the algorithmic, microarchitectural, and compiler levels. These strategies are essential because, as established in this work, the mathematical novelty of PQC schemes introduces HFB profiles distinct from classical cryptography. Effective hardening depends not on the assumed security of the mathematical primitives alone, but on engineering practices that eliminate observable, secret-dependent variations in implementation behavior. This section provides a clear engineering roadmap for achieving robust and secure PQC software.

### B.1 Systematic Hardening for Lattice-Based Cryptography

**B.1.1 Side-Channel Protection** The primary goal of side-channel protection is to eliminate any correlation between secret data and observable non-functional properties of the implementation, such as its execution time, power consumption, or memory access patterns.

A core principle of side-channel hardening is to ensure all arithmetic operations execute in constant time. Secret-dependent branching and variable-latency machine instructions are major sources of timing leakage. For example, integer division instructions like `idiv` on x86 architectures have input-dependent latencies. As demonstrated by the KyberSlash attacks, this specific vulnerability can be exploited to recover the private keys of Kyber. To prevent this, such operations must be replaced with constant-time reduction algorithms, such as Montgomery reduction or a branch-free implementation of Barrett reduction, which use a fixed sequence of multiplications and shifts.

Another critical vulnerability arises from conditional branches within core algorithms. In Number Theoretic Transform (NTT) operations, a naive modular reduction like `if (t >= q) t -= q;` creates a timing vulnerability that leaks information about secret coefficients. This must be replaced with a branch-free equivalent that uses bitwise masking to achieve the same result in a constant number of cycles.

To thwart attacks based on memory access patterns or power analysis, the relationship between secret data and its physical representation must be broken. This can be achieved through blinded memory access techniques. One such technique is arithmetic masking, where polynomial coefficients are combined with fresh random values before processing. The computation is performed on this masked data, and the mask is removed only at the end. Another technique is to randomize the execution order of independent operations, such as the sequence of NTT butterfly calculations, which disrupts an attacker's ability to correlate power traces with specific computations.

Samplers that use rejection loops are another source of leakage, as the number of iterations can depend on secret data. To prevent this, discrete Gaussian samplers should be implemented using constant-time algorithms. Methods based on precomputed Cumulative Distribution Tables (CDT) can be made secure by ensuring they execute with a fixed number of iterations, thereby making their timing independent of the values being sampled.

**B.1.2 Oracle Attack Prevention** Oracle attacks exploit logical flaws where an implementation's response reveals information about the correctness of an internal computation. This is often achieved by measuring timing variations caused by early-exit logic in verification routines.

To prevent such attacks, all verification steps must execute in constant time. The decapsulation logic, for instance, must not terminate prematurely based on secret-dependent checks. A loop checking the validity of polynomial coefficients must not use a `break` statement upon finding an invalid coefficient, as this creates a clear timing difference. The correct approach is to accumulate failure flags using

bitwise operations over the entire set of coefficients and perform a single check only after the loop has completed.

Furthermore, an implementation's observable response must be uniform across all failure conditions. This involves returning a generic error message regardless of the specific internal check that failed. Normalizing the timing of rejection responses, potentially by adding a random delay, also helps prevent an attacker from distinguishing different error pathways and gaining information about the secret data.

**B.1.3 Mitigating Floating-Point Vulnerabilities in Falcon** A primary issue is precision divergence. Floating-point operations are not fully standardized across different hardware architectures, such as x86 and ARM, which can lead to minute, non-deterministic differences in computation. It has been shown that these divergences can be collected and exploited as an oracle to recover a Falcon secret key. One mitigation is to replace the floating-point logic entirely with deterministic, integer-only sampling methods. A complementary approach is to incorporate rigorous cross-platform validation into testing pipelines, ensuring that signatures generated for identical inputs are bit-for-bit identical across all supported architectures.

Another vulnerability is mantissa leakage. The Hamming weight of the mantissas of floating-point values can be leaked through physical side channels like electromagnetic emissions. This information can be used in attacks to recover the FFT coefficients, which ultimately leads to key recovery. To mitigate this threat, implementers can use fixed-point arithmetic instead of floating-point, or apply masking schemes to the FFT coefficients to decorrelate the mantissa values from the underlying secrets.

### B.2 Systematic Hardening for Code-Based Cryptography

A key area of concern is decoder hardening against reaction oracles. Iterative decoders may have iteration counts or memory access patterns that depend on the secret error vector, creating an observable reaction that leaks information. To mitigate this, one must enforce a fixed-time decapsulation wrapper that ensures a constant amount of work is performed, or design decoders with a provably fixed iteration count. The correct implementation of a Fujisaki–Okamoto (FO) style transform can also mask decryption failures, but this is only effective if the re-encryption path is also fully protected from side channels.

Sampler protection is critical. Samplers used for generating constant-weight vectors can leak information through variable-time rejection loops or data-dependent shuffles. Implementations must adopt constant-iteration sampling algorithms or use a fixed-iteration wrapper around rejection-based samplers. Any shuffling operations must be implemented in a data-independent manner.

The finite-field arithmetic in code-based schemes, while simpler than in lattices, can still be a source of leakage. Table-based implementations are particularly vulnerable to cache-timing attacks and should be replaced with bitsliced

or purely arithmetic implementations. It is also important to ensure that any polynomial reduction is performed using constant-time algorithms.

Finally, the use of structured codes, such as quasi-cyclic codes, introduces the risk of algebraic weaknesses. While these codes reduce key sizes, they can create vulnerabilities. Implementations should therefore include tests to detect and filter any weak keys and should use conservative parameter choices to limit the effectiveness of structure-specific attacks.

### B.3 Compiler-Level and Microarchitectural Protections

Even a securely written implementation can be undermined by compiler optimizations or underlying hardware behaviors.

At the hardware level, microarchitectural hardening is necessary. This includes inserting serialization instructions, such as `lfence` on x86, to prevent out-of-order execution from creating timing leaks. Where feasible, disabling dynamic voltage and frequency scaling (DVFS) during cryptographic operations can ensure a more stable timing baseline, making anomalies easier to detect.

At the build level, developers must enforce strict code generation constraints. This involves using compiler flags that ensure predictable behavior, such as `-fwrapv` for defined integer overflow. For security-critical functions, it may be necessary to use lower optimization levels or specific function attributes to prevent the compiler from making transformations that inadvertently introduce vulnerabilities, such as reintroducing a division instruction where a constant-time reduction was intended.

Lastly, secure memory management is crucial. Sensitive data such as private keys and intermediate states must be securely wiped from memory after use, typically by XORing the data with randomness. Using hardware memory protection features, like guard pages via `mprotect`, can also help prevent accidental reads or writes outside of intended buffers, mitigating a large class of common programming errors.