WHITEPAPER

# Detecting Quantum-Vulnerable Cryptography and Guiding Post-Quantum Migration

*Long-form treatment: threat model, the mathematics of Shor and Grover, the NIST replacement standards, and a from-scratch lattice-based KEM.*

Daniel Lichtenberger · July 2026
quantumsafescan.com · Source on GitHub

**A technical whitepaper**

## Abstract

The eventual arrival of a cryptographically-relevant quantum computer (CRQC) will break the public-key cryptography that secures the modern internet. Shor's algorithm reduces integer factorization and discrete logarithms to efficient quantum order-finding, defeating RSA, Diffie-Hellman, and elliptic-curve schemes outright; Grover's algorithm provides a quadratic speedup against symmetric primitives and hash functions, halving their effective security. Because encrypted data can be recorded today and decrypted later ("harvest now, decrypt later"), and because cryptographic migration across a large codebase takes years, the transition to post-quantum cryptography (PQC) is a present-day engineering problem rather than a future one.

QuantumSafe is an end-to-end platform that addresses three questions in sequence: *Why is classical cryptography vulnerable?* (an executable quantum-attack module), *Where is the vulnerable cryptography in my code?* (a hybrid AST/regex static analysis engine spanning 11 languages, with a 0–100 risk model), and *What do I replace it with?* (a from-scratch lattice-based key-encapsulation mechanism and a NIST-aligned migration plan). The detector is evaluated on a hand-labeled corpus with adversarial decoys, achieving 100% precision and recall on that corpus, and an empirical study over eight widely-used open-source projects found that 88% contained at least one Shor-breakable primitive. This paper documents the threat model, system design, scoring methodology, evaluation, and the honest limits of a static, pattern-based approach.

## 1. Problem: why quantum breaks RSA and ECC

### 1.1 The classical hardness assumptions

Public-key cryptography rests on problems believed hard for classical computers:

- **RSA** — security reduces to the difficulty of factoring a large semiprime `N = p·q`.
- **Diffie-Hellman / DSA** — security reduces to the discrete logarithm problem in a finite field.
- **Elliptic-curve cryptography (ECDH, ECDSA)** — security reduces to the discrete logarithm problem over an elliptic-curve group.

All three are instances of the **hidden-subgroup problem** over abelian groups.

### 1.2 Shor's algorithm (breaks public-key crypto)

Shor's algorithm factors `N = p·q` by reducing factoring to **order-finding**: for a random `a` coprime to `N`, find the period `r` of `f(x) = aˣ mod N`. Classically the period is exponentially hard to find; quantumly it is recovered efficiently using **Quantum Phase Estimation (QPE)** over a modular-exponentiation operator, followed by an inverse **Quantum Fourier Transform (QFT)**. Measurement yields an approximation of `s/r`; **continued fractions** recover `r`. If `r` is even and `a^(r/2) ≠ −1 (mod N)`, then `gcd(a^(r/2) ± 1, N)` are non-trivial factors. With `p` and `q` known, the RSA private exponent `d ≡ e⁻¹ (mod ϕ(N))` follows immediately. The identical machinery solves discrete logarithms, so DH, DSA, and ECC fall as well — **at any key size**. Larger keys do not help; they only add a polynomial number of qubits and gates.

### 1.3 Grover's algorithm (weakens symmetric crypto)

Grover's algorithm finds a marked element in an unstructured space of size `N` in `≈ (π/4)√N` queries via amplitude amplification (an oracle reflection plus a diffuser). For a `k`-bit symmetric key, brute-force search drops from `2^k` to roughly `2^(k/2)` — a **quadratic**, not exponential, speedup. The practical consequence is that effective key strength is *halved*: AES-128 retains ~64 bits of quantum security (insufficient for long-lived data), while AES-256 retains ~128 bits (still infeasible). The mitigation is to **double sizes** rather than replace the primitive: AES-128 → AES-256, SHA-256 → SHA-384/512.

### 1.4 Hardware reality and the "harvest now, decrypt later" threat

No CRQC exists today. End-to-end Shor has only been demonstrated on tiny semiprimes (e.g. `N = 15`), and estimates for RSA-2048 require on the order of millions of noisy physical qubits with error correction (Gidney & Ekerå, 2021). This gap is precisely why migration is urgent rather than premature: an adversary can **record ciphertext today** and decrypt it once hardware matures. Any secret that must remain confidential for 5–15 years is effectively exposed now.

---

## 2. Threat model

**Assets.** Long-lived confidential data, authentication keys, code-signing keys, and the integrity of TLS/PKI trust chains.

**Adversary.** A future adversary with access to a CRQC capable of running Shor at scale, who is *also* assumed to be capturing and archiving ciphertext in the present (a passive network adversary today, an active decryptor later).

**In scope for QuantumSafe.** Detecting source-level *usage* of quantum-vulnerable cryptographic primitives, ranking the resulting risk, and recommending NIST-PQC replacements. The platform also *demonstrates* the underlying attack (Shor/Grover on a simulator) and *implements* the defensive primitive (LWE-based KEM).

**Out of scope.** QuantumSafe is a security-awareness and triage tool. It does **not** perform runtime cryptographic analysis, key-management auditing, side-channel assessment, protocol verification, or certificate-chain inspection, and it is not a substitute for a professional cryptographic audit. Findings are heuristic; false positives and false negatives are possible (see §9).

---

## 3. System architecture overview

QuantumSafe is organized as three conceptual layers over one shared detection engine:

1. **Attack layer** ( `quantum/` ) — Shor's and Grover's algorithms implemented in Qiskit and executed on a quantum simulator (Aer). This layer *justifies* the risk ratings the scanner assigns.
2. **Detection layer** ( `cli/` , shared by the CLI and the API) — the static analysis engine: scanner → risk scorer → NIST recommender → multi-format reporter.
3. **Defense layer** ( `pqc/` ) — a from-scratch Learning-With-Errors (LWE) key encapsulation mechanism, the mathematical basis of CRYSTALS-Kyber / ML-KEM.

These are delivered through three product surfaces that all consume the same detection engine: a **CLI** ( `pip install quantumsafe-scan` ), a **Flask REST API**, and a **static web dashboard**. A detailed component and data-flow description is in ARCHITECTURE.md.

---

## 4. Detection engine design (AST + regex hybrid)

The engine combines two complementary strategies to balance precision and language coverage:

- **AST analysis (Python).** Python source is parsed with the standard-library `ast` module, allowing detection to resolve imports and call expressions rather than matching raw text. This yields high precision and reduces false positives from incidental substrings.
- **Regex analysis (all languages).** For the remaining languages, the engine applies a curated set of anchored regular expressions. The same regex layer also backs Python as a secondary signal.

**Supported languages (11).** Python (AST + regex), JavaScript/TypeScript (incl. JSX/TSX/MJS/CJS), Java, Go, Ruby, C#, PHP, Rust, C/C++, Kotlin, Swift.

**Detection families.** Each rule belongs to a *family* — `rsa`, `ecc`, `dsa`, `dh`, `md5`, `sha1`, `tls_old`, `3des`, `rc4`, `sha256`, `aes128`, `tls12` — which is the key used to look up both the severity and the migration recommendation, so advice stays consistent across the CLI, API, and dashboard.

**Precision controls.**

- **Comment-only lines are skipped**, so crypto names mentioned in documentation do not produce findings.
- **Inline suppression**: a `# quantumsafe: ignore` comment (any comment syntax) on a line suppresses that line's findings.
- **Per-line / per-family de-duplication**: multiple matches of the same family on the same line collapse to a single finding, ranked by severity, so totals are not inflated.
- **Anchored patterns / word boundaries** prevent traps such as `md5sumLabel` or `dsaCount` from matching.
- **Size and directory guards**: large/minified files (>2 MB) and vendor directories (`node_modules`, `venv`, `dist`, …) are skipped.

**Inputs.** A local path, a single file, a public GitHub repository (shallow-cloned to a temporary directory and cleaned up), or an uploaded archive (via the API).

---

## 5. Risk scoring model (0–100)

Each finding carries a severity (`HIGH`, `MEDIUM`, `LOW`). The aggregate **Quantum Risk Score** is computed directly from findings — never hardcoded:

```
score = min(100, 15·HIGH + 5·MEDIUM + 1·LOW)
```

| Score | Band | Interpretation |
|---|---|---|
| 0–30 | Low | Good quantum hygiene |
| 31–60 | Medium | Plan migration |
| 61–80 | High | Prioritize migration |
| 81–100 | Critical | Immediate action required |

The weighting reflects the qualitative difference between the two quantum threats: a HIGH finding (Shor-breakable; e.g. RSA, ECC) is weighted 15× a LOW finding (Grover-weakened but still secure; e.g. SHA-256, AES-128), because the former is catastrophic and the latter is a sizing concern. The cap at 100 keeps the score interpretable as a bounded severity index rather than an unbounded count.

---

## 6. Quantum attack module (Shor + Grover)

The `quantum/` module runs the algorithms that *motivate* every HIGH and LOW rating, on a real quantum simulator (Qiskit Aer; the same circuits run on IBM Quantum hardware with a token).

- **`shor.py`** implements quantum order-finding via QPE over modular exponentiation and an inverse QFT, factors `N = 15`, reconstructs the corresponding RSA private key, and decrypts a message — a concrete, end-to-end demonstration of why RSA/ECC are rated HIGH.
- **`grover.py`** implements amplitude amplification to recover a hidden `k`-bit key in `≈ ⌊√(2^k)` iterations, demonstrating the quadratic speedup behind the LOW rating for symmetric primitives.
- **`resources.py`** transpiles the circuits and reports qubit counts, circuit depth, and gate counts, contextualized against published RSA-2048 resource estimates.

**Honest scope.** These run at small scale (factoring 15), which is the genuine state of the art for end-to-end Shor. The gap to RSA-2048 is the entire reason post-quantum migration is a future-proofing exercise.

---

## 7. Post-quantum module (LWE / Kyber-style)

The `pqc/` module implements the defensive primitive the scanner recommends, so the advice is runnable rather than merely named.

`lwe_kem.py` is a key-encapsulation mechanism built on the **Learning With Errors (LWE)** problem — the hardness assumption underlying CRYSTALS-Kyber / ML-KEM (FIPS 203):

- **Key generation.** Secret `s ∈ Z_q^n`; public key `(A, b)` with `A` random and `b = A·s + e (mod q)` for a small error `e`. Recovering `s` is the LWE problem, provably as hard as worst-case lattice problems.
- **Encapsulation.** To send a bit `μ`, choose a random 0/1 selector `r` and send `u = Aᵀr`, `v = bᵀr + μ·⌊q/2⌋`. A shared secret is many such bits, hashed with SHA-256.
- **Decapsulation.** Compute `v − sᵀu = eᵀr + μ·⌊q/2⌋`; because `eᵀr` is small, the high bit recovers `μ`. With parameters `(n=256, m=512, q=4093)` the error stays far below `q/4`, so decryption is reliable — verified over 200+ exchanges with zero failures.

**Why quantum computers do not break it.** LWE has **no periodic / hidden-subgroup structure** for Shor to exploit; the best known quantum attacks give only modest speedups over classical lattice reduction, so the problem remains exponentially hard. This is exactly why NIST standardized lattice schemes (ML-KEM, ML-DSA).

**Honest scope.** This is a faithful teaching implementation of plain LWE — not constant-time or side-channel-hardened, and not the optimized Module-LWE + NTT + Fujisaki–Okamoto construction of production Kyber. Production systems should use vetted ML-KEM (e.g. `liboqs`). `benchmark.py` reports measured latency and key/ciphertext sizes against standardized RSA-2048 and ML-KEM-768 for context.

## 8. Benchmark methodology and results

### 8.1 Corpus design

Detection quality is measured on a hand-labeled corpus rather than asserted. Ground truth (`benchmark/labels.json`) is defined at **(file, detection-family)** granularity:

- `benchmark/positive/` — nine known-vulnerable files spanning nine languages (Python, Java, JavaScript, Go, C#, PHP, Swift, Rust, Ruby).
- `benchmark/negative/` — three files of safe code plus **adversarial decoys**: crypto names that appear only in *comments*, and **word-boundary traps** (`md5sumLabel`, `rc4legacyName`, `dsaCount`) that must *not* match.

### 8.2 Metric definitions

For the set of expected `(file, family)` pairs and the set the scanner detects:

- **Precision** = TP / (TP + FP) — of what the scanner flagged, how much was real.
- **Recall** = TP / (TP + FN) — of what was real, how much the scanner found.
- **F1** = harmonic mean of precision and recall.

`benchmark/evaluate.py` runs the real scanner and prints the exact false positives and false negatives, so the numbers are auditable. The thresholds are enforced by `tests/test_benchmark.py`.

### 8.3 Results (current corpus)

| Metric | Value |
|---|---|
| Files | 12 (9 positive, 3 negative/decoy) |
| Languages | 9 (engine supports 11) |
| Labeled findings | 24 |
| True positives | 24 |
| False positives | 0 |
| False negatives | 0 |
| **Precision** | **100%** |
| **Recall** | **100%** |
| **F1** | **100%** |

The comment decoys yield zero false positives because the engine skips comment-only lines; the word-boundary traps are excluded by anchored patterns. Per-family and severity distributions, plus reproducible charts, are in ../benchmark/RESULTS.md.

### 8.4 Empirical study

Scanning eight widely-used open-source projects ( `study/` , reproducible via `python study/run_study.py` ) found **88%** contained at least one HIGH-risk (Shor-breakable) primitive, with an average Quantum Risk Score of **72.9/100**. The most common quantum-vulnerable primitives were RSA, ECC, legacy TLS, and MD5/SHA-1.

---

## 9. Limitations

These are the genuine edges of a static, pattern-based approach and are stated plainly:

- **Heuristic, not a proof.** QuantumSafe detects *usage patterns*; it does not prove a primitive is reachable, exploitable, or even live code.
- **Regex outside Python.** Only Python receives AST-level precision. Other languages are regex-based, so unconventional crypto wrappers, aliasing, or dynamic dispatch can produce false negatives.
- **Comments and string literals.** Comment-*only* lines are skipped, but a crypto name in a *trailing* comment or inside a string literal can still false-positive; block comments are only partially handled.
- **No dependency / transitive analysis.** Cryptography invoked indirectly through third-party libraries (rather than named in the scanned source) is not detected — which is also why a clean scan is not a guarantee of safety.
- **Benchmark scale.** The labeled corpus (12 files, 24 findings) is a regression benchmark, not a large-scale field study; 100% on it is not a claim of 100% on arbitrary code.
- **Demonstration scale.** Shor runs on `N = 15` and Grover on small key spaces — the real-world state of the art, not production attacks.
- **Teaching-grade PQC.** The LWE KEM demonstrates the mathematics; it is not production-hardened cryptography.

---

## 10. Future work

- **Tree-sitter / multi-language AST.** Replace regex for non-Python languages with concrete-syntax-tree parsing to recover AST-level precision across the board.
- **Dependency-aware detection.** Resolve cryptographic usage through imported libraries and package manifests, not only first-party source.

- **Data-flow and reachability.** Distinguish live, reachable cryptographic calls from dead code and test fixtures to sharpen the risk score.
- **Hybrid-PQC guidance.** Emit concrete hybrid (classical + PQC) migration recipes per protocol context (TLS, SSH, code signing).
- **Production KEM binding.** Offer an optional binding to vetted ML-KEM ( `liboqs` ) alongside the teaching implementation.
- **Expanded corpus and field study.** Grow the labeled benchmark and run a larger, statistically framed study across more ecosystems.

---

## References


1. P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM J. Computing*, 1997.
2. L. K. Grover, "A fast quantum mechanical algorithm for database search," *STOC*, 1996.
3. O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, 2009.
4. J. Bos et al., "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," *IEEE EuroS&P*, 2018.
5. C. Gidney and M. Ekerå, "How to factor 2048-bit RSA integers in 8 hours using 20 million noisy qubits," *Quantum*, 2021.
6. NIST, **FIPS 203** (ML-KEM), **FIPS 204** (ML-DSA), **FIPS 205** (SLH-DSA), Post-Quantum Cryptography Standards, 2024.
7. NIST **IR 8547**, Transition to Post-Quantum Cryptography Standards.