

QuantumSafe: Detecting Quantum-Vulnerable Cryptography

What the scanner is, how the hybrid AST/regex engine works, and how well it measures against a labeled nine-language benchmark.

Daniel Lichtenberger · July 2026

quantumsafescan.com · [Source on GitHub](#)

A concise, current technical account of what QuantumSafe is, how it works, and how well it works. For the long-form treatment (full threat model, algorithm math, and references) see [WHITEPAPER.md](#); for the module/data-flow breakdown see [ARCHITECTURE.md](#).

1. The problem in one paragraph

A cryptographically-relevant quantum computer breaks the public-key cryptography that secures almost everything online. **Shor's algorithm** reduces integer factorization and discrete logarithms to efficient quantum order-finding, defeating RSA, Diffie–Hellman, and elliptic-curve schemes *at any key size*. **Grover's algorithm** gives a quadratic speedup against symmetric primitives, halving their effective strength. No such machine exists yet — but an adversary can **record ciphertext today and decrypt it later** ("harvest now, decrypt later"), and migrating cryptography across a large codebase takes years. That is why NIST finalized the post-quantum standards (FIPS 203/204/205) in 2024, and why the transition is a present-day engineering problem. QuantumSafe is a tool for doing that transition: **find** the vulnerable crypto, **understand** why it's vulnerable, and **adopt** the replacement.

2. What it is: three layers over one engine

QuantumSafe covers the full arc of the post-quantum problem rather than only naming it:

Layer	Module	What it does
Attack	<code>quantum/</code>	Shor's and Grover's algorithms in Qiskit , run on a simulator. Shor factors <code>N = 15</code> via quantum phase estimation + inverse QFT and reconstructs a toy RSA key — an <i>educational</i> demonstration of why RSA/ECC are rated HIGH, not a break of real key sizes. Grover recovers a hidden k -bit key in $\approx \sqrt{2^k}$ steps, motivating the LOW rating for symmetric crypto.
Detection	<code>cli/</code>	A hybrid AST + regex static-analysis engine across 11 languages, with cross-language usage-awareness, dependency + lockfile scanning, reachability ranking, and call-site remediation. Scores risk 0–100 and emits a NIST-aligned migration plan. This is the core of the product.
Fix	<code>ppc/</code>	A from-scratch Learning-With-Errors (LWE) key-encapsulation mechanism — the math behind CRYSTALS-Kyber / ML-KEM (FIPS 203) — so the recommended replacement is runnable, not just named.

The detection engine is consumed by three surfaces that all share **one** code path: a free **CLI** (`pip install quantumsafe-scan`), a **Flask REST API**, and a **web dashboard**. Because there is a single engine, advice never drifts between surfaces.

3. The detection engine

Hybrid AST + regex. Python source is parsed with the standard-library `ast` module, so detection resolves real imports and call expressions instead of matching raw text. The remaining ten languages (JavaScript/TypeScript, Java, Go, Ruby, C#, PHP, Rust, C/C++, Kotlin, Swift) use a curated set of anchored regular expressions. Every finding is tagged with a **family** (`rsa`, `ecc`, `dsa`, `dh`, `md5`, `sha1`, `tls_old`, `3des`, `rc4`, `sha256`, `aes128`, `tls12`); the family is the key used to look up both severity and the migration recommendation.

Usage-awareness is the core design decision. In security tooling, *precision is the product* — a scanner that cries wolf gets turned off. So the engine deliberately distinguishes *code that uses* a primitive from *text that merely names it*, and — as of 0.2.0 — it does so in **every** supported language, not just Python:

- **Python** masks string, docstring, and comment content precisely with the `tokenize` module before the regex pass, while the AST engine runs on the original source (so `hashlib.new("md5")` is still caught).
- **Every other language** is handled by a small **lexer-style state machine** that blanks the content of comments (line, trailing, and multi-line block) and string literals. Go is import-aware — it masks strings *except* on `import` lines, because Go's import paths (`"crypto/dsa"`) are the detection signal.
- Because masking would hide algorithm names that legitimately live *inside* a string argument (`MessageDigest.getInstance("SHA-1")`, `createCipheriv("aes-128-gcm", ...)`), a **string-argument recovery pass** re-detects those, but only when the callee is a known crypto factory and the argument normalizes to a real algorithm — the cross-language analogue of the Python AST engine.
- Anchored patterns / word boundaries defuse traps like `md5sumLabel` or `dsaCount`; an inline `# quantumsafe: ignore` suppresses a line; matches are de-duplicated per `(file, line, family)` so one line can't inflate the score.

Dependency + lockfile scanning. Most real exposure arrives through third-party libraries, so the scanner also parses dependency manifests **and lockfiles** across pip/npm/go/maven/gem (`requirements.txt` , `pyproject.toml` , `package.json` , `go.mod` , `pom.xml` , `Gemfile` , plus `package-lock.json` , `yarn.lock` , `poetry.lock` , `Pipfile.lock` , `Gemfile.lock` , `go.sum`). Known quantum-vulnerable crypto packages are flagged with a **purl** and **direct/transitive scope**, and rendered in the CBOM as `library` components linked to the crypto assets they provide. These findings are marked `origin="dependency"` , `confidence="medium"` — capability exposure, not proof a call site is exercised.

Reachability ranking. A finding in dead code or an example is not the same risk as one on a live path, so a Python call-graph pass labels each source finding `reachable` (module-level, or inside a referenced/decorated/entrypoint function), `test/example` (under a test or docs path), or `unreferenced` (a top-level function whose name never appears anywhere else — dead code, conservatively defined). Reports rank exploitable findings first. The signal only ever *demotes* in ranking; it never drops a finding.

Call-site remediation. Each finding carries a concrete fix, not just a family-level recommendation: a before/after for drop-ins (MD5/SHA-1 → SHA-256, 3DES/RC4 → AES-256-GCM, AES-128 → AES-256, TLS 1.0/1.1/1.2 → 1.3) or, for asymmetric families where no like-for-like swap exists, the target PQC scheme plus a language-appropriate library (liboqs, BouncyCastle PQC, Cloudflare CIRCL) and an honest note that it is a design change. Fixes appear in JSON, HTML, and the SARIF rule help text.

Optional interprocedural taint analysis (`--taint`). A direct scan flags `hashlib.md5(...)` but is blind to a wrapper that hides it:

```
def _legacy_digest(data):          # wraps MD5
    return hashlib.md5(data).hexdigest()
def sign_request(payload):          # re-exposes it one hop up
    return _legacy_digest(payload)
token = sign_request(body)          # the real blast radius - no "md5" in sight
```

The taint pass builds a **call graph** and propagates taint to a **fixpoint**: any function that *transitively* reaches a trusted (AST-detected) primitive becomes tainted, and its call sites are reported as **indirect** findings with the wrapper chain attached. This works **across files**, not just within one: `cli/callgraph.py` builds a *whole-program* call graph by resolving imports (absolute, relative, and unique-suffix) to qualified `module.func` symbols, so a wrapper defined in `crypto/legacy.py` and called from `api/handlers.py` — with no crypto keyword at the call site — is still caught, and its provenance names the resolving module. A call edge is only created when it resolves to an indexed function, so an unrelated same-named function can't create a phantom edge. The pass is opt-in (`--taint`) and strictly *additive* (it never emits where the direct scan already fired), so it extends coverage without ever regressing precision.

Risk score (computed from findings, never hardcoded):

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

HIGH (Shor-breakable: RSA/ECC/DSA/DH, MD5/SHA-1) is weighted 15× LOW (Grover-weakened but still secure: SHA-256, AES-128) because the two threats are qualitatively different — one is catastrophic, the other a sizing concern. Bands: 0–30 Low · 31–60 Medium · 61–80 High · 81–100 Critical.

Six export formats: terminal table, JSON, **SARIF** (validated against the OASIS 2.1.0 JSON Schema in the test suite, so GitHub code-scanning ingests it), **CycloneDX CBOM** (validated against the 1.6 crypto structure), HTML, and an embeddable SVG badge.

4. Does it actually work? (measured, not asserted)

Labeled benchmark — precision under adversarial pressure. A hand-labeled corpus of **18 files across 9 languages (26 findings)** includes adversarial decoys: crypto names inside comments, docstrings, log strings, exception messages, trailing and block comments, plus word-boundary traps. `evaluate.py` runs the scanner **twice** so the effect of usage-awareness is *measured against a baseline*, not claimed:

Configuration	TP	FP	FN	Precision	Recall	F1
Naive line-regex baseline	26	27	0	49.1%	100%	65.8%
QuantumSafe (usage-aware)	26	0	0	100%	100%	100%

Usage-awareness removes **27 false positives** — crypto keywords in docstrings, logs, exception strings, and comments — **without losing a single true positive**, and now across *all* languages (13 of the 27 come from Java/JS/Go decoys). Thresholds are enforced by `tests/test_benchmark.py`, so a regression fails CI.

Seeded benchmark — recall with ground truth by construction. A small labeled corpus measures precision well but says little about *recall* at scale, so `seeded.py` runs a **mutation benchmark**: it embeds real quantum-vulnerable API calls (many idiomatic variants per family, 7 languages) into host files and asserts each is detected, then embeds the *same algorithm name* only in a comment and a string and asserts it is not. Because every case is a real call, a miss is unambiguously a false negative.

- **50 seeded positive cases → 100% recall;**
- **50 negative mutations → 0 false positives (100% mutation precision).**

Enforced by `tests/test_seeded.py`. This building of the harness also *found and fixed* a real bug — 7 Go false positives where crypto names in ordinary Go strings leaked through, resolved by import-aware masking — which is the whole reason to measure yourself.

Head-to-head vs. a generic scanner. `comparison.py` scores each installed tool against the same 26 labeled (file, family) pairs. The point it makes is structural:

Tool	Caught / 26	Recall	Families found
QuantumSafe	26	100%	all 11 families, 9 languages
Bandit 1.8	2	8%	md5, sha1 only

Bandit is an excellent *classical* Python linter doing its job — it flags MD5/SHA-1 and says nothing about RSA/ECC/DSA/DH because those are not classical vulnerabilities. Its 2/26 is the thesis in one number: **quantum readiness needs a quantum-aware tool**. Full methodology and a capability matrix vs. Semgrep/SonarQube/CBOM tools in [benchmark/COMPARISON.md](#).

Real-world scale — discoveries, not a labeled corpus. `realworld.py` pulls the latest sdist of **37 of the most-downloaded PyPI packages** straight from the PyPI JSON API and scans them (nothing is built or executed):

- **32 of 37 (86%)** contained ≥ 1 finding; **11,275** Python files analyzed;
- **5,609** total findings, **4,236 HIGH-risk** (run of 2026-10-28, scanner 0.3.1);
- every finding is a concrete `file:line` an auditor can open — e.g. Diffie–Hellman in `twisted/conch/ssh/transport.py`, MD5 in `requests/auth.py` (HTTP Digest).

The paramiko case shows why usage-awareness matters at scale: on paramiko 5.0.0 (July 2026 run) a naive line-regex reports **451 raw matches**, most of them the same algorithms repeated in docstrings and SSH protocol-name strings; focusing on real code brings it to **110** findings — a $\sim 4\times$ noise reduction. (*Honest caveat: some of the 37 packages are cryptography libraries that implement RSA/ECC on purpose, so their large counts are expected; the telling signal is the application/infrastructure code — `django`, `botocore`, `requests`, `scrapy` — still reaching for quantum-vulnerable primitives.*)

Empirical study. Across 8 widely-used projects (`study/`), **88%** contained a HIGH-risk (Shor-breakable) usage, at an average Quantum Risk Score of **61.4/100**.

5. The post-quantum fix, implemented

`pqc/lwe_kem.py` is a key-encapsulation mechanism on the **LWE** problem. Key generation publishes $(A, b = A \cdot s + e \bmod q)$ for a small error e ; recovering the secret s is LWE, provably as hard as worst-case lattice problems. With $(n=256, m=512, q=4093)$ the accumulated error stays well below $q/4$, so decapsulation is reliable — **verified over 200+ exchanges with zero failures**. A quantum computer can't shortcut it because LWE has **no periodic / hidden-subgroup structure** for Shor to exploit — which is exactly why NIST standardized lattice schemes. (*Honest scope: this is a faithful teaching implementation of plain LWE, not the constant-time Module-LWE + NTT + Fujisaki–Okamoto construction of production Kyber; real systems should use vetted ML-KEM such as `liboqs`.*)

6. Beyond the codebase: live endpoints and deadlines

Two things a code scan cannot answer: what a deployed service actually negotiates, and whether the migration finishes in time.

The TLS handshake (cli/tlsscan.py). A website's quantum exposure is not in its source — it is in two brief public-key steps of the handshake. The **key exchange** is the urgent one: Shor recovers the session secret, and because an adversary can record traffic now and decrypt it later, data sent today is exposed *retroactively*. The **certificate signature** is also Shor-breakable, but only to forge *future* certificates, and those rotate every ~90 days. Symmetric cryptography is not the problem at all: Grover square-roots a brute-force search, so AES-128 retains ~64-bit security and AES-256 is comfortable.

Reading the key exchange needed a hand-built ClientHello. TLS 1.2 named the key exchange in the cipher suite (`ECDHE_RSA_WITH_AES_128_GCM_SHA256`); TLS 1.3 removed it, so every modern site reports only `TLS_AES_128_GCM_SHA256` — which says nothing. The group lives in the `key_share` extension, and Python's `ssl` cannot report it (`set_groups` and `group_used` need OpenSSL 3.5+ and Python 3.14; ML-KEM groups need OpenSSL 3.5 at all). So the scanner offers the hybrid groups in `supported_groups` but a `key_share` for X25519 only: a server preferring a hybrid cannot proceed from that share and must answer with a HelloRetryRequest naming the group it wants (RFC 8446 §4.1.3). No ML-KEM implementation and no new dependency are needed to ask the question. One handshake, 89–257 ms.

The whole certificate chain (cli/certchain.py). A chain is only as strong as its weakest signature, and the weak link is usually not the leaf: `cloudflare.com` serves an ECDSA P-256 leaf, but an intermediate above it is signed **SHA-256 with RSA**. Forging that signature mints a leaf for any name beneath it. The signature algorithm and public key are precisely the two fields `ssl.getpeercert()` omits, so rather than take on a `cryptography` dependency they are walked out of the DER with a minimal tag-length-value reader that parses structure and OIDs and verifies nothing — `ssl` has already validated the chain by then. A root's self-signature is shown but not scored: a root is trusted because it is in the client's trust store, so forging that signature gains an attacker nothing.

Mosca's inequality (cli/horizon.py). The risk score says *how much* vulnerable cryptography exists. It cannot say whether the migration lands in time, which is the question the threat model actually poses: one RSA key exchange protecting data that must stay secret for fifty years is worse than forty MD5 checksums.

```
X  how long the data must stay confidential
Y  how long the migration takes
Z  how long until a quantum computer can break it

X + Y > Z    ->    data sent today is already exposed
```

What the tool refuses to claim matters as much as what it reports. **Z is unknowable, so there is no default for it** — omitting it yields `no_verdict` rather than a guess. What is always computed is the **break-even year**: the latest a quantum computer could arrive while still leaving time to migrate, which needs no estimate of Z and is a property of the code and the data rather than a forecast. Y is an engineering estimate, so it is a range, counted per cryptographic *family* rather than per finding (the design decision is paid once; call sites scale sub-linearly), and all six coefficients are emitted under `horizon.assumptions` so a reader can reject a specific number instead of the whole figure.

The origins a page depends on (`cli/origins.py`). A site can negotiate a hybrid on its own domain and still pull scripts, fonts and analytics from a dozen other origins over classical key exchange — and the author chose every one of them, so the weakest is part of the page's exposure. `--origins` fetches the entry page, extracts the hosts its sub-resources reference, and runs the same scan against each. Measured on `github.com` : 2 of 5 third-party origins post-quantum, 3 classical. On `quantumsafescan.com` : 2 of 2 ready.

The result is one aggregate finding rather than one per origin — a dozen separate HIGHS for other people's servers would bury the findings a reader can act on — and it is MEDIUM rather than HIGH when the page's own key exchange is fine, because a third-party server is not the author's to fix. The action there is to drop the dependency or press the vendor, which is a different call to action from repairing your own endpoint.

A page is attacker-controlled input, so the SSRF guard is applied to every *discovered* origin rather than once to the entry URL and then trusted: anyone can publish markup whose script tags point at `169.254.169.254` , and a sweep that followed what it found would be an SSRF with extra steps. The page read, the origin count and each handshake are all bounded, so one hostile page cannot turn a scan into an unbounded crawl.

Scanning someone else's server is bounded to a single TLS handshake — what any browser does on every visit. It is not port or vulnerability scanning. The target must resolve *entirely* to public addresses on port 443/8443, and the connection is made to the address that was checked rather than re-resolved, which closes the DNS-rebinding window; private ranges, loopback, link-local (cloud instance metadata at `169.254.169.254`), carrier-grade NAT and the IPv4-mapped IPv6 forms of all of them are refused.

7. Engineering

One shared engine behind CLI + API + dashboard; **559 automated tests**; CI; Docker and `docker-compose` ; a reusable **GitHub Action** (`action.yml`) that fails a build over a risk threshold; multi-service deploy configs; and a published PyPI package (`pip install quantumsafe-scan` , v0.3.0). The benchmark, real-world study, seeded recall harness, comparison, and every chart are **reproducible from scripts** and generated from live scanner output, not hand-drawn.

8. Honest limitations

- **Heuristic, not a proof.** It detects *usage patterns*; reachability is a conservative ranking signal, not a proof that a primitive is exploitable or live.
- **AST precision is Python-only.** The other ten languages use usage-aware masking plus string-argument recovery — a large improvement over naive regex — but still lack full AST resolution, so an algorithm name passed through an *unrecognized* wrapper can be a false negative.

- **Dependency findings are capability-level.** A flagged library ships quantum-vulnerable crypto; that is not proof a given call path is exercised (hence `confidence="medium"`). Detection uses a curated catalog of known crypto packages, so an obscure library may be missed.
- **Reachability is intra-project and Python-focused.** Cross-file import resolution is not modeled; non-Python findings get the path-based signal only.
- **Benchmark scale.** 18 files / 26 labeled findings is a *precision* regression set; the seeded harness adds 50 constructed recall cases. Neither is a claim of 100% on arbitrary real-world code.
- **Demonstration scale.** Shor runs on `N = 15`; the LWE KEM is teaching-grade.
- **The TLS scan reports what a server *selects*, not everything it supports.** The result is the group chosen from the set offered — the same answer a browser offering that set would get — so a server that supports a hybrid but ranks it lower reads as classical. Findings say "negotiated", not "supports", for exactly that reason.
- **One endpoint per handshake, and only the entry page is fetched.** `--origins` sweeps the hosts a page's sub-resources reference, but it does not follow redirects, probe other ports, or recurse into those origins' own dependencies, and it reads markup only — a resource pulled later by JavaScript is invisible to it.
- **Migration time is an estimate, not a measurement.** The horizon's coefficients are assumptions about how long cryptographic work takes, and they are printed alongside the result precisely because they should be replaced with figures from a real migration. The break-even year is the only number there that rests on no guess about the future.
- **A diff cannot prove a fix.** A finding that disappeared may have been deleted, excluded or renamed — a rename reads as one fixed plus one new. The diff reports counts and locations, not conclusions, and says so in its own output.

9. What's next

Tree-sitter AST parsing to bring non-Python languages up to AST-level precision; import-resolving reachability (the taint pass is already cross-file); a larger hand-labeled real-world corpus to complement the seeded recall benchmark; per-protocol hybrid (classical + PQC) migration recipes; and an optional binding to production ML-KEM alongside the teaching implementation.

Every figure in this report is reproducible: `python benchmark/evaluate.py` (precision), `python benchmark/seeded.py` (recall), `python benchmark/comparison.py` (vs. other tools), `python benchmark/realworld.py` (the 37-package study), and `python study/run_study.py` (the 8-project study).