

PQC-Readiness of Open-Source Software: An Empirical Study

Lukas Mehl 

Washburn University, Topeka, KS, USA

Abstract

Migrating open-source systems to post-quantum cryptography requires knowing where classical public-key primitives still appear in code, yet much empirical work has targeted dependency graphs or *misuse* of APIs rather than primitive-level, multi-language usage at scale. This paper describes a static analyzer that scans Python, Java, and Go repositories using syntax-aware parsing, classifies cryptographic calls and imports as post-quantum-vulnerable, quantum-safe, or PQC-ready, and aggregates findings for ecosystem-level measurement.

Applied to a stratified sample of about fourteen and a half thousand GitHub repositories, vulnerable primitives are found to be widespread at the level of explicit calls and imports, with strongly language-dependent rates shaped in part by how transport and certificate stacks are treated in static analysis. Evidence of PQC-ready APIs is exceedingly rare by comparison, underscoring a stark adoption gap relative to migration goals. Methodological limitations are stated so the baseline can be interpreted, reproduced, and extended.

2012 ACM Subject Classification Security and privacy → Cryptography; Software and its engineering → Software creation and management; General and reference → Empirical studies

Keywords and phrases post-quantum cryptography, static analysis, empirical study, open-source software, cryptographic primitives

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.53

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material *Software (Scanner source, aggregate dataset, and evaluation reference set):*
<https://doi.org/10.5281/zenodo.19634475>

Acknowledgements The author thanks Joseph Kendall-Morwick for comments on an earlier draft of this manuscript.

1 Introduction

Much of today’s deployed *public-key* cryptography (RSA, (EC)DSA, finite-field and elliptic-curve Diffie–Hellman, and related schemes) rests on the hardness of integer factorization or discrete logarithms. Cryptographically relevant quantum computers are widely expected to break those assumptions (e.g., using Shor’s algorithms), so such primitives are called *post-quantum-vulnerable*: confidential data or signatures that rely on them could later be forged or decrypted by a sufficiently capable adversary. Symmetric ciphers and cryptographic hashes behave differently under generic quantum search (typically quadratic speedups); they are not overturned in the same way and are often retained with conservative parameters during migration. The practical risk is therefore uneven: long-lived public-key material is especially sensitive, including TLS and software-signing keys and the “harvest now, decrypt later” scenario in which ciphertext recorded today is broken once quantum hardware matures.

Standardization bodies have begun to define the post-quantum migration path. NIST published the first PQC standards in 2024 [8, 9, 10]: **ML-KEM** (a key-encapsulation mechanism, i.e. a replacement for RSA/Diffie–Hellman key exchange, formerly known as Kyber), **ML-DSA** (a digital-signature scheme, formerly Dilithium), and **SLH-DSA** (a



© Lukas Mehl;

licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 53; pp. 53:1–53:14



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

hash-based signature scheme, formerly SPHINCS+). Together with **NTRU**, an earlier lattice-based encryption scheme still used in some libraries, these form what this paper calls *PQC-ready* primitives – algorithms designed to remain secure against quantum attackers. Authorities such as the BSI recommend early transition and hybrid deployment (running a classical and a post-quantum primitive side by side during migration) [2]. For vendors and open-source maintainers, a central engineering question is *where* vulnerable public-key primitives still appear in code so that migration can be prioritized and progress tracked.

Why this needs source-level, multi-language measurement. Existing empirical work on cryptography in software has focused on dependency- and package-level analysis [18] or on *misuse* of cryptographic APIs in single languages [26, 17, 7]. Dependency manifests (e.g. a Maven or PyPI entry for a crypto library) say that a project *could* use RSA or AES somewhere; they do not say whether it actually calls a key-generation routine, an import used only in a test file, or a PQC binding. Misuse detectors, symmetrically, assume the algorithm choice itself (e.g. RSA) is acceptable and only check whether it is invoked correctly. Neither view answers the question this paper asks: *which cryptographic algorithms – classical or post-quantum – are open-source projects actually calling in their source code, at what prevalence, and how far is that from the post-quantum migration target?* A gap remains for **multi-language, primitive-level** measurement of *which* algorithms are used (classical vs. post-quantum) rather than merely which libraries are declared or whether an API is called correctly. Without such a baseline, it is unclear how far the open-source ecosystem is from the migration targets those standards envision, and maintainers and funders have no empirical basis for prioritizing migration effort.

This paper addresses that gap by asking three concrete research questions:

- **RQ1 (Prevalence).** How common is source-level usage of post-quantum-vulnerable cryptographic primitives in popular, actively maintained open-source repositories?
- **RQ2 (Language dependence).** Does that prevalence differ systematically across programming languages/ecosystems (Python, Java, Go), and if so, is the difference a genuine usage pattern or an artifact of what static analysis can observe per language (Section 6)?
- **RQ3 (Adoption gap).** How common is source-level evidence of PQC-ready primitives (e.g. ML-KEM/Kyber, ML-DSA/Dilithium, SLH-DSA/SPHINCS+) today, i.e. how far has the migration NIST and the BSI call for actually progressed in practice?

To answer these questions, a static analysis tool is presented that scans Python, Java, and Go repositories for cryptographic API usage and imports, classifies each finding as post-quantum-vulnerable, quantum-safe, or PQC-ready, and aggregates results for empirical study. **Syntax-aware parsing** means the tool reads each file into an abstract syntax tree (AST) – the same structural representation a compiler builds – rather than searching raw text with regular expressions; this lets it tell an actual function call (`RSA.generate(...)`) apart from a mention in a comment, a string, or an unrelated identifier that happens to contain the word “rsa”. Section 3 gives the concrete classification mapping and states the caveats that follow from this design (e.g., TLS/X.509 import heuristics, dynamic calls). The implementation and methodology are documented together with limitations so that aggregate statistics can be interpreted and reproduced; headline empirical outcomes answering RQ1–RQ3 are reported in Section 5.

Contributions. This paper contributes:

1. **A multi-language scanner** for Python, Java, and Go that classifies cryptographic API calls and imports as post-quantum-vulnerable, quantum-safe, or PQC-ready, released as open-source software (Section 3).

2. **A documented sampling and aggregation methodology** for repository-scale measurement, including an explicit test-vs.-production heuristic and a precision/recall check against a curated reference set (Sections 3–4).
3. **An empirical baseline on 14,508 stratified GitHub repositories** quantifying the prevalence of vulnerable primitives, its variation by language, and the PQC-ready adoption gap (Section 5).
4. **A transparent discussion of what the measurement can and cannot support**, including a sensitivity analysis isolating how much of the language difference in RQ2 is attributable to a single import-level heuristic rather than genuine usage differences (Section 6).

Long-term objectives and intended impact. This submission reports an initial, reproducible baseline (tool, evaluation, large-sample measurement) rather than a finished longitudinal study; that scope matches the Emerging Results track’s purpose of sharing early results for community feedback. The longer-term research agenda this baseline is meant to seed includes: (1) *longitudinal rescans* of the same repository population to track migration progress over time as ML-KEM/ML-DSA library support matures; (2) *broader language and library coverage* (JavaScript/TypeScript, C/C++, additional crypto libraries per language) to reduce the detection gaps discussed in Section 3.9; and (3) *refining the TLS/X.509 signal* (Section 6) so that transport-layer usage can be distinguished from direct key-material handling. For the open-source software engineering community, the intended contribution is a reusable, extensible measurement instrument and an open dataset that other researchers and maintainer communities can rescan, audit, and build on – rather than a one-off number – so that PQC migration progress becomes something the community can track empirically instead of anecdotally.

The remainder of the paper is structured as follows. Section 2 reviews related work, including empirical repository-mining methodology; Section 3 describes detection, classification, and sampling; Section 4 reports a reference-set spot check; Section 5 presents empirical results answering RQ1–RQ3; Section 6 discusses implications, threats to validity, and closes.

2 Related Work

2.1 Post-Quantum Standards and Migration

NIST’s 2024 standards (ML-KEM, ML-DSA, SLH-DSA) [8, 9, 10] and hybrid transition guidance from authorities such as the BSI [2] define migration targets; this work measures *where* source code still relies on quantum-vulnerable public-key primitives rather than prescribing how to migrate.

2.2 Empirical Studies on Cryptographic Usage

Prior work has studied the use of cryptography in software at different levels. Dependency- and package-level analyses track which crypto libraries or versions are declared in manifests (e.g. Maven, PyPI, Go modules) and have been used to study vulnerability propagation and supply-chain risk [18]. Such studies are valuable for supply-chain and vulnerability management but do not reveal *which primitives* are actually invoked in source code, e.g. whether a project uses RSA, ECDSA, or both, or whether it has already adopted PQC APIs. Source-code-based studies exist for single languages or specific libraries; for example, Wickert et al. analyze cryptographic API *misuse* in 895 Python projects from GitHub [26]. A gap remains for **multi-language, primitive-level** measurement of classical vs. post-quantum *usage* (rather than misuse) at scale, which the present scanner and dataset aim to provide.

2.3 Static Analysis for Cryptography

Static analysis tools for cryptography have largely focused on **misuse detection**: finding incorrect or insecure *usage* of cryptographic APIs (e.g. weak modes, hardcoded keys, improper randomness). CryptoGuard [17] and CogniCrypt/CrySL [7] are prominent examples that use AST- or pattern-based analysis to flag such issues. They assume that the *algorithm* in use is an accepted choice; their goal is to ensure it is used correctly.

The contribution addresses a **complementary but orthogonal** question. Existing tools such as CryptoGuard and CogniCrypt address incorrect usage of cryptographic primitives; the focus here is the *algorithm-level* quantum vulnerability of open-source software, independent of implementation correctness. A correctly implemented RSA-2048 is a non-issue for prior tools but a critical migration target under NIST’s post-quantum standards. Thus, this work does not replace misuse detectors; it answers a different, policy-relevant question: how far is the ecosystem from the post-quantum migration path, and where are classical primitives still in use?

Technically, the approach shares with prior work the use of syntax-aware parsing to reduce false positives from comments and string literals and to distinguish direct API calls from imports: rather than matching raw text, the scanner walks each file’s abstract syntax tree – for Python via the standard-library `ast` module, for Java and Go via `tree-sitter` [24], an incremental parser-generator that produces a concrete syntax tree independent of a full language toolchain – and inspects import statements and call-expression nodes directly (see Section 3 for the exact node patterns per language). CryptoGuard and CogniCrypt/CrySL apply the same AST-based principle to misuse detection [17, 7]; this work reuses it for a different classification task (primitive identity rather than usage correctness). Limitations such as dynamic or reflective invocation (e.g. algorithm names from configuration) apply equally [17, 7] and are discussed in the Methodology and Limitations sections.

2.4 Empirical Methods for Repository Mining

Because the study’s evidence comes from mining a large, unlabeled population of GitHub repositories, its design choices are checked here against established guidance from the mining-software-repositories (MSR) literature rather than treating GitHub as a neutral, representative corpus. Combining a large-scale analysis of GitHub metadata with a manual inspection of 434 repositories, Kalliamvakou et al. document nine “perils” of using GitHub data for research, including that most projects are inactive, that a large share of repositories are not used for software development at all (experimentation, storage, personal websites), and that roughly two-thirds are personal, effectively single-committer projects rather than collaborative software efforts [6]. This motivates treating “a GitHub repository” as a noisy, non-representative unit rather than a *de facto* software project. The sampling design (Section 3) is consistent with that concern: it stratifies by star-count range rather than taking a flat top-*N* sample, and applies a minimum popularity threshold intended to bias the sample away from personal or abandoned repositories. This is a partial mitigation, not one derived from Kalliamvakou et al. or a solved problem: Section 6 reports the threshold explicitly and notes that popularity alone does not guarantee the properties they describe. Vidoni’s systematic literature review of MSR studies finds that most do not follow or report an explicit, systematic protocol for repository selection and data extraction, and recommends making such choices explicit so they can be scrutinized [25]; the per-repository JSON output, the aggregation CSV, and the CLI report pipeline (Section 3), together with the Limitations and Threats to Validity discussion (Sections 3.9 and 6), are structured in that spirit. Finally,

Gold and Krinke argue that MSR research is, in an important sense, human-subjects research: the repositories studied here reflect real developers' work and choices, and those developers did not consent to having their code analyzed for this specific purpose [4]. Consistent with the ethical caution they call for, the scanner only inspects already-public repositories, records no personal data about individual contributors, and the paper's per-repository discussion (Section 5) reports only externally observable facts about code (which cryptographic APIs are imported or called) rather than characterizing the people who wrote it.

3 Methodology

3.1 Overview

The study uses a static analysis tool that scans source code repositories for uses of cryptographic primitives and classifies each finding as **post-quantum-vulnerable**, **quantum-safe**, or **PQC-ready**. The tool targets three languages (Python, Java, and Go) and produces per-repository summaries plus an aggregated dataset suitable for empirical analysis of PQC adoption in open-source software.

3.2 Detection Approach

Syntax-aware parsing is used rather than plain text or regex search. This reduces false positives from comments, string literals, and documentation, and allows direct API usage (e.g. key generation) to be distinguished from mere imports. For **Python**, the standard library **ast** module is used; for **Java** and **Go**, **tree-sitter** [24] with language-specific grammars. The tool does not execute code; therefore dynamic or reflective invocations (e.g. algorithm names passed via configuration) may be missed and are a known limitation.

3.3 Per-Language Detection

- **Python.** Each `.py` file is parsed for (1) imports of 11 known crypto libraries (`cryptography`, `PyCryptodome`, `ssl`, `hashlib`, `hmac`, `rsa`, `ecdsa`, `oqs`, etc.) and (2) 14 API call patterns (e.g. `RSA.generate`, `ECC.generate`, `load_pem_private_key`). Each finding is associated with a primitive name and a confidence level (high for direct calls, medium for imports).
- **Java.** `.java` files are parsed with `tree-sitter`; the detector collects (1) imports from `javax.crypto`, `java.security`, or `org.bouncycastle` (including `org.bouncycastle.pqc`), and (2) invocations of `getInstance(...)` with a string argument. The implementation recognizes 24+ algorithm strings (classical: RSA, EC, ECDH, DSA, AES, SHA-256; PQC: Kyber, Dilithium, ML-KEM, ML-DSA, SPHINCS+) and 6 PQC import-path patterns.
- **Go.** `.go` files are parsed with `tree-sitter`; the detector collects (1) imports from 10 paths (`crypto/rsa`, `crypto/ecdsa`, `crypto/elliptic`, `crypto/dh`, `crypto/tls`, `crypto/x509`, `golang.org/x/crypto`, `liboqs-go`) and (2) 11 call signatures (e.g. `rsa.GenerateKey`, `ecdsa.GenerateKey`, `elliptic.P256`). Imports of `crypto/tls` and `crypto/x509` are classified as post-quantum-vulnerable (medium confidence), since TLS and X.509 usage in practice implies classical RSA/ECDSA certificates; this is an *import-level* signal, not a claim about the actual certificates in use (see Limitations and Discussion).

3.4 Classification of Primitives

Section 1 motivated the split between public-key schemes whose standard hardness assumptions fall to quantum algorithms (hence migration targets) and symmetric-style primitives that remain comparatively robust under known quantum attacks and are grouped here as *quantum-safe* for measurement purposes. The categories below instantiate that split for concrete API names and library paths; edge cases (e.g., elliptic-curve key agreement used ephemeral in TLS) are flagged where they affect interpretation.

Each detected primitive name is mapped to one of four categories:

- **Post-quantum-vulnerable:** RSA, ECDSA, ECDH, DH, DSA, ElGamal, classic curves (e.g. P-256), and **X25519/Ed25519** (counted as vulnerable for transparent reporting; ephemeral TLS use is discussed in Section 3.9) [2].
- **Quantum-safe:** Symmetric and hash primitives (AES, SHA-2, SHA-3, ChaCha20, HMAC, PBKDF2) that are not broken by known quantum algorithms in the same way as public-key schemes.
- **PQC-ready:** Algorithms standardized or in wide use for post-quantum cryptography: KyberML-KEM, DilithiumML-DSA, SPHINCS+, NTRU, and usage of the Open Quantum Safe library (liboqs/oqs) [20].
- **Unknown:** Any primitive name not in the mapping table; such names are not counted as vulnerable or PQC-ready in the aggregate metrics.

The mapping table is extensible and documented in the tool’s source (see repository).

3.5 Test vs. Production Code

Repositories often contain test code or third-party code (e.g. under `test/`, `tests/`, `vendor/`, `node_modules`, `third_party`). Each finding is tagged with a boolean `is_test_file` if the file path (relative to the repo root) contains any of these segments. Code under `vendor/` is thus counted as non-production. The paper reports both: (1) vulnerability metrics over **all** findings, and (2) metrics restricted to **production** code only, so that the share of vulnerable findings in test code can be clearly communicated. Many cryptographic libraries ship their own tests; if a repository is primarily such a library, a large fraction of vulnerable findings may fall in test paths. That pattern is treated as a valid empirical outcome; test detection is **path-based only** (no semantic analysis), as noted in the Limitations subsection.

3.6 Dataset and Sampling

The dataset uses **stratified sampling** over Python, Java, and Go (roughly equal shares), and **star-range strata** (e.g. $\geq 10k$ down to lower ranges) to diversify beyond the GitHub Search API’s per-query result cap, rather than a flat top- N query that would only capture already-famous projects. A fixed repo list is built once (`build-repo-list` or `snapshot; docs/SCALING.md`) for reproducibility. As Section 2.4 discusses, popularity-based filtering is a mitigation against, not a guarantee against, the kind of unrepresentative repositories documented in the MSR literature [6]; the sample retained in the final 14,508-repository run has no repository below **92 stars** (median 946, maximum 408,536 – see Section 5 for the exact per-language breakdown), which excludes toy or single-commit repositories but still skews toward already-popular projects, a limitation carried through to Section 6.

3.7 Output and Aggregation

The scanner writes a JSON result per repository (list of findings, summary counts, vulnerability score) and appends one row per repo to an aggregated CSV. Detectors emit raw primitive names (e.g. `RSA`, `ECDSA`, `crypto/rsa`, `rsa.GenerateKey`, or language-specific spellings). For aggregation, these labels are **normalized** to a **canonical key** per logical primitive (e.g. all RSA variants and Go `crypto/rsa` / `rsa.GenerateKey` are merged into one entry), so that the top-primitives tables are not split by spelling or language. A CLI report command produces paper-ready statistics: fraction of repos with ≥ 1 vulnerable primitive, fraction with ≥ 1 PQC-ready primitive, breakdown by language, top primitives (aggregated), and criticality (vulnerable in production vs. test).

3.8 Evaluation of the Scanner (Precision and Recall)

Before large-scale use, repo-level labels in a CSV (cited README or official docs, not a line-by-line audit) are compared to scanner output on `has_vulnerable` and `has_pqc_ready` as a deterministic **sanity check** (Section 4); broader threats are in Section 3.9.

3.9 Limitations

The following limitations are stated explicitly so that reviewers and readers can judge the generalizability of the results:

- **Dynamic and reflective usage:** The scanner relies on static analysis only. Algorithm names or library choices that are determined at runtime (e.g. from configuration, reflection, or string concatenation) are not detected. This may lead to **false negatives** in projects that use such patterns.
- **Unsupported libraries and patterns:** Only a fixed set of libraries and API patterns per language is recognized. Use of other crypto libraries or custom wrappers is not captured and may be underreported.
- **Languages omitted:** JavaScript/TypeScript and other ecosystems are out of scope; extending the tree-sitter approach would require new pattern libraries comparable to adding a fourth language.
- **Path-based test heuristic:** The distinction between “test” and “production” code is based on path heuristics (e.g. `test/`, `tests/`, `vendor/`). Misclassification (e.g. production code under a test path, or tests outside these paths) can affect the criticality breakdown; both “all” and “production-only” metrics are reported to reduce reliance on this heuristic.
- **TLS and X.509 (Go):** Imports of `crypto/tls` or `crypto/x509` count as vulnerable (import-level signal; actual certificates are not inspected). Top-primitive tables and Section 6 discuss interpreting rates with and without these signals.

These limitations are also reflected in the Discussion and Threats to Validity where appropriate.

4 Evaluation

Section 3.8 sketched a **reference-set spot check** ($n = 19$). Labels tie `has_vulnerable/has_pqc_ready` to **citable docs** (README, official API text), not a line-by-line audit; entries skew to mainstream crypto libraries and PQC bindings, so the set is **transparent but coverage-biased** and does not estimate ecosystem-wide error rates. Nineteen repositories are enough to sanity-check that the detectors fire on well-known crypto

■ **Table 1** Agreement with the curated reference set ($n = 19$); spot check, not ecosystem-wide validation.

Task	Precision	Recall	F1
has_vulnerable	1.00	0.93	0.97
has_pqc_ready	1.00	0.88	0.93

and PQC libraries and to surface concrete failure modes (below), but they are far too few to estimate precision/recall with any statistical confidence across the full 14,508-repository sample, or across less mainstream libraries and wrapper patterns that this reference set does not include. Growing this reference set – more repositories per language, more non-mainstream and wrapper libraries, and ideally multiple independent annotators with an agreement statistic – is the most direct way to strengthen this evaluation and is left as immediate follow-up work rather than attempted within this Emerging Results submission.

Labels group into **vulnerable-only** exemplars (`pyca/cryptography` [15], `golang/crypto` [21], `paramiko/paramiko` [14], `jpadilla/pyjwt` [16], `golang/go` [22], `square/go-jose` [19], and others), **PQC-ready-only** liboqs bindings [20, 13, 12, 11], and **mixed** stacks (`bcgit/bc-java` [1], `cloudflare/circl` [3], `tink-crypto/tink` [5, 23]); the CSV in the replication package lists all 19 with cited sources. For Tink, PQC-ready labels depend on which subtree is scanned [5]. The pipeline clones each repo, runs the scanner, and compares predictions to the CSV; Table 1 gives precision/recall as **regression-style** checks, not calibrated ecosystem-wide error.

Replication. Table 1 reflects a full run of `evaluation/run_evaluation.py` on the CSV with the scanner version documented in the replication package; verification output ships with that package (see Data Availability).

Recall is 0.93 (`has_vulnerable`) and 0.88 (`has_pqc_ready`); precision is 1.0 on both tasks. The two false negatives illustrate coverage limits rather than label errors. `square/go-jose` implements RSA/ECDSA JOSE operations via `github.com/square/go-jose`, which is outside the Go detector’s fixed stdlib-oriented import and call patterns [19]. `tink-crypto/tink` is a multi-module monorepo: PQC-ready APIs appear in language-specific trees (e.g. `tink-go/v2` ML-DSA/SLH-DSA paths), while the ground-truth label follows roadmap-level PQC support [5]; a root-level scan can miss sparse submodule imports.

5 Results

Headline outcomes. Across a stratified sample of 14,508 repositories (sampling procedure below), three patterns dominate: (1) **19.38 %** of repositories exhibit at least one post-quantum-vulnerable primitive at the level of explicit API calls and imports, with strong variation by language (**Go 40.7 %**, Java 12.6 %, Python 4.8 %), driven in part by import-level TLS/X.509 coverage in Go; (2) a pronounced **adoption gap**, with only **0.08 %** (11 of 14,508) showing any PQC-ready usage; and (3) vulnerable findings split almost evenly between production and test/example paths, suggesting that classical primitives remain the default across both.

5.1 Sample

A stratified sample of $N = 14,508$ repositories was scanned (Python, Java, and Go in roughly equal shares: 4,986 / 4,635 / 4,884 per GitHub primary language metadata, with three repos lacking language metadata), drawn by star-range strata and a minimum star count to ensure

■ **Table 2** Repositories and vulnerability by language.

Language	Repos	With vuln.	% w. vuln.	PQC-ready	% PQC-ready
Go	4,884	1,987	40.7%	5	0.10 %
Java	4,635	584	12.6%	5	0.11 %
Python	4,986	239	4.8%	1	0.02 %

diversity and exclude toy projects. Concretely, every repository in the final sample has **at least 92 GitHub stars** (median 946; maximum 408,536), which is the empirical minimum enforced by the star-range strata used to build the list rather than a single global cutoff applied afterward. The aggregate data and exact stratification parameters are documented in the replication package ([docs/SCALING.md](#)). The summary statistics reported below come from a full scan using the scanner’s `report` command; raw aggregates and per-repo outputs from large runs are supplied there rather than in a public VCS snapshot (see Data Availability).

5.2 Overall Vulnerability Rate

The *PQC vulnerability rate* is defined as the fraction of repositories that contain at least one finding classified as post-quantum-vulnerable (RSA, ECDSA, ECDH, classical curves, or, for Go, imports of `crypto/tls` or `crypto/x509`). In this sample, **19.38 %** of repositories (2,811 of 14,508) have at least one such finding, for a total of 79,802 vulnerable findings (and 3,948 quantum-safe findings). This rate is methodologically conservative in the sense of *detection*: the scanner only counts *explicit* API calls and imports; indirect use through frameworks or configuration is not detected. At the same time, import-level TLS/X.509 signals (especially in Go) can substantially increase the number of repositories flagged compared to patterns that omit those imports; interpretation is discussed in Section 6.

5.3 By Language

The vulnerability rate varies sharply by language (Table 2). With the updated detectors and aggregation pipeline, **Go** shows the largest share of repositories with at least one vulnerable finding (1,987 of 4,884, 40.7 %), followed by **Java** (584 of 4,635, 12.6 %) and **Python** (239 of 4,986, 4.8 %). The Go result is consistent with import-based classification of `crypto/tls` and `crypto/x509` (Section 3.9) and with those primitives dominating the top of the frequency table (Section 5, Top primitives). Java projects often use cryptographic infrastructure directly (e.g., Bouncy Castle, JCA `getInstance`) [17]; Python code frequently relies on higher-level libraries that abstract crypto and may not surface as direct primitive usage in the implemented patterns [26]. Repositories with PQC-ready findings are split across languages (Table 2), reflecting bindings and libraries in more than one ecosystem [20].

Note. Three repositories lack GitHub language metadata; they are omitted from Table 2 (one of them has a vulnerable finding).

Sensitivity: is Go actually less secure, or does it just import more visibly? Because `crypto/tls`/`crypto/x509` imports are counted as vulnerable at import level and are Go-specific labels in the mapping table (Section 3), part of Go’s lead in Table 2 could be

■ **Table 3** Vulnerability rate with vs. without `tls/x509`-labeled findings.

Language	% w. vuln. (all)	% w. vuln. (excl. <code>tls/x509</code>)
Go	40.7%	21.4%
Java	12.6%	7.0%
Python	4.8%	4.6%

an artifact of that single heuristic rather than a genuine difference in cryptographic usage. Table 3 recomputes the per-language vulnerability rate after dropping findings labeled exactly `tls` or `x509`. Go’s rate falls from 40.7% to **21.4%** – roughly half of its repositories with a vulnerable finding owe that finding solely to a `crypto/tls/crypto/x509` import – while Java falls from 12.6% to 7.0% and Python is almost unchanged (4.8% to 4.6%, since its `cryptography.x509` and `ssl.PROTOCOL_TLS*` labels are spelled differently and are not removed by this filter). Two things follow: (1) a non-trivial share of the Go vulnerability rate is indeed a transparency effect of one import-level heuristic, so the headline **40.7%** should not be read as “40.7% of Go repositories call classical key-generation APIs”; but (2) even after removing that heuristic, Go’s residual rate (21.4%) remains roughly three times Java’s and more than four times Python’s, so the language gap is not *purely* an artifact – Go repositories in this sample also more often contain direct calls such as `rsa.GenerateKey` or `ecdsa.GenerateKey` than their Java or Python counterparts. Both readings are reported so that neither the raw rate nor the sensitivity-adjusted rate is mistaken for the whole picture; Section 6 returns to the interpretive consequences.

5.4 PQC Adoption

The clearest result regarding **PQC adoption** is that only **0.08%** of repositories (11 of 14,508) exhibit at least one PQC-ready finding (Kyber, Dilithium, SPHINCS+, NTRU, or liboqs/oqs usage [20]). In total there are 947 PQC-ready findings across these repositories; the most frequent labels include NTRU, SPHINCS+, Dilithium, ML-DSA, ML-KEM, and Kyber (consistent with the aggregated primitive frequencies from the same reporting pipeline). This quantifies a clear gap between current open-source practice and the post-quantum migration target [8, 9, 10, 2] and constitutes a central finding of the study. The scanner also reports quantum-safe primitives (AES, SHA-2, ChaCha20, HMAC); with 3,948 findings they are omitted from the vulnerability rate but kept in the aggregate pipeline for context.

Who are the 11 adopters? Inspecting the 11 repositories individually (they are not part of the curated evaluation set) and cross-referencing each PQC-ready finding’s file path against the test/vendor path heuristic (Section 3) refines the headline number further. Two repositories owe their PQC-ready label entirely to a *vendored third-party dependency* rather than their own code: `GhostTroops/scan4all`’s 22 findings and `darkweak/souin`’s single finding both resolve to a bundled copy of Cloudflare’s CIRCL library under a `vendor/` path – the host project itself does not call these APIs. Several more show PQC-ready evidence concentrated in *test code for library features still under development*: `openjdk/jdk` (21 of 22 findings in tests, consistent with the JDK’s ML-KEM/ML-DSA support landing first as tested library internals) and `pyca/cryptography` (20 of 22 in tests) follow this pattern, and `apache/camel`’s dedicated `camel-pqc` component is mostly test code (33 of 41) with a small non-test core. The remaining repositories show PQC-ready primitives in *non-test, non-vendor code*: `bcgit/bc-java` (Bouncy Castle; 523 of 823 findings outside test paths, by far the

■ **Table 4** Top five post-quantum-vulnerable primitive labels (finding counts).

Label	Findings
<code>tls</code>	20,275
<code>x509</code>	18,603
<code>rsa</code>	13,681
<code>ecdsa</code>	11,201
<code>ec</code>	5,513

largest genuine contributor via its `org.bouncycastle.pqc` packages), `Keyfactor/ejbca-ce` (a certificate-authority product), `XTLS/Xray-core`, `XTLS/REALITY`, `ivpn/desktop-app`, and `i2p/i2p.i2p` – all networking, VPN, or anonymity tools with production-path post-quantum key-exchange or signature code. This breakdown (informal, derived from inspecting these 11 repositories rather than a systematic classification) suggests that even the already-tiny 0.08% adoption figure overstates *deployed* PQC usage: at least 2 of the 11 repositories show no adoption of their own, several more show it only in test/development code for not-yet-released features, and clearly production-path adoption today is effectively confined to cryptography-library internals (Bouncy Castle) and a handful of privacy/networking early adopters.

5.5 Criticality (Test vs. Production)

Path heuristics (`test/`, `tests/`, `vendor/`, etc.) tag each finding. Among vulnerable findings, **39,807** fall in test or example code and **39,995** in production paths (roughly even). Classical primitives therefore appear at scale in both settings; reporting production-only and all-findings metrics (as here) helps readers judge criticality.

5.6 Top Primitives

Table 4 lists the five most frequent vulnerable labels; `tls` and `x509` dominate, reflecting Go import-level `crypto/tls/crypto/x509` classification together with TLS/X.509 usage elsewhere. These are import-level signals where applicable: importing `crypto/tls` for HTTPS counts as vulnerable without inspecting certificates or cipher suites (Section 6).

6 Discussion

The near-even split of vulnerable findings across production and test paths shows classical primitives remain the default in both settings; when developers write tests or demos involving cryptography, they still reach for classical APIs because PQC alternatives are not yet part of the default toolkit.

Is Go actually less secure, or just more transparent to the scanner (RQ2)?

Table 3 (Section 5) gives a direct, data-backed answer rather than a purely qualitative caveat: roughly half of Go’s headline gap over Java and Python is a transparency effect of the `crypto/tls/crypto/x509` import heuristic (40.7% falls to 21.4% once those two labels are excluded), but the other half is not – Go’s residual rate remains about three times Java’s and more than four times Python’s even on a like-for-like comparison that excludes the heuristic entirely. A plausible reading is that Go’s standard library exposes

`crypto/rsa`, `crypto/ecdsa`, and `crypto/elliptic` as commonly-used, low-ceremony stdlib calls for everyday networking code, where equivalent Java code more often goes through a higher-level provider abstraction (`getInstance`) that this detector also tracks, and Python code more often goes through libraries this detector’s Python patterns do not cover as exhaustively [26]. In other words, *both* things are true: the scanner is more “transparent” to Go’s import style than to Python’s, *and* Go code in this sample does call classical key-generation and signing APIs directly more often. Consumers of the **19.38 %** repository-level rate and the Go-specific share in Table 2 should read both numbers in Table 3 together, not the 40.7 % figure in isolation (Section 3.9).

The headline statistics are a **readiness signal**, not a quantum timeline forecast: they inventory where classical APIs and TLS paths remain default and how rarely PQC-ready calls appear (**0.08 %** of repos, and rarer still once vendored dependencies and test-only usage are excluded, Section 5). That gap suggests **limited embodied PQC practice** at scale before policy, certificates, and library defaults converge, while *harvest now, decrypt later* still motivates long-lived confidentiality planning. Dependency upgrades alone seldom remove vulnerable signals without application-level algorithm changes.

Threats to validity

Construct validity: labels reflect static API/import patterns, not runtime cipher suites or certificate contents (especially for Go TLS/X.509); the sensitivity analysis in Table 3 bounds, rather than eliminates, this threat for the language-comparison claim (RQ2). **Internal validity:** the $n = 19$ reference set is a documentation-backed regression check, not a calibrated sample for ecosystem-wide precision; false negatives on wrapper libraries (Section 4) bound recall on curated exemplars only, and growing this set is planned follow-up work rather than a solved problem. **External validity:** results apply to popular GitHub repositories (all above 92 stars in this sample, Section 5) in Python, Java, and Go; popularity filtering only partially mitigates the risks documented in the MSR literature [6], and JavaScript/TypeScript and unstarred code are out of scope. Main claims rest on transparent rules, the large sample, and the sensitivity analysis for RQ2.

Contribution to the open-source software engineering community

Beyond the headline numbers, this work’s practical contribution is a reusable measurement instrument: the scanner, its per-language pattern tables, and the aggregation/report pipeline are released as open-source software so that maintainer communities and other researchers can point it at their own repositories rather than this paper’s specific sample. Because the classification mapping (Section 3) is an explicit, editable table rather than logic buried in the parser, extending it to a new library, algorithm, or language is a scoped engineering task, not a re-implementation. The intended impact is turning “is our project PQC-ready?” from a manual, ad-hoc audit into a repeatable measurement that maintainers, security teams, and standardization bodies can rerun over time.

For maintainers, low PQC adoption and pervasive classical usage suggest underused migration guidance [2, 8, 9, 10]. For researchers, the scanner and dataset support longitudinal rescans and extensions (more languages, dependency-level views [18]), building on the MSR guidance in Section 2.4 [6, 25, 4]. This complements misuse-centric studies [17, 7] by measuring algorithm-level distance to the post-quantum path at scale.

Data Availability Statement

Replication materials – scanner source code, the per-language detection patterns, the classification mapping table, the 14,508-repository aggregate CSV and per-repository raw JSON outputs, the evaluation reference set and its CSV of cited labels, and the scripts used to produce the tables in this paper – are archived on Zenodo at <https://doi.org/10.5281/zenodo.19634475>. Any material that cannot be shared (e.g. full clones of the scanned repositories, which remain available from GitHub directly) is noted in the replication package’s README.

The author used large language models for grammar and phrasing support during manuscript preparation; all claims, analyses, and citations were verified by the author.

References

- 1 Bouncy Castle. Big news from nist: Post-quantum cryptography in bouncy castle. <https://www.bouncycastle.org/resources/big-news-from-nist/>, 2024. NIST PQC support in Bouncy Castle Java (Kyber, Dilithium, SPHINCS+).
- 2 BSI. Migration to post-quantum cryptography. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf, 2021. German Federal Office for Information Security; English version; recommends hybrid schemes and early transition.
- 3 Cloudflare. Circl: Cryptographic library (go). <https://github.com/cloudflare/circl>; <https://pkg.go.dev/github.com/cloudflare/circl>. RSA, ECDSA, P-256 etc.; Kyber, Dilithium, ML-DSA, SLH-DSA.
- 4 Nicolas E. Gold and Jens Krinke. Ethics in the mining of software repositories. *Empirical Software Engineering*, 27(17), 2022. doi:10.1007/s10664-021-10057-7.
- 5 Google. Tink roadmap: Post-quantum cryptography. <https://developers.google.com/tink/roadmap>, 2025. ML-DSA and SLH-DSA supported in C++ and Go; Java in progress; Kyber/ML-KEM under development.
- 6 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014. ACM, 2014. doi:10.1145/2597073.2597074.
- 7 Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. In *ECOOP*, volume 109 of *LIPIcs*, pages 10:1–10:27, 2018. doi:10.4230/LIPIcs.ECOOP.2018.10.
- 8 NIST. FIPS 203: Module-lattice-based key-encapsulation mechanism standard. <https://csrc.nist.gov/pubs/fips/203/final>, 2024. ML-KEM. Finalized 13 August 2024.
- 9 NIST. FIPS 204: Module-lattice-based digital signature standard. <https://csrc.nist.gov/pubs/fips/204/final>, 2024. ML-DSA. Finalized 13 August 2024.
- 10 NIST. FIPS 205: Stateless hash-based digital signature standard. <https://csrc.nist.gov/pubs/fips/205/final>, 2024. SLH-DSA. Finalized 13 August 2024.
- 11 Open Quantum Safe. liboqs: C library for post-quantum cryptography. <https://github.com/open-quantum-safe/liboqs>; <https://openquantumsafe.org/liboqs/>. Kyber, Dilithium, SPHINCS+; PQC-only.
- 12 Open Quantum Safe. liboqs-go: Go bindings for liboqs. <https://github.com/open-quantum-safe/liboqs-go>. Kyber, Dilithium, SPHINCS+.
- 13 Open Quantum Safe. liboqs-java: Java bindings for liboqs. <https://github.com/open-quantum-safe/liboqs-java>. Kyber, Dilithium, SPHINCS+ via JNI.
- 14 Paramiko. Key handling — paramiko documentation. <https://docs.paramiko.org/en/stable/api/keys.html>. RSAKey, ECDSA key types; <https://github.com/paramiko/paramiko>.

- 15 PyCA. Cryptography: Python cryptographic recipes and primitives. <https://cryptography.io>. Documentation: RSA, ECDSA, and hazmat APIs.
- 16 PyJWT. Digital signature algorithms — pyjwt documentation. <https://pyjwt.readthedocs.io/en/stable/algorithms.html>. RS256 (RSA), ES256 (ECDSA); <https://github.com/jpadilla/pyjwt>.
- 17 Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. GitHub: <https://github.com/CryptoGuardOSS/cryptoguard>. doi:10.1145/3319535.3345659.
- 18 Daniel Setó-Rey, Carlos López-Nozal, and José Ignacio Santos-Martín. Vulnerability of software package repositories: Pypi, maven and npm. *ERCIM News*, 2024(139), 2024. URL: <https://ercim-news.ercim.eu/en139/special/vulnerability-of-software-package-repositories-pypi-maven-and-npm>.
- 19 Inc. Square. go-jose: JOSE standards (JWE, JWS, JWT) in go. <https://github.com/square/go-jose>. RSA and ECDSA key operations for signing/verification.
- 20 Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. Cryptology ePrint Archive, Paper 2016/1017, 2016. URL: <https://eprint.iacr.org/2016/1017>.
- 21 The Go Authors. golang.org/x/crypto: supplementary cryptography libraries. <https://pkg.go.dev/golang.org/x/crypto>. RSA, ECDSA, TLS, and related packages beyond standard library.
- 22 The Go Authors. Package crypto — go standard library. <https://pkg.go.dev/crypto/rsa>, <https://pkg.go.dev/crypto/ecdsa>, <https://pkg.go.dev/crypto/tls>. `crypto/rsa`, `crypto/ecdsa`, `crypto/tls` used throughout; <https://github.com/golang/go>.
- 23 Tink Cryptographic Library. Tink: Multi-language cryptography library. <https://github.com/tink-crypto/tink>. RSA signatures; PQC in language-specific subrepos (`tink-cc`, `tink-go`, `tink-java`, `tink-py`).
- 24 Tree-sitter. Tree-sitter: a parser generator tool and an incremental parsing library. <https://tree-sitter.github.io/tree-sitter/>. Used for Java and Go parsing in our scanner.
- 25 Melina Vidoni. A systematic process for mining software repositories: Results from a systematic literature review. *Information and Software Technology*, 144, 2022. doi:10.1016/j.infsof.2021.106791.
- 26 Anna-Katharina Wickert, Lars Baumgärtner, Florian Breitfelder, and Mira Mezini. Python crypto misuses in the wild. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3475716.3484195.