CT-KAT: A Multilayer Analysis Platform for Automated Screening of Constant-Time Risks in PQC C Implementations

Seung-Won Lee[1 [0009−0009−9583−1387]], Min-Seo Kim[1 [0009−0008−1979−5778]], Su-Min Jeong[1 [0009−0007−9643−4872]], and Hwa-Jeong Seo[1 [0000−0003−0069−9061]]

Department of Information Computer Engineering,
Hansung University, Seoul, 02876 Korea
dkajdfhd1@gmail.com, msqoo0212@gmail.com, jeong9sumin@gmail.com, hwajeong84@gmail.com

Abstract. Following the standardization of major post-quantum cryptography (PQC) algorithms, C implementations of ML-KEM, ML-DSA, and SLH-DSA have been rapidly deployed. However, known-answer tests(KATs) verify only functional correctness and do not establish the absence of timing leakage caused by secret-dependent branches, memory accesses, or variable-latency instructions. This paper presents CT-KAT, an integrated screening platform for assessing constant-time risks in PQC C implementations. CT-KAT is configured through a single YAML file. Using this configuration, it validates the build and KAT execution. It also automatically generates harnesses for the functions under analysis. It then performs several analyses within a unified pipeline. These include Valgrind/Memcheck-based structural checks, a ct-matrix over compiler and optimization-level combinations, and an assembly-level asm-scan. When configured, the pipeline also runs dudect-based timing tests. CT-KAT aggregates the results of these checks into a single verdict drawn from a nine-class taxonomy. CT-KAT does not treat a `PASS` result as proof of constant-time behavior. Instead, it applies a default-deny policy when assigning verdicts. It also provides supporting evidence to guide manual triage. When applied to a PQClean-based PQC corpus, CT-KAT produced four main classifications. The baseline ML-KEM target was classified as `robust`. The KyberSlash reproduction was flagged as a secret-dependent variable-latency risk, labeled `varlat-secret-risk`. The variable-time behavior of ML-DSA and SLH-DSA/SPHINCS$^+$ was accepted under the label `accepted-variable-time`. Falcon-512 was retained for further analysis and labeled `needs-analysis`. These results show that CT-KAT provides a screening framework that integrates signals from multiple analysis layers while making the limitations of each tool explicit.

Keywords: Post-quantum cryptography · Constant-time implementation · Timing side channels · KyberSlash · ML-KEM · Build sensitivity · Dynamic analysis

1 Introduction

Timing side-channel attacks have been recognized as a central threat to cryptographic implementations since Kocher's work in 1996 [9]. If execution time, cache-access patterns, or instruction latency depend on a secret key, an adversary may statistically exploit these dependencies to recover the key [2]. The standard countermeasure is constant-time (CT) programming. Under this programming discipline, secret values must not influence branch conditions. They must not determine memory addresses, including array indices. They must also not affect the operands of variable-latency instructions.

In 2024, NIST finalized standards for ML-KEM, ML-DSA, and SLH-DSA, the standardized form of SPHINCS$^+$ [13,12,14]. C implementations of these algorithms are being rapidly integrated into cryptographic libraries and embedded systems. However, a transition involving the large-scale deployment of new code also creates opportunities for new CT defects to be introduced. A prominent example is **KyberSlash** [3]. In affected Kyber/ML-KEM implementations, polynomial coefficients are used as operands in division operations. On certain compiler–platform combinations, these operations have input-dependent latency. This timing leakage originates from *instruction latency*, not from secret-dependent branches or memory accesses. Therefore, it can evade structural analysis tools that are specialized in tracking branches and addresses.

Assessing CT security in practice is challenging for several reasons. First, known-answer tests (KATs) determine only whether an implementation produces the correct outputs; they do not assess whether those outputs are computed without timing leakage. Second, existing CT analysis tools provide fragmented coverage. For example, ctgrind [10] detects secret-dependent branches and memory accesses, whereas dudect [16] tests for differences between execution-time distributions. Because each tool provides only a partial view, meaningful screening requires multiple tools to be used together. In practice, a test program, or harness, must often be written manually to invoke the target function for each tool. Analysts must then compare and reconcile tool outputs presented in different formats. This process is cumbersome and error-prone.

Third, a `PASS` result can easily be overinterpreted. Dynamic analysis inherently observes only the execution paths exercised during testing. Interpreting `PASS` as "proof of constant-time behavior" can therefore be misleading. Such an interpretation may cause practitioners to omit subsequent analyses, including masking assessment and power or electromagnetic (EM) leakage measurements. Finally, CT behavior is sensitive to the compiler and optimization settings. Consequently, a result obtained under a single build configuration may not represent the security of the binary that is ultimately deployed. To address these issues, we propose **CT-KAT**, an integrated platform for screening CT risks.

The contributions of this paper are as follows.

1. From a single YAML specification, CT-KAT validates builds and KATs. It then performs Memcheck-based structural checks, build-sensitivity checks,

and scans for variable-latency candidates. When configured, it also runs statistical timing tests. All stages are executed within a unified workflow. CT-KAT also automatically generates C harnesses for the target functions from templates. This reduces tool-specific manual harness development and supports reproducible screening.
2. CT-KAT consolidates findings on secret-dependent branches and memory addresses, timing distributions, and candidate variable-latency instructions into a tabular report. It provides a nine-class verdict taxonomy and a continuous-integration (CI) gate. Its triage workflow explicitly separates automated candidate collection from the manual determination of whether each candidate is secret dependent.
3. We evaluate CT-KAT on a PQClean-based corpus [8]. The corpus includes implementations of NIST-standardized PQC algorithms. The baseline ML-KEM target produces no new risk signals. The KyberSlash reproduction target passes branch- and memory-based structural checks, but assembly-level scanning and manual triage identify the risk posed by secret-dependent division. For ML-DSA and SLH-DSA/SPHINCS$^+$, manual triage attributes the reported structural failures to specification-permitted variable-time behavior. These findings are therefore distinguished from genuine implementation defects. Cases with inconclusive evidence, such as Falcon-512, are not accepted by default. Instead, they are retained for further analysis.

The remainder of this paper is organized as follows. Section 2 reviews the CT threat model and the major detection methods. Section 3 surveys related work and discusses how CT-KAT differs from previous approaches. Section 4 describes the configuration model, automatic harness generation, screening layers, and triage-based verdict system of CT-KAT. Section 5 presents the evaluation results obtained using synthetic control cases and a PQClean-based PQC corpus, together with representative case studies. Finally, Section 6 concludes the paper and discusses its limitations and directions for future work.

2 Background

Constant-time (CT) programming aims to ensure that a program's execution time does not depend on secret values [11]. If execution time varies with a secret, an adversary may infer information about the secret by collecting and analyzing timing measurements. In practice, a CT implementation should avoid three major forms of secret-dependent behavior.
First, a *secret-dependent branch* causes control flow to depend on a secret value. As a result, different execution paths may have different execution times. Second, a *secret-dependent memory access* uses a secret value to determine a memory address or array index. This can make the access latency depend on the selected memory location. If that location is or is not present in the cache, the timing difference may reveal information about the secret [2]. Third, a secret-derived value may be used as an operand of a variable-latency instruction. Some instructions,

such as integer division, have execution latency that depends on their operands. When those operands are derived from secret values, timing leakage may occur. This can happen even in the absence of secret-dependent branches or memory accesses [3]. Detection techniques for these patterns fall into three broad categories: structural taint analysis, statistical timing tests, and assembly-level scanning. These categories have distinct blind spots. In addition, any verdict must account for the build configuration, which can itself change the result.

CT behavior is, moreover, a property of the compiled binary rather than of the source code alone. Simon et al. [17] experimentally demonstrated that the same source code can produce different structural-analysis verdicts depending on the compiler version and optimization level. For example, a branch causing a violation under `-O0` may become a passing conditional move under `-O2`, while optimization-induced inlining can introduce a new violation. Consequently, a verdict obtained under a single build configuration may not represent the CT behavior of the deployed binary.

These observations motivate the combined use of multiple detection methods across multiple build configurations.

3 Related Work

CT-KAT builds on two established techniques.The first is Valgrind-based structural analysis, pioneered by Langley's ctgrind [10]. In this approach, Valgrind/Memcheck [15] issues a warning when a secret-marked value influences a branch or memory address. The idea was incorporated into SUPERCOP's TIMECOP for large-scale screening [4], and Microwalk extends it by also quantifying leakage magnitude [18]. These approaches can localize leakage precisely. However, they analyze only executed paths and may miss instruction-latency leakage, such as KyberSlash. They also typically target a single analysis objective.

The second technique is statistical timing analysis. dudect applies the Welch t-test from Test Vector Leakage Assessment (TVLA) [6] to execution-time measurements. It compares fixed and random input classes to detect empirical timing differences. These differences may include cache and branch-prediction effects. This analysis does not require a hardware timing model. However, it does not localize the responsible code and is sensitive to measurement noise, so it may miss rare leakage. Structural and statistical analyses thus have complementary blind spots; neither can replace the other.

Recent surveys reinforce this gap. Geimer et al. compared roughly 40 CT tools and showed that their coverage and usability vary widely [5]. Jancar et al. found that developers rarely use such tools in practice [7]. Similar concerns are also reported in a study of NIST PQC signature candidates [1].

CT-KAT is designed to address this gap. Rather than introducing a new detection algorithm, it integrates existing checks into a unified workflow. Specifically, it combines Valgrind-based structural analysis and dudect-based statistical testing with assembly-level instruction scanning, build-sensitivity analysis, and KAT-based correctness validation in a single declarative pipeline. CT-KAT also

automatically generates harnesses for target functions. In addition, it consolidates heterogeneous analysis results into a verdict framework that supports manual triage. CT-KAT provides automatic harness generation and unified triage of heterogeneous results. These two capabilities are difficult to obtain from existing standalone tools.

4 System Design and Implementation

CT-KAT is designed as a conservative screening pipeline rather than a proof system. It uses one declarative YAML specification to describe the project, build command, KAT target, CT harnesses, optional dudect harnesses, matrix builds, and reporting policy. The same specification records compilers, optimization flags, seeds, and thresholds, so that a reported verdict can be reproduced and audited.

4.1 Configuration and Harness Generation

CT-KAT first validates that the configured build succeeds and that all declared artifacts are produced. It then runs the known-answer test and accepts it only when the configured number of test cases is observed in the output. A target that fails this functional check is reported as inconclusive rather than as a constant-time failure.

For analysis, CT-KAT generates small C harnesses from YAML metadata. The user declares the target function, argument roles, and either full secret buffers or explicit secret byte ranges. This distinction is important for PQC secret-key objects, which often contain embedded public components. By marking only the true secret regions, CT-KAT avoids false positives caused by legitimate operations on public data. A coverage check further confirms that the declared regions span the secret-key length reported by the implementation. This prevents a mistaken specification from silently under-tainting the secret and producing a false pass.

The framework selects a generic-function, KEM, or signature template. Structural harnesses mark secret data with Valgrind client requests and invoke the target function under Memcheck. Timing harnesses construct fixed-versus-random input classes for dudect. Because the two analyses require different inputs, CT-KAT uses separate harnesses when both are configured. The current artifact provides dudect templates for generic functions, KEM APIs, and signature APIs. When a harness uses fresh-key classes, the KEM and signature timing templates perform cache normalization before each measurement. This step reduces the imbalance introduced by per-iteration key generation.

4.2 Screening Layers

After build and KAT validation, CT-KAT applies four complementary screening layers. First, Valgrind/Memcheck reports whether secret-tainted values reach

a branch condition or memory-address computation. This layer localizes structural violations but observes only executed paths and does not detect operand-dependent instruction latency.
Second, ct-matrix recompiles the same harness across both gcc and clang at five optimization levels from `-O0` to `-Os`. A target whose structural verdict changes across builds is not treated as clean merely because one build passes.
Third, asm-scan compiles each source at several optimization levels, from `-O0` through `-Os`. It then searches the generated assembly for integer-division instructions, because their latency depends on the operands. asm-scan ignores multiplies by design. This is because an optimizing compiler frequently replaces a secret-dependent division with a constant-time multiply-by-reciprocal. As a result, a surviving `div` represents the genuine risk. Critically, such a division may be emitted only at some optimization levels. It may be absent at `-O0`, the configuration used by the Valgrind layer. Therefore, a single-build structural check may miss it. asm-scan collects candidates only; whether an operand is secret derived is forwarded to manual triage.
Finally, when a dudect harness is configured, CT-KAT collects timing samples from two input classes and applies Welch's t-test.
Following Reparaz et al. [16], CT-KAT reports the largest $|t|$ over a set of percentile crops of the upper tail. This design prevents a few slow outliers from masking a real timing difference.
CT-KAT also computes a per-batch t-statistic over the samples in arrival order. A genuine effect is expected to remain stable across batches. By contrast, a statistic that swings between batches indicates a noisy measurement environment rather than leakage.
Cohen's d is reported alongside each $|t|$ to separate effect size from sample count. This layer surfaces empirical timing effects, but it is sensitive to the measurement environment and does not localize the responsible code.

4.3 Triage-Based Verdict Assignment

The classifier separates automated evidence from analyst judgment. Clean structural results, build-sensitive verdicts, asm-scan candidates, timing alarms, and KAT failures are first collected into a report. When the evidence is unambiguous, CT-KAT assigns a verdict automatically. When a decision depends on semantic information, the reviewer records a triage justification. This justification may state, for example, that a division operand is public or that an observed variable-time path is accepted by the relevant specification.
The nine verdict classes pair a plain meaning with the label used in Table 2. These classes cover targets with no risk signals (`robust`); structurally clean targets with untriaged candidates (`ct-clean-untriaged`); structurally clean targets with an incomplete assembly scan (`ct-clean-asm-incomplete`); verdicts that flip across builds (`build-sensitive-ct`); confirmed secret-dependent variable-latency divisions (`varlat-secret-risk`); specification-accepted variable-time behavior (`accepted-variable-time`); unresolved findings retained under default-

deny (`needs-analysis`); confirmed leaks (`ct-leak`); and tool or environment problems (`tool-problem`).
The CI gate passes only clean targets or targets whose variable-time behavior has been explicitly accepted through recorded triage. Unreviewed candidates, build-dependent verdicts, timing alarms, and tool failures block the target by default.

5 Evaluation

The evaluation is designed to answer four questions. First, can structural analysis detect known secret-dependent branches and memory accesses? Second, can asm-scan and manual triage identify KyberSlash-like variable-latency risks that structural analysis misses? Third, when applied to real PQC implementations, can CT-KAT distinguish targets with no observed risk signals from those with specification-permitted variable-time behavior or inconclusive evidence? Fourth, does the dudect timing layer corroborate these verdicts, and how sensitive is it to the measurement environment?
To answer these questions, we use two corpora. The synthetic control corpus consists of small C functions that reproduce representative leakage patterns. The implementation corpus consists of PQClean implementations [8] of ML-KEM, ML-DSA, SLH DSA/SPHINCS$^+$, and Falcon, together with a modified ML-KEM target that reproduces KyberSlash.

Table 1. Experimental environments used for structural and timing analyses.

Item	Structural analyses	dudect timing snapshot
Machine	Samsung Galaxy Book	non-virtualized machine
CPU	Intel Core Ultra 5 125H	Intel Core i7-10875H
Operating system	Ubuntu 24.04.4 LTS on WSL2	Ubuntu 24.04
Compiler	GCC 13.3.0, Clang 18.1.3	GCC 13.3.0
Measurement setting	Virtualized WSL2 environment	`performance` governor, Turbo Boost disabled, fixed 2.3 GHz, single-core pinning

Table 2 summarizes the final CT-KAT verdicts for the principal evaluation targets. In the synthetic control corpus, structural analysis detected the intentionally introduced secret-dependent branches and secret-indexed array accesses. Constant-time counterparts implementing the same functionality were classified as `robust`. A further control changed its structural verdict depending on whether compiler optimization was enabled. CT-KAT classified this target as `build-sensitive-ct`. These results show that CT-KAT does not generalize a `PASS` result obtained under a single build configuration to the implementation as a whole. Instead, it preserves build-dependent verdict changes as a separate risk signal. The secret-index control is intentionally leaky and was manually reviewed as a confirmed `ct-leak`. It is therefore not treated as an unresolved `needs-analysis` case.

Table 2. Summary of screening results for representative PQC and synthetic targets. *Structural*: the Memcheck-based constant-time verdict; *Var.-latency*: the triage outcome for asm-scan division candidates (`public`, `secret-risk`, or `untriaged`).

Target	Structural	Var.-latency	Final Verdict
ML-KEM-512/ 768/1024	`PASS`	`public`	`robust`
ML-KEM-768 (KyberSlash)	`PASS`	`secret-risk`	`varlat-secret-risk`
ML-DSA-44/65/87	`FAIL`	`public`	`accepted-variable-time`
SLH-DSA/ SPHINCS$^+$-SHA2-128f	`FAIL`	`public`	`accepted-variable-time`
Falcon-512	`FAIL`	`untriaged`	`needs-analysis`
Secret-index control	`FAIL`	–	`ct-leak`
Secret-index CT counterpart	`PASS`	–	`robust`
Build-sensitivity control	`FAIL/PASS`	–	`build-sensitive-ct`
Build-sensitivity CT counterpart	`PASS`	–	`robust`

To address the fourth question, we collected a dudect timing snapshot for the targets with configured timing harnesses. The results are summarized in Table 3. The dudect templates cover generic functions, KEM APIs, and signature APIs. Thus, ML-DSA, SLH-DSA/SPHINCS$^+$, and Falcon are exercised by dudect rather than reported through structural analysis alone.

The synthetic control reproduced the expected separation. The leaky function failed with $|t| = 157.2$ over 50,000 measurements, whereas the constant-time variant passed with $|t| = 1.38$.

For ML-KEM-768, the normal-path (ct) and implicit-rejection (fo) axes both hold the secret key fixed and vary only the ciphertext. Both axes passed, with $|t| = 2.30$ and 2.10, respectively.

The fixed-versus-fresh secret-key axis instead produced a large and stable statistic, with $|t| = 145$. This result is consistent with a harness confound rather than a decapsulation leak. The axis regenerates a key pair in every iteration, causing per-iteration key generation to perturb cache and predictor state asymmetrically between the classes. We therefore rely on the sk-fixed ct and fo axes for ML-KEM.

For the signature targets, the same fixed-message axis left every scheme below the warning threshold on the native machine. The observed values were $|t| \leq 1.75$ for ML-DSA, 1.59 for Falcon, and 1.52 for SLH-DSA/SPHINCS$^+$. We interpret this result as showing no stable key-class timing difference under this harness, not as a constant-time guarantee. The rejection-sampling and Gaussian signers have large within-class variance, which dominates the statistic. Cohen's $|d|$ was below 0.06 for every target. The accepted variable-time behavior had already been recorded

through triage. Controlled measurement also mattered. An earlier unpinned, frequency-varying run produced a spurious Falcon alarm with $|t| = 13.7$. The alarm vanished once the frequency was fixed.

Table 3. dudect timing snapshot, collected on a native x86_64 machine (Intel Core i7-10875H, fixed 2.3 GHz, single core). A `PASS` bounds the fixed-vs-fresh secret-key or ciphertext timing difference below the warning threshold; it is not a constant-time proof. The ML-KEM fixed-vs-fresh-key axis is excluded as a harness confound (see text).

Target	**Harness scope**	**Samples**	$\|t\|$	**Status / interpretation**
Synthetic timing leak	generic	50,000	157.21	`FAIL`; expected positive control
Synthetic timing CT	generic	50,000	1.38	`PASS`; constant-time counterpart
ML-KEM-768 (ct / fo)	KEM	50,000	2.30 / 2.10	`PASS`; sk-fixed axes (fixed-vs-fresh-key axis excluded, see text)
ML-DSA-44/65/87	signature	50,000	1.15–1.75	`PASS`; no key-dependent offset
Falcon-512	signature	20,000	1.59	`PASS`; within-class variance dominates
SLH-DSA/ SPHINCS$^+$-128f	signature	3,000	1.52	`PASS`; within-class variance dominates

5.1 ML-KEM and KyberSlash

The baseline ML-KEM-512/768/1024 implementations produced no findings of secret-dependent branches or secret-dependent memory accesses under any evaluated build configuration.

asm-scan flagged division instructions only under gcc `-Os` and clang `-O0`, in the `shake128` and `shake256` functions. Manual triage attributed these instructions to public symmetric computations rather than secret-derived data. CT-KAT therefore classified the baseline ML-KEM targets as `robust`.

A second structural harness decapsulates a single-bit-corrupted ciphertext under the same tainted key. This lets Valgrind cover the Fujisaki–Okamoto implicit-rejection path, rather than only valid ciphertexts. The additional harness produced no findings for the baseline.

We also configured secret regions to exclude public components embedded in secret-key objects from secret tainting. This reduced false positives caused by legitimate operations on public data stored in the same buffers as secret material.

In contrast, the KyberSlash reproduction target passed structural analysis. The reproduced pattern does not involve a secret-dependent branch or a memory address selected by a secret. Instead, a secret-derived value is supplied as an operand to an integer division. asm-scan flagged this division in `poly_compress` under gcc `-Os` and clang `-O0`, but not under `-O1`–`-O3`, where the compiler lowers it to a constant-time reciprocal multiply. Because the Valgrind layer runs at `-O0`, a single optimized build would have left this operand-dependent division undetected. Manual triage confirmed that its operand was secret derived, and CT-KAT classified the target as `varlat-secret-risk`. This case demonstrates that CT-KAT surfaces instruction-latency risks that structural analyses limited to branch and memory dependencies miss.

5.2 ML-DSA, SLH-DSA/SPHINCS$^+$, and Falcon

The ML-DSA signing implementations produced a `FAIL` result in structural analysis. Manual review, however, attributed the principal findings not to direct branching or indexing on the long-term secret key, but to signing-time rejection behavior and public-output generation paths. The reviewer judged these paths to be permitted by the relevant specification.

CT-KAT does not suppress such findings automatically. It assigns the `accepted-variable-time` verdict only when the manual attribution and its specification-based justification have been recorded. This verdict records acceptance under CT-KAT's screening policy. It is not a proof that the signing implementation is constant time.

Similarly, for the SLH-DSA/SPHINCS$^+$ signing implementation, the findings were attributed to signature-generation state that is later represented in the public signature. They were not attributed to an unresolved long-term-key dependency. After this reasoning was recorded through triage, the target was assigned the same `accepted-variable-time` verdict.

By contrast, the findings for Falcon-512 were associated with Gaussian sampling, floating-point arithmetic, and secret-key handling. No public-only explanation was sufficient to close the case. CT-KAT therefore did not allow the target to pass and retained it as `needs-analysis`.

This contrast shows that CT-KAT neither treats every structural `FAIL` result as a confirmed defect nor accepts an insufficiently justified finding as safe.

Overall, CT-KAT separates targets with no observed risk signals as `robust`. It classifies KyberSlash-like risks missed by structural analysis as `varlat-secret-risk`. It treats specification-permitted variable-time behavior with documented justification as `accepted-variable-time`. It retains unresolved targets under a default-deny policy as `needs-analysis`.

6 Conclusion

This paper presented CT-KAT, an integrated screening platform for the practical assessment of constant-time risks in C cryptographic implementations. The evaluation focused on PQC C implementations.

From a single YAML specification, CT-KAT validates the build and KAT execution and automatically generates harnesses for the target functions. It then executes a unified pipeline consisting of Valgrind-based structural analysis, ct-matrix analysis across multiple build configurations, assembly-level collection of candidate variable-latency instructions, and, where configured, dudect-based statistical timing testing.

CT-KAT does not regard successful KAT execution as evidence of constant-time behavior. Instead, it makes the blind spots of each analysis layer explicit and combines their complementary signals.

CT-KAT is a screening tool rather than a formal verification tool. It surfaces risk candidates and supports verdict assignment, but it does not prove constant-time behavior. Because its Valgrind- and dudect-based analyses are dynamic, it cannot expose leakage on paths that the harness does not exercise, including rare or adversarially induced paths.

Its scope is limited to timing-relevant branches, memory accesses, and variable-latency instructions. Power and electromagnetic (EM) side channels, fault attacks, and higher-order leakage in masked implementations require separate analyses.

asm-scan reports only candidate instructions, so final verdicts may require manual triage. A `public` or `accepted-variable-time` verdict is only as sound as its recorded justification. Statistical timing analysis is also environment sensitive. Scheduling, cache state, frequency scaling, and virtualization can all add variability. Thus, dudect results should be treated as complementary evidence alongside structural analysis and asm-scan.

Finally, because a YAML specification invokes configured build and test commands, an untrusted specification must not be run without review.

The first direction for future work is to strengthen automated secret tracking for asm-scan candidates. CT-KAT currently collects candidate variable-latency instructions and requires an analyst to inspect the provenance of their operands. Incorporating operand-taint analysis at the assembly or intermediate representation (IR) level could reduce this manual triage burden.

The second direction is to broaden the scope of the evaluation. This paper demonstrated the utility of CT-KAT using ML-KEM, ML-DSA, SLH-DSA/ SPHINCS$^+$, Falcon, and synthetic control cases. The dudect timing layer now covers signature APIs in addition to KEM and generic-function harnesses. These measurements were collected on a native x86_64 machine. Refining the KEM fixed-versus-fresh-key axis so that it isolates decapsulation timing from key-generation side effects is a useful next step for this layer.

Including a wider range of PQC implementations and library-specific build configurations would enable a more rigorous evaluation of the verdict framework. In addition, linking CT-KAT's screening results to formal verification tools, masking analysis, and power/EM measurements could support a staged security-assessment workflow, progressing from automated preliminary screening to high-assurance validation.

References

1. Adjonyo, O., Bardin, S., Bellini, E., Dione, G.N., Ameen, M.F.A., Merget, R., Recoules, F., Sellami, Y.: Systematic timing leakage analysis of NIST PQDSS candidates: Tooling and lessons learned. arXiv preprint arXiv:2509.04010 (2025)
2. Bernstein, D.J.: Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (2005)
3. Bernstein, D.J., Bhargavan, K., Bhasin, S., Chattopadhyay, A., Chia, T.K., Kannwischer, M.J., Kiefer, F., Paiva, T., Ravi, P., Tamvada, G.: Kyberslash: Exploiting secret-dependent division timings in Kyber implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(2), 209–234 (2025). https://doi.org/10.46586/tches.v2025.i2.209-234
4. Bernstein, D.J., Cottaar, J., Di Giandomenico, E., Hövelmanns, K., Hülsing, A., Kudinov, M., Lange, T., Mahzoun, M., Meijers, M., Pellegrini, A., et al.: Report on evaluation of KPQC round-2 candidates. Cryptology ePrint Archive (2024)
5. Geimer, A., Vergnolle, M., Recoules, F., Daniel, L.A., Bardin, S., Maurice, C.: A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 1690–1704 (2023)
6. Gilbert Goodwill, B.J., Jaffe, J., Rohatgi, P., et al.: A testing methodology for side-channel resistance validation. In: NIST non-invasive attack testing workshop. vol. 7, pp. 115–136 (2011)
7. Jancar, J., Fourné, M., Braga, D.D.A., Sabt, M., Schwabe, P., Barthe, G., Fouque, P.A., Acar, Y.: "They're not that hard to mitigate": What cryptographic library developers think about timing attacks. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 632–649. IEEE (2022)
8. Kannwischer, M.J., Schwabe, P., Stebila, D., Wiggers, T.: Improving software quality in cryptography standardization projects. In: IEEE European Symposium on Security and Privacy Workshops (EuroS&P Workshops). pp. 19–30. IEEE Computer Society (2022). https://doi.org/10.1109/EuroSPW55150.2022.00010, https://eprint.iacr.org/2022/337
9. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Advances in Cryptology – CRYPTO '96. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996). https://doi.org/10.1007/3-540-68697-5_9
10. Langley, A.: ctgrind: Checking that functions are constant time with Valgrind. https://github.com/agl/ctgrind (2010)
11. Miao, Y., Kandemir, M.T., Zhang, D., Zhang, Y., Tan, G., Wu, D.: Hardware support for constant-time programming. In: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 856–870 (2023)
12. National Institute of Standards and Technology: Module-lattice-based digital signature standard. Federal Information Processing Standards Publication NIST FIPS 204, U.S. Department of Commerce (2024). https://doi.org/10.6028/NIST.FIPS.204
13. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. Federal Information Processing Standards Publication NIST FIPS 203, U.S. Department of Commerce (2024). https://doi.org/10.6028/NIST.FIPS.203
14. National Institute of Standards and Technology: Stateless hash-based digital signature standard. Federal Information Processing Standards Publication NIST FIPS

205, U.S. Department of Commerce (2024). https://doi.org/10.6028/NIST.FIPS.205
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices **42**(6), 89–100 (2007)
16. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. pp. 1697–1702. IEEE (2017)
17. Simon, L., Chisnall, D., Anderson, R.: What you get is what you C: Controlling side effects in mainstream C compilers. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 1–15. IEEE (2018)
18. Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B.: Microwalk: A framework for finding side channels in binaries. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 161–173 (2018)