electronics

MDPI

Article

CT-KAT: Beyond Pass/Fail in Constant-Time Screening of PQC Implementations

Seung-Won Lee, Min-Seo Kim, Su-Min Jeong and Hwa-Jeong Seo *

Department of Information Computer Engineering, Hansung University, Seoul 02876, Republic of Korea
* Correspondence: hwajeong84@gmail.com

Abstract

Constant-time screening of post-quantum cryptographic implementations must simultaneously examine secret-dependent execution structure, binary instructions, and timing. Because tests observe different phenomena, no single result supports assessment. We present CT-KAT, which generates test harnesses from configurations and connects functional validation, structural analysis across builds, instruction scanning, and timing measurements. CT-KAT records measurement validity, input relationships, and manual-review status, withholding a no-issue status when evidence is insufficient. We evaluate 25 target-operation pairs and 206 build combinations from 24 pairs with build artifacts. Structural-review targets increase from 11 under one deployment build to 13 across the full matrix; instruction scanning expands the review set to 22. Three of 28 timing comparisons show repeatable differences, but all change public and secret values together. In KyberSlash, structural tests show no signal while binary evidence identifies secret-related divisions. Thus, evidence must be preserved with its interpretation conditions.

Keywords: post-quantum cryptography; constant-time; timing side channel; compilation conditions; software testing; KyberSlash

1. Introduction

The transition from conventional public-key cryptography to post-quantum cryptography (PQC) is accelerating in preparation for advances in quantum computing. In 2024, the National Institute of Standards and Technology (NIST) standardized three PQC algorithms: the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) for key establishment, and the Module-Lattice-Based Digital Signature Algorithm (ML-DSA) and Stateless Hash-Based Digital Signature Algorithm (SLH-DSA) for digital signatures [1–3]. As these algorithms enter products and communication software, their behavior in real programs becomes as important as the mathematical security of the algorithms themselves. If a cryptographic program follows different branches, accesses different memory locations, or uses instructions whose latency depends on secret data, an attacker may infer secrets by observing execution time. Such risks have long been studied in conventional public-key cryptography and remain important at the implementation level for PQC.

Constant-time implementations aim to preserve the same control flow and memory-access pattern regardless of secret values and to avoid feeding secrets to operations whose latency may depend on their inputs. No single test, however, captures all of these properties. **Valgrind**-based dynamic analysis checks whether secrets influence branch decisions or memory addresses [4,5]. Binary inspection examines assembly and executables produced under multiple build conditions for instructions such as division, whose latency may vary

Check for updates

Academic Editors: João Pedro Menoita Henriques and Filipe Caldeira

Received: 28 August 2026
Revised: 29 October 2026
Accepted: 29 October 2026
Published: 1 November 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

with their operands. Statistical timing tools such as **dudect** repeatedly execute two input classes and test whether their timing distributions differ [6]. Because these methods observe different phenomena, each has strengths and blind spots. Practitioners likewise report difficulties. A survey of 44 developers of 27 open-source cryptographic libraries found that the developers were aware of timing attacks but often gave priority to other issues because defending against such attacks was perceived as costly. The same survey also identified several shortcomings of existing constant-time analysis tools [7].

The central difficulty is that these results do not carry the same meaning. Finding a suspicious instruction in a binary does not by itself demonstrate leakage because the instruction may process only public data. Conversely, a clean branch-and-memory result is insufficient: as KyberSlash shows, a secret may reach a variable-latency instruction even when neither control flow nor memory access depends on that secret [8]. On two Arm platforms, these division timings alone were enough to recover Kyber secret keys within minutes for KyberSlash2 and within a few hours for KyberSlash1. A large timing difference is also ambiguous if the experiment changes a secret key and a public ciphertext together, because the difference cannot be attributed to the key alone. Results further depend on compilation and the measurement environment. A finding absent from one build may appear under another compiler or optimization setting [9], while unstable systems or failed controls can make a numerical timing result unfit for a security conclusion.

Simply running several tests in sequence is therefore insufficient. Each result must be examined together with what the test observed, the source and executable that produced it, the values that differed between input classes, and the suitability of the measurement environment. This paper accordingly asks how heterogeneous test results can be connected and interpreted, and how an unsupported claim of safety can be prevented when evidence is missing or causality remains unclear.

To address this problem, we propose the **CT-KAT** framework. Rather than introducing a new standalone detector, **CT-KAT** makes multiple testing methods repeatable under shared configuration and reporting rules. It first checks functional correctness and then tests secret-dependent branches and memory accesses. It rebuilds the same source with multiple compilers and optimization settings, examines whether potentially variable-latency instructions remain in the resulting binaries, and finally measures actual execution time. Timing results are interpreted only when the measurement environment and controls are valid. Repetitive execution and evidence collection follow the configuration, while judgments such as whether a suspicious instruction actually depends on secret data remain subject to human review.

This work makes the following three contributions.

1. We propose **CT-KAT**, a framework that repeatedly runs multiple constant-time tests under a shared configuration and links every result to the same target and execution conditions. We evaluate it on 25 target–operation pairs, 206 build combinations, and 28 timing conditions.
2. We introduce a common evidence format that separates an observed signal from its interpretation by recording the source, binary, build conditions, input relationship, measurement validity, value provenance, and human-review status. We also define a decision rule that withholds a no-issue status whenever required evidence is missing or invalid.
3. By applying **CT-KAT** to multiple PQC implementations, we show that the set of risk candidates varies with build conditions and testing methods. The number of pairs requiring structural review grows from 11 under one deployment build to 13 across the full build matrix, and adding instruction scanning raises it to 22. We further

demonstrate that a timing difference must be interpreted together with the values changed between the two input classes rather than by magnitude alone.

The remainder of this paper is organized as follows. Section 2 introduces the concepts needed to understand constant-time implementations, major PQC operations, compilation, and timing measurement. Section 3 reviews prior methods for examining execution structure and timing, discusses KyberSlash, and identifies the position of **CT-KAT** relative to prior work. Section 4 presents the design and implementation of **CT-KAT**, from test configuration and harness generation to complementary tests, evidence validation, and final status assignment. Section 5 evaluates the effects of multiple build conditions, the distribution of final statuses, KyberSlash, timing results, and applications to diverse projects and existing tools. Section 6 summarizes the main findings, limitations, and directions for future work.

2. Background

Cryptographic programs process both public values, such as messages and ciphertexts, and values that must remain secret, such as secret keys. Even when a program produces correct outputs, an attacker may infer secrets if execution time or memory behavior changes with secret data [10,11]. A constant-time implementation does not attempt to make every execution take exactly the same wall-clock time; rather, it follows implementation principles that prevent program behavior from depending on secrets [12].

The PQC schemes considered in this paper comprise key-establishment and digital-signature schemes. Key establishment includes key generation, encapsulation that produces a ciphertext and a shared secret, and decapsulation that recovers the same shared secret from a secret key and the ciphertext. Digital signatures comprise key generation, signing with a secret key, and verification with a public key. These operations use secrets and external inputs in different roles. Those roles determine which values an attacker can vary and what may be inferred from timing. Consequently, a similar timing difference can have different security implications depending on whether it occurs during key generation, decapsulation, or signing.

PQC implementations use computations over large integers and polynomials, modular reduction, conditional selection, and loops that continue until a condition is met. C code expresses these computations as arithmetic, branches, and memory accesses, which a compiler then converts into processor instructions. The form of a branch or the selected instruction may change with the compiler and optimization setting.

Compiled binaries run on concrete hardware, where measured time depends not only on program computation but also on operating-system scheduling, cache state, processor temperature, and clock frequency. Statistical timing tests therefore examine distributions from repeated measurements rather than relying on a single observation. They repeatedly execute two input classes and compare their mean difference with measurement variability. The meaning of that comparison depends on which values are held fixed and which are changed between the classes.

3. Related Work

This section first classifies existing constant-time tests by what they observe and reviews how recent PQC studies use them. It then examines large-scale studies that combine multiple methods and the KyberSlash case before identifying the questions that remain unresolved. Table 1 summarizes the tools and studies discussed in this section, together with the scale of their evaluations and the metrics they report.

Table 1. Constant-time testing tools and empirical studies discussed in this section, with the scale of their evaluations and the metrics they report. Entries are taken from the cited works.

Work	Approach and Observation	Evaluated Scale	Reported Metrics and Findings
Testing tools			
ctgrind [5]	Dynamic; Valgrind Memcheck on values marked as secret	Case studies (donna-c64, OpenSSL)	Secret-dependent memory access in OpenSSL's constant-time modular exponentiation; no aggregate metrics
TIMECOP [13]	Dynamic; Valgrind-based	More than 2700 SUPERCOP implementations	Scale of application only
dudect [6]	Statistical; Welch *t*-test on two input classes	Several variable-time implementations	Detection of known variable-time code; about 300 lines of C
MicroWalk [14]	Execution traces with mutual information	15 implementations in Intel IPP and Microsoft CNG	Previously unknown leakages; 112 million instructions in about 105 CPU minutes
ct-verif [15]	Formal; product programs on optimized LLVM	Components of NaCl, OpenSSL, and FourQ	Verified constant-time components
Binsec/Rel [16,17]	Relational symbolic execution on binaries	338 implementations; 4148 binaries in a compiler study	`gcc -O0` and Clang back-end passes introduce violations
CtChecker [18]	Sound static analysis of source code	24 K lines of code	Analysis in under one minute; fewer false positives than prior sound tools
Empirical studies			
Geimer et al. [19]	Systematic tool evaluation	34 frameworks classified, 5 evaluated	Missed vulnerabilities due to SIMD, implicit flows, and internal secret generation
Hanyecz et al. [20]	ctgrind and RTLF on SQIsign lattice reduction	One GCC build; 144,000 timing inputs	No signal; positive control detected; compiler effects not studied
Adjonyo et al. [21]	TIMECOP, Binsec/Rel2, dudect, and RTLF	302 implementation items in 15 schemes	26 reports, 5 fixed; 67.1% of a reviewed TIMECOP sample not violations
Zhang and Barthe [22]	Secret flow in LLVM IR (CT-LLVM)	9 libraries	36% of functions analyzable; 61% of these constant time
Bernstein et al. [8]	Division-timing attack (KyberSlash)	Kyber implementations on 2 Arm platforms	Key recovery within minutes (KS2) to hours (KS1)

3.1. Testing Methods

Dynamic tools differ in what they observe. **ctgrind** and **TIMECOP** use the **Valgrind** instrumentation framework to locate branches and memory addresses influenced by secret values [5,13,19], which lets them identify the code locations responsible for a finding. **dudect** instead repeatedly measures two input classes to detect differences in their timing distributions, whereas **MicroWalk** compares instruction traces and memory-access patterns across inputs [14]. Although all of these tools address constant-time risks, they produce different forms of evidence: execution structure, observed time, and execution traces. Systematic comparisons likewise report that the detectable risks and the precision of results vary across methods. Geimer et al. classified 34 side-channel detection frameworks and evaluated five of them on a common benchmark, finding that the tools often missed vulnerabilities because of limited support for SIMD instructions, implicit flows, and internal secret generation [19].

Whereas dynamic and timing tests search for signals along executed paths, formal analyses verify whether two executions preserve the same control flow and memory accesses under a specified model. **ct-verif** [15] applies this approach to optimized LLVM code, while **Binsec/Rel** [16] uses relational symbolic execution on compiled binaries to establish constant-time behavior within a defined scope [17,18]. When they complete, these methods offer stronger evidence than screening; however, external functions and complex execution paths may be difficult to model or analyze to completion.

In 2025, Hanyecz et al. implemented the lattice-reduction component of SQIsign [23] in constant time and evaluated it with **ctgrind** and the statistical timing tool **RTLF** [20]. A **ctgrind** test under one GCC build found no secret-dependent branch or change in execution behavior, and a timing experiment with 144,000 inputs found no difference between its two classes. The same procedure did distinguish a known variable-time implementation, providing a positive control for the measurement. This study demonstrates the value of combining structural and timing evidence. Its scope, however, was limited to lattice reduction rather than the complete SQIsign implementation, and the authors explicitly avoided claiming that the entire signer was constant time. Compiler-optimization effects were also left for future work.

3.2. Large-Scale Analysis and Tool Integration

In 2025, Adjonyo et al. assembled **TIMECOP, Binsec/Rel2, dudect, and RTLF** into a tool suite for analyzing NIST's additional digital-signature candidates [21]. They automated harness and configuration generation and analyzed 302 implementation items across 15 schemes. Of 26 reports sent to developers, five were confirmed and fixed. The study shows that a combined workflow can cover a broader range of risks than isolated tests.

At the same time, the study demonstrates that integrated execution does not eliminate the need for interpretation. **TIMECOP** explored more paths and produced roughly 17 times as many warnings as **Binsec/Rel2**, yet 67.1% of a limited manually reviewed TIMECOP sample were not actual constant-time violations. Major causes included treating public regions embedded in a secret-key object as secret and ignoring feasible value ranges. **Binsec/Rel2** produced more precise results but could fail to complete on complex paths. The input partitioning used by **dudect** and **RTLF** also relied on generic key bits rather than internal scheme behavior and thus did not capture relationships involving unselected bits or intermediate values. Large-scale coverage and the interpretation of individual findings must therefore remain distinct.

In the same year, Zhang and Barthe proposed **CT-LLVM**, which tracks secret flow in LLVM intermediate representation, and applied it to nine cryptographic libraries [22]. On average, 36% of all functions could be analyzed automatically, and 61% of the analyzable functions satisfied the specified constant-time property. The study also reported previously unknown violations and problems introduced during compiler optimization. Functions containing external calls, indirect calls, or inline assembly remained difficult to analyze completely, so the same assurance could not be extended to every function. These results show both the potential of automated analysis at scale and the need to distinguish unanalyzed regions from passing results.

3.3. KyberSlash and Instruction Latency

The KyberSlash study published in 2025 demonstrated in real Kyber implementations that timing differences can arise when a value derived from a secret becomes the operand of a division instruction. Because tests limited to branches and memory accesses may miss this behavior, recent work has begun to include the operands of instructions such as division. The 2025 tool suite discussed above inspected multiplication and division operands with **Binsec/Rel2**, while **CT-LLVM** checked whether secrets flowed into operations whose latency may depend on their inputs.

Finding such an instruction in a binary nevertheless establishes only a potential risk, not actual information leakage. Its operands may be public, and an observable timing difference may depend on the processor and input range. KyberSlash is therefore more than a failure case for existing tests: it shows that instruction presence, operand provenance, and timing observed on concrete hardware must be treated as distinct forms of evidence.

Recent work has shown that multiple constant-time tools can be configured, executed, and applied to large collections of PQC implementations. Yet running tools in one workflow does not by itself connect every result consistently to its target, source, build conditions, inputs, and measurement environment. A clear mechanism is also needed to distinguish missing or invalid evidence from a genuine passing result. The remaining problem is therefore to preserve heterogeneous evidence together with the conditions that produced it and to expose unresolved questions in the final assessment.

4. Design and Implementation of CT-KAT

To address the remaining problem, **CT-KAT** treats the target implementation, operation, build method, values changed between input classes, and measurement system as a single evidence unit. It thereby preserves not only a risk signal but also the conditions that produced it and the scope within which it can be interpreted.

4.1. Overall Workflow

Figure 1 presents the **CT-KAT** workflow based on this design principle. The user first describes the implementation, operation, inputs, secret regions, and build conditions in a configuration file. The framework generates a test harness from that configuration and checks functional correctness so that subsequent results are meaningful. It then examines execution structure, build sensitivity, suspicious instructions, and actual execution time from complementary perspectives. Finally, it combines test results with measurement validity, input conditions, and human-review status to assign a final status.

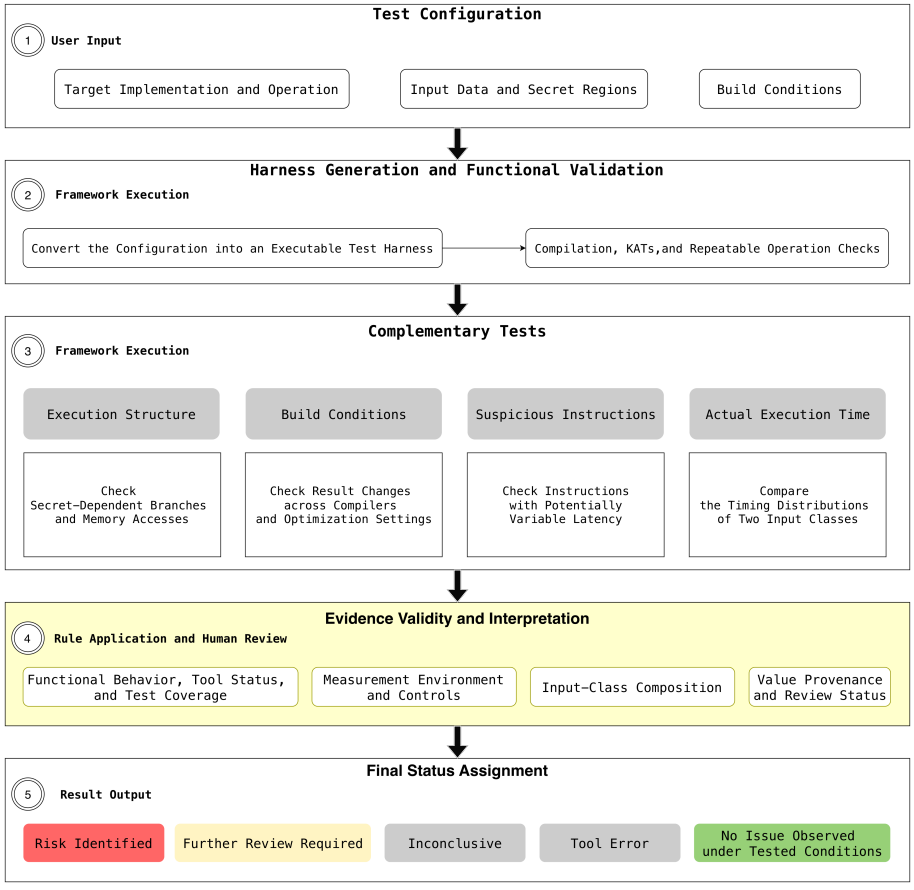


Figure 1. Stage-by-stage testing and evidence-review workflow of CT-KAT.

This workflow does not automate every judgment. The framework performs repetitive compilation, execution, result collection, and the application of machine-checkable rules. Questions such as whether a suspicious operation is permitted by the algorithm, processes only public data, or can lead to a practical attack remain subject to separate review. This division reduces variation in repetitive work without hiding judgments that resist automation. The following subsections describe Stages 1 through 5 of Figure 1 in order.

4.2. Test Configuration (Stage 1)

Cryptographic implementations differ in the form and size of operation inputs, and even the location of protected regions within a secret key may vary. A single fixed harness therefore cannot be applied unchanged to every implementation. Writing each harness by hand is possible, but it can omit required source files, mark only part of a secret, or execute a path different from the one that was intended for testing.

CT-KAT records these target-specific differences in a configuration file named `ctkat.yaml`. This file is not a measurement report; it specifies what to test and under which conditions. The user provides the operation, source and header files, ranges containing secret data, and build conditions. The same specification then serves as the shared basis for rerunning the target.

The framework does not infer the semantics of the target merely from this file. The user must still choose the operation and identify the secret regions. **CT-KAT** refuses to proceed when required fields are missing or malformed. Every run therefore starts from a complete and well-formed configuration, although whether the declared secret regions and operations match the implementation remains the user's responsibility. A validated configuration becomes the basis for generating the test harness in the next stage.

4.3. Harness Generation and Functional Validation (Stage 2)

Figure 2 shows how a Stage-1 configuration becomes a Stage-2 test harness. After the user supplies the target description, the framework selects a prepared template for decapsulation, signing, or a generic operation. It instantiates the template with code that prepares inputs, marks secret data, and invokes the selected operation. The resulting C harness is compiled with the target sources listed in the configuration, and the executable is used by later tests.

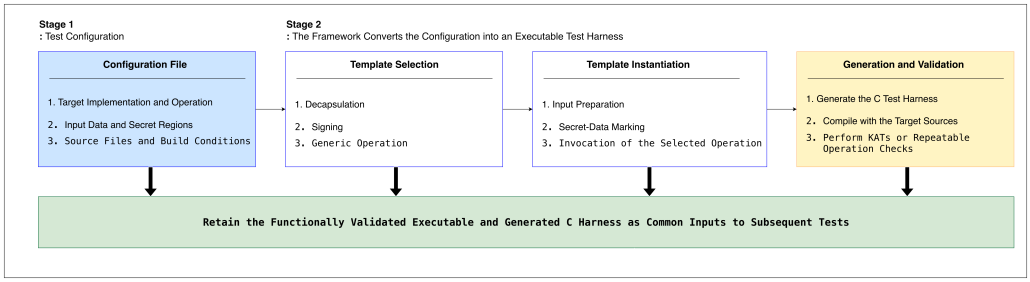


Figure 2. Generating a test harness from the configuration file.

The generated C code is retained so that an analyst can verify that the configuration was instantiated as intended. This preserves target-specific input and secret layouts while reducing variation caused by repeatedly writing harnesses by hand.

Harness generation alone is not sufficient to begin constant-time testing. A lack of timing variation in a functionally incorrect binary has no security meaning. For example, decapsulation that always fails or signing that skips part of its computation may terminate quickly and consistently without representing the actual implementation. Before further

testing, **CT-KAT** therefore checks successful compilation, required output files, known-answer tests (KATs), or the results of repeatable cryptographic operations.

If functional validation fails, the framework does not issue a no-issue status even when later tests produce no suspicious signal. Conversely, functional correctness alone does not establish constant-time behavior. By tracking correctness and timing safety as separate conditions, **CT-KAT** prevents one passing result from substituting for another requirement. Validated targets proceed to complementary tests in the next stage.

4.4. Complementary Tests (Stage 3)

This subsection corresponds to Stage 3 in Figure 1. After Stage 2 confirms that the target performs its intended function, Stage 3 collects risk evidence from four perspectives: execution structure, build conditions, suspicious instructions, and actual execution time. Figure 3 shows the question addressed by each test, the tools used, and the evidence retained for subsequent interpretation.

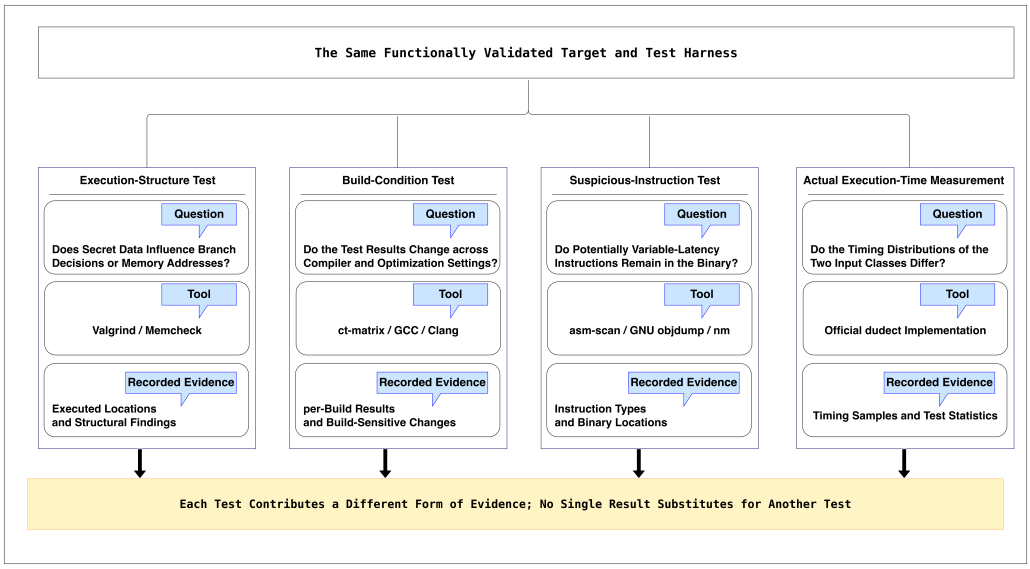


Figure 3. Four complementary tests and the evidence produced by each.

4.4.1. Execution Structure and Build Conditions

The structural test uses **Valgrind** Memcheck to observe whether values marked as secret influence branch decisions or memory-address calculations. Locating the executed code responsible for a finding helps narrow manual review. The method, however, covers only executed paths and does not directly observe instruction-latency variation.

To avoid drawing a source-wide conclusion from one build, **CT-KAT** rebuilds the same harness under multiple compilers and optimization settings. Its `ct-matrix` component produces GCC and Clang executables for each condition and applies Valgrind/Memcheck to each one. Every combination is preserved separately, including findings that appear in only one build. Compilation failures and tool errors are retained as incomplete evidence rather than discarded. The purpose is not to select the most favorable result, but to determine whether structural findings change with build conditions. We call a finding *build-sensitive* when the structural test passes under some builds and fails under others. If the same source behaves differently depending on the build, the cause lies in compilation rather than in the source, so such a finding is treated as risk evidence without review. A finding that appears under every build is different. It reflects the behavior of the source itself, which may be permitted by the algorithm, as in rejection sampling. Because the structural test

alone cannot distinguish such behavior from a real leak, the finding remains at Further Review Required until a human reviews it and only becomes Risk Identified if the review confirms a secret-related risk. This method alone still cannot detect latency differences that do not appear as branch or memory behavior.

4.4.2. Suspicious Instructions

To address this blind spot, binary inspection searches for integer division and remainder instructions whose latency may depend on their operands. The `asm-scan` component analyzes GCC- and Clang-generated files with GNU objdump and, when necessary, uses nm to locate the instructions. It compiles the same source under multiple optimization settings and records which builds retain the instruction and where it occurs in the binary. Treating every common instruction, such as multiplication, as a candidate would make review impractical; the current scope therefore focuses on the division family directly associated with KyberSlash. The result identifies candidates for review but does not establish that they produce an observable timing difference.

4.4.3. Actual Execution Time

Finally, timing measurement repeatedly executes two input classes and compares their timing distributions. **CT-KAT** constructs the input classes and collects timings, while statistical analysis uses a fixed version of the official dudect implementation. Data used to prepare a measurement are separated from the data analyzed as timing samples. For operations with complex input preparation, such as signing and decapsulation, the two classes are generated in advance; immediately before each measurement, one is selected through an equivalent operation so that preparation does not distort the measured interval.

Each Stage-3 test thus contributes a different form of evidence. None of these signals alone confirms either vulnerability or safety. Stage 4 therefore examines the validity and meaning of each item before it can contribute to the final assessment.

4.5. Evidence Validity and Interpretation (Stage 4)

This subsection corresponds to Stage 4 in Figure 1. Once Stage 3 has collected signals, Stage 4 determines whether they relate to secret data, whether the measurements are valid, which values differ between input classes, and whether human review is complete. These conditions must be considered together before assigning security meaning to an observation.

A suspicious-instruction scan identifies items for review rather than confirming vulnerabilities. For example, it may also report a division used for a public loop count or constant computation. **CT-KAT** therefore distinguishes instruction presence from whether its operands originate in secret data. In selected cases, it uses TIMECOP's modified Valgrind environment to check whether a secret reaches an instruction operand. When provenance cannot be established automatically, a human reviewer classifies the value as public, secret, mixed, or unknown by applying the propagation rules in Table A5. Each decision is stored as a review record that names its scope (target and operation), the decision, the supporting evidence files, and its limitations, so the basis of every manual classification can be audited and repeated.

Independently of operand provenance, **CT-KAT** validates whether a timing result is trustworthy through three controls. First, dividing equivalent inputs into two classes must not create a spurious difference. Second, comparing only the preparation code without the cryptographic operation must also show no difference. Third, the tool must detect a deliberately inserted delay of known size. An environment that cannot detect a known difference cannot support a safety conclusion merely because the main experiment produces a small statistic.

Passing the controls is still insufficient if the execution environment is unstable. The evidence therefore records whether measurements were pinned to one CPU, whether vir-

tualization or emulation was absent, and whether processor frequency and concurrent hardware features were controlled. Failure to satisfy any requirement for controls, environment, sample size, or result-file integrity produces an Inconclusive status regardless of the observed timing difference.

After establishing measurement validity, **CT-KAT** examines which values are fixed and changed between input classes. If a secret, a public input, and related values all change together, an observed difference cannot be attributed to any one of them. The result record therefore identifies fixed and changed values and whether each is public or secret. On this basis, each contrast is classified as *secret-only* when only secret values differ, *public-only* when only public inputs differ, or *mixed* when secret and public values differ together. When several values change, the result is not interpreted as the effect of a particular secret; even when only one input changes, the conclusion remains limited to the tested input range and environment. Operand provenance, measurement validity, and input relationships then feed the final status decision. Figure 4 summarizes how Stage 4 validates and interprets the collected evidence and how Stage 5 assigns a final status according to the decision priority.

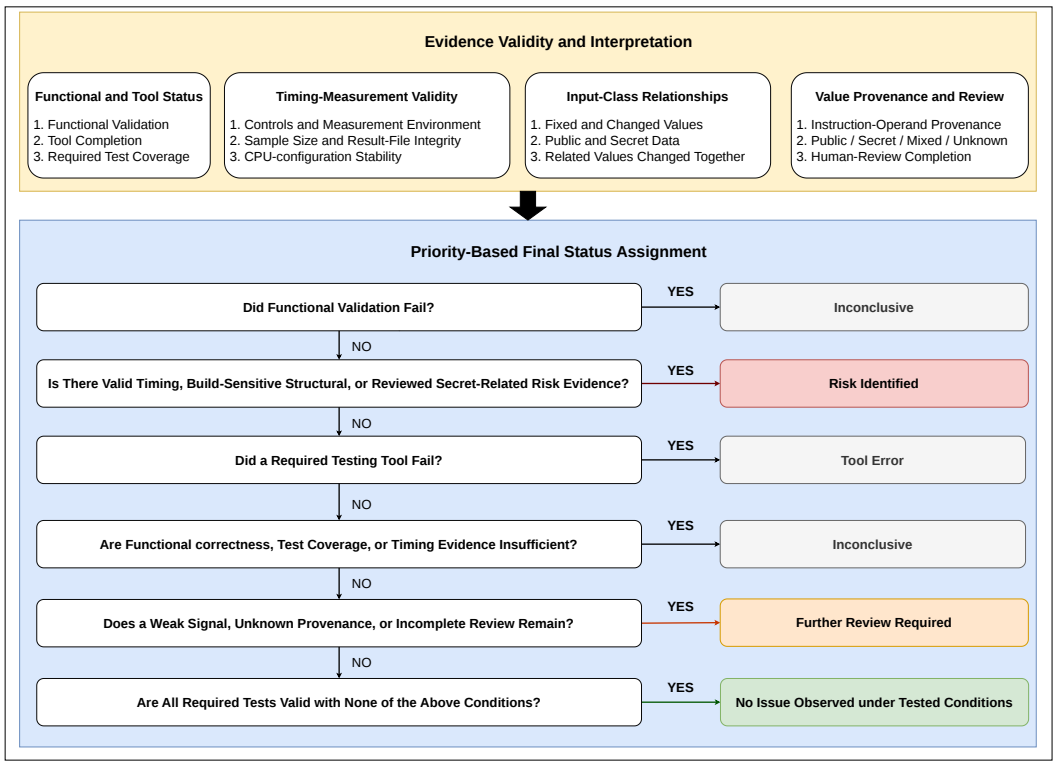


Figure 4. Evidence validation and priority-based final status assignment in Stages 4 and 5.

4.6. Final Status Assignment and Result Preservation (Stage 5)

This subsection corresponds to Stage 5 in Figure 1. After the preceding stages establish the meaning and validity of each item of evidence, the final stage combines measurement validity, input conditions, and human-review status. Rather than copying an external tool's pass or fail result, the decision rules incorporate both the results of every stage and any remaining uncertainty.

Table 2 defines the final statuses used by **CT-KAT** in human-readable terms. "No Issue Observed under Tested Conditions" is not a proof of safety; it means only that no new risk signal was observed for the tested binary, inputs, system, and analysis scope.

Table 2. Final result statuses of CT-KAT.

Status	Meaning	Does Not Mean
Risk Identified	A valid test exceeds its risk threshold, or reviewed risk evidence exists	Exploitable in every environment
Further Review Required	A suspicious result remains, but its public/secret relationship or human review is incomplete	Confirmed safe or vulnerable
Inconclusive	Functional behavior, environment, sample size, or controls are insufficient	No timing difference exists
Tool Error	A testing tool does not terminate normally	The target program is safe
No Issue Observed under Tested Conditions	Required tests are valid and no defined risk signal appears	Constant-time behavior is proven

Status definitions alone do not resolve cases in which several conditions hold simultaneously. **CT-KAT** therefore applies the decision procedure in Algorithm 1. Each condition is checked from top to bottom, and the first applicable status becomes the final result.

Algorithm 1 Priority-based final status assignment

Require: Evidence collected for one target–operation pair
Ensure: A primary status together with the evidence flags F
$F \leftarrow$ outcome of each evidence layer (functional, structural, instruction, timing, review) and the input relationship of each timing contrast
if functional validation fails **then**
    return (Inconclusive, F)
end if
if valid timing or build-sensitive structural risk evidence exists **then**
    return (Risk Identified, F)
end if
if reviewed secret-related risk evidence exists **then**
    return (Risk Identified, F)
end if
if a required testing tool fails **then**
    return (Tool Error, F)
end if
if correctness is unchecked or required coverage is incomplete **then**
    return (Inconclusive, F)
end if
if the timing environment, sample size, or controls are insufficient **then**
    return (Inconclusive, F)
end if
if a weak timing signal or unknown value provenance remains **then**
    return (Further Review Required, F)
end if
if human review is incomplete or invalid **then**
    return (Further Review Required, F)
end if
return (No Issue Observed under Tested Conditions, F)

The order of these checks follows one principle. A risk status needs only one valid item of risk evidence, whereas a no-issue status needs every required item to be present and valid. Functional validation therefore comes first, because a binary that does not work correctly says nothing about the target. Risk evidence comes next, because a valid signal from one test should not be hidden by a failure or gap in another test. Tool errors, missing

evidence, and pending reviews follow, and a no-issue status is assigned only when none of the earlier conditions applies.

Under this priority order, "Risk Identified" means that the observed evidence is valid; it does not prove immediate leakage of secret information. For example, a valid timing difference between classes that change both public and secret values remains risk evidence, but its cause is interpreted only within that mixed-input condition. Conversely, even when a suspicious instruction is shown to process public data, a no-issue status is available only after all other required tests and reviews are valid. The final status is therefore not a summary of one test statistic but an account of which requirements are satisfied and which questions remain.

Algorithm 1 returns this account in the evidence flags F that accompany the primary status. When the status rests on a timing difference, F also states whether the contrast was secret-only, public-only, or mixed, and only a secret-only contrast can support secret attribution. Tables A1 and A2 list the evidence flags next to each final status.

5. Experimental Evaluation

The evaluation examines whether **CT-KAT** connects evidence from heterogeneous tests in one workflow while limiting each interpretation to its test conditions. We first define the targets and shared experimental conditions and then measure how the review set changes when the build matrix is expanded. We next assign final statuses and use KyberSlash and timing experiments to illustrate how different evidence is interpreted. Finally, we apply the same record structure to projects with different implementation styles and to existing testing tools.

5.1. Targets and Experimental Environment

We divide the targets into a core status set and additional applicability targets. The core set contains 25 target–operation pairs: ML-KEM, ML-DSA, SLH-DSA, and Falcon implementations from PQClean [24]; the Kyber reference implementation; c-fn-dsa; and four controls. We use mlkem-native and mldsa-native as additional comparisons to examine whether the workflow transfers across implementation styles, and we also test integration with OpenSSL 3.5. After separating these roles, we apply one status-assignment policy to all 25 core pairs.

Structural and binary tests cover the 24 pairs for which build artifacts exist. Across their compiler and optimization settings, these pairs yield 206 target-specific build combinations. The remaining pair, an ML-KEM-768 comparison that varies a public ciphertext, is timing-only and is therefore not included in the build count. It remains a separate evidence unit for final status assignment, so the core evaluation still contains 25 pairs.

The base build matrix, shown in Table 3, combines two compilers with five optimization levels for ten conditions. When an implementation supports only one compiler or cannot be built under a particular setting, we retain only applicable conditions. ML-DSA also receives an additional condition that makes the location of findings easier to inspect. Accounting for these target-specific differences produces the 206 combinations above.

Table 3. Build conditions used for structural and binary tests.

Compiler	Optimization Settings	Application
GCC 13.3.0	`-O0`, `-O1`, `-O2`, `-O3`, `-Os`	Base conditions
Clang 18.1.3	`-O0`, `-O1`, `-O2`, `-O3`, `-Os`	Base conditions
GCC and Clang	`-O2`	Additional condition for ML-DSA finding locations

We select GCC 13.3.0 with `-O2` as the single deployment build because it applies to all 24 targets, approximates deployment more closely than an unoptimized diagnostic build, and provides a common comparison point. We compare this condition with the full matrix to determine whether broader build coverage reveals additional targets.

Separately from the build analysis, we perform timing measurements for 28 conditions. To reduce unrelated variation, the process is pinned to one logical CPU, while simultaneous multithreading and automatic frequency boosting are disabled. Table 4 reports the hardware and runtime environment.

Table 4. Hardware and runtime environment for timing evaluation.

Item	Environment and Setting
Processor	Intel Core i7-10875H CPU @ 2.30 GHz
Architecture and cores	x86_64, 8 physical cores, 16 logical processors
Cache	L1 data 256 KiB, L1 instruction 256 KiB, L2 2 MiB, L3 16 MiB
Operating system	Ubuntu 24.04.4 LTS, Linux kernel 6.17.0-19-generic
Execution platform	Bare-metal system; no virtualization or emulation
Compiler and clock source	GCC 13.3.0, RDTSC
CPU controls	Pinned to one logical CPU, performance governor, SMT and Turbo Boost disabled

Each condition is measured 30,000 times in each of three independent processes, yielding 90,000 primary samples. In each process, the comparison of equivalent inputs, the comparison of the preparation code alone, and each of three injected delays use 10,000 samples for key establishment and decapsulation and 5000 for most signing operations. The longer-running SLH-DSA condition uses 2000. The injected delays form a three-step ladder chosen per target, from 64 to 16,384 ticks for the fast KEM and operand targets and up to 1,048,576 ticks for the slowest signing targets. Table A3 lists the ladder of every condition.

Welch's t-statistic indicates whether the difference between class means is large relative to measurement variation, and Holm correction reduces the chance of incidental findings when several results are compared. We analyze each process independently. Following the official **dudect** procedure, we compute 102 tests: one on the uncropped samples, 100 that progressively exclude large measurements, and a second-order test for variance differences. Among tests meeting the minimum sample requirement, the largest $|t|$ becomes the result for that process, and by the **dudect** convention, $|t| > 10$ is treated as a timing signal.

A timing result is valid only when all three processes agree on pass or fail and all controls and environmental checks pass. We report the process with the largest $|t|$ as the representative value. Holm correction applies only to the secondary cross-implementation comparisons: for these, we apply a two-sided Welch's test to the class-mean differences from the three processes and adjust the p-values within each preregistered comparison family. Because the threshold is fixed in advance, we also report how the classification would change under other thresholds. For each condition, we further report the minimum detectable difference $(10 + z_{0.80}) \cdot \mathrm{SE}$, the smallest class-mean difference that the $|t| > 10$ rule detects with a probability of 0.80, where SE is the observed standard error of the class-mean difference, together with the dudect estimate $(5/\tau)^2$. These quantities serve the same purpose as recent work on the statistical power of software timing tests [25], but we use them only to state the sensitivity of each result, not as a different test. With these common conditions and interpretation rules in place, we first examine the effect of expanding the build matrix.

5.2. Effect of Multiple Build Conditions

The same source can produce different control flow and instructions under different compilers and optimization settings. A test limited to one deployment build may therefore miss signals present elsewhere. We compare the number of review targets under the single build and full matrix in Table 5.

Table 5. Change in review targets as test coverage expands.

Test Scope	Build Combinations	Structural Targets	All Review Targets
Single deployment build	24	11	11
Full build matrix	206	13	13
Full matrix and instruction scan	206	13	22

Under the single deployment build, 11 of the 24 target–operation pairs require structural review. The full set of 206 build combinations increases this count to 13, adding two build-sensitive pairs. Including the suspicious-instruction scan raises the total review set to 22. Multiple build conditions and binary inspection therefore contribute evidence absent from the single-build structural test.

These counts represent candidates for human review, not confirmed vulnerabilities. They may include instructions that process only public data or behavior permitted by the algorithm. **CT-KAT** retains the build condition and evidence associated with each candidate and routes previously hidden targets to follow-up review. Candidate counts are therefore not treated as final outcomes; they are combined with other evidence when assigning target statuses.

The structural test, however, runs one input per harness and does not examine paths that this input does not execute. Measured over the target-operation call alone, the input reaches 92–99% of the lines and 68–95% of the branches in the executed functions, as shown in Table 6. A structural no-signal result therefore holds only for the executed paths, which matters most for the c-fn-dsa pairs, whose branch coverage is about 70%.

Table 6. Coverage of the structural-test operation call. Each structural harness was rebuilt with GCC coverage instrumentation (`-O0`); counters were reset immediately before the call to the target operation and recorded immediately after it, so input preparation is excluded. Lines and branches are counted within the functions of the target sources that the operation executes.

No.	Target and Test	Functions	Lines (%)	Branches (%)
1	PQClean ML-KEM-512, decapsulation	70	98.1	91.3
2	PQClean ML-KEM-512, internal comparison	70	98.1	91.3
3	PQClean ML-KEM-768, key-and-ciphertext comparison	68	98.1	91.1
4	PQClean ML-KEM-768, ciphertext-only comparison	structural test not run		
5	PQClean ML-KEM-768, internal comparison	68	98.1	91.1
6	PQClean ML-KEM-1024, decapsulation	68	98.6	92.6
7	PQClean ML-KEM-1024, internal comparison	68	98.1	91.2
8	Input-dependent-work control	1	100.0	100.0
9	Input-independent-work control	1	100.0	100.0
10	Build-sensitive control	1	100.0	100.0
11	Build-independent control	1	100.0	100.0
12	PQClean ML-KEM-768, KyberSlash KS1	68	98.5	92.5
13	PQClean ML-KEM-768, KyberSlash KS2	68	98.5	92.5
14	PQClean ML-KEM-768, KyberSlash KS1/KS2	68	98.0	91.1

Table 6. *Cont.*

No.	Target and Test	Functions	Lines (%)	Branches (%)
15	Kyber reference, decapsulation	58	98.8	94.6
16	PQClean Falcon-512, signing	55	97.5	83.3
17	PQClean Falcon-1024, signing	55	97.5	83.3
18	c-fn-dsa-512, native-arithmetic signing	51	91.8	68.2
19	c-fn-dsa-512, integer-emulated signing	64	94.0	71.9
20	c-fn-dsa-1024, native-arithmetic signing	51	91.8	68.2
21	c-fn-dsa-1024, integer-emulated signing	64	94.0	71.9
22	PQClean ML-DSA-44, signing	76	97.4	89.7
23	PQClean ML-DSA-65, signing	76	97.3	89.7
24	PQClean ML-DSA-87, signing	76	96.6	87.3
25	PQClean SLH-DSA-SHA2-128f, signing	44	96.8	87.1

5.3. Status Distribution of the Screening Set

Candidate counts alone do not determine the final status of each target. A candidate may process only public data, while an absence of signals may coincide with insufficient measurement evidence. We therefore classify the 25 target–operation pairs by combining functional correctness, execution structure, suspicious instructions, measurement validity, and human-review status.

We validate functional behavior for the 21 pairs drawn from cryptographic implementations and use four controls to distinguish known problematic and expected behavior. Table 7 summarizes the status counts; Appendix A Tables A1 and A2 report the individual pairs.

Table 7. Final statuses of the screening set.

Result Status	Target–Operation Pairs
Risk Identified	7
Further Review Required	10
Inconclusive	2
No Issue Observed under Tested Conditions	6

The resulting distribution contains seven Risk Identified, ten Further Review Required, two Inconclusive, and six No Issue Observed under Tested Conditions pairs; none ends in Tool Error. The timing evidence for these pairs comes from the eight core-implementation comparisons in Table 8. Importantly, results that require additional review and results that lack sufficient evidence remain separate rather than being absorbed into the no-issue category.

Six of the seven Risk Identified pairs have a known risk. They are the three KyberSlash variants, the Kyber reference implementation, and the two controls with a planted risk. The remaining pair is the PQClean ML-KEM-768 comparison that changes the key and the ciphertext together. Its timing difference is clear, but because the comparison changes public and secret values at once, the difference cannot be attributed to the secret, and Section 5.5 shows that it actually comes from the public key stored in the secret key. The two Inconclusive pairs are the negative controls. We deliberately did not validate their functional behavior, and Algorithm 1 never gives a no-issue status to a binary whose behavior has not been checked. Two of the six No Issue Observed pairs are the other ML-KEM-768 comparisons, which change only the ciphertext or isolate part of decapsulation and show no timing difference. The other four, for ML-KEM-512 and ML-KEM-1024, were not part of the timing measurements, so their status rests on the functional, structural, and reviewed instruction evidence alone.

Ten pairs remain at Further Review Required because their findings require further manual review. For Falcon-1024 and the four c-fn-dsa pairs, the structural findings have not yet been reviewed, and for Falcon-1024, the attribution of its suspicious instructions is also open. Falcon-512, the three ML-DSA parameter sets, and SLH-DSA show no timing difference in valid measurements, but the reviews of their structural findings and suspicious instructions are pending or have expired. **CT-KAT** does not interpret the absence of an observed timing difference as a clean result and keeps these pairs at Further Review Required until the reviews are complete.

To quantify what the status model adds, we also compared it with two naive aggregations of the same evidence. A rule that labels a pair vulnerable whenever Valgrind or the timing test reports a problem, and safe otherwise, gives 15 of the 25 pairs a label that the recorded evidence does not support. It labels safe the three KyberSlash variants, whose division operands were reviewed as secret-derived, and the two negative controls, whose functional behavior was never checked, and it labels vulnerable the ten Further Review Required pairs before their findings are reviewed. A rule that also counts every suspicious instruction as a problem no longer misses the KyberSlash variants, but it labels vulnerable five pairs whose candidates were attributed to public data and still labels the unvalidated controls safe, so 17 pairs disagree with the recorded evidence. **CT-KAT** instead reports the same evidence in four statuses that reflect what was actually established. The following KyberSlash case shows more concretely how this status model preserves apparently conflicting evidence.

5.4. KyberSlash: Coexisting Results from Different Tests

KyberSlash shows that timing risk may arise when a secret-related value reaches division even if branches and memory accesses do not change. It is therefore well suited to testing whether the evidence categories above complement one another in practice. We compare four implementations: one in which the known divisions are removed and three in which the divisions are reintroduced.

The branch-and-memory test finds no secret-dependent difference in any of the four implementations. Binary inspection, however, finds the expected instructions in all three variants containing the reintroduced divisions. The auxiliary provenance test identifies a secret-related operand at one KS1 location and at two KS2 locations. In particular, KS2 produces no warning when the entire decapsulation path is tested, but both locations appear when the division sites are examined directly. The broader execution test and instruction-specific test thus contribute different evidence.

We next measure each implementation on the x86-64 host to test whether a timing difference is observable there. This experiment is not expected to reproduce the KyberSlash attack: the published attacks were demonstrated on Arm Cortex-A7 and Cortex-M4 platforms, and the x86-64 host serves to test whether the stages of **CT-KAT** keep their evidence separate, not whether the vulnerability manifests on this processor. The primary KyberSlash set within the 28-condition timing evaluation contains four whole-decapsulation and six division-site comparisons. None of these ten exceeds the threshold. All ten timed binaries were disassembled before sampling. In the `-Os` builds, the KS1 binary contains one `div` in `poly_tomsg`, the KS2 binary contains one `idiv` in `poly_compress` and one `div` in `polyvec_compress`, the combined binary contains all three, and the unmodified binary contains none at these sites. Each division-site binary likewise contains one `div` in its vulnerable form and none in its patched form. The smallest class-mean difference that each comparison could detect with a probability of 0.80 under the $|t| > 10$ rule was about 4500 cycles for the whole-decapsulation comparisons, far larger than the latency of a single division, and between 1 and 355 cycles for the division-site comparisons. Three additional

paired comparisons between affected and fixed implementations are secondary and are not counted among the 28 primary conditions. Table 9 summarizes the evidence and its meaning, focusing on the primary set.

Table 8. Twenty-eight timing comparisons.

Evaluation Group	Count	Timing Difference	No Difference Above Threshold
Core algorithm implementations	8	1	7
KyberSlash comparisons	10	0	10
Falcon/FN-DSA comparisons	6	0	6
mlkem-native/mldsa-native comparisons	4	2	2
Total	28	3	25

Table 9. Evidence from each testing stage in the KyberSlash case.

Test Focus	Observation	Interpretation
Functional behavior	Same result across all four variants	Comparable functional behavior
Branches and memory accesses	No secret-dependent difference	Division latency not covered
Division instructions	Secret-related division operands in three variants	Risk evidence remains
Actual execution time	No above-threshold difference in ten comparisons	Not expected on this host

These results make clear that the stages answer different questions. The branch-and-memory test asks whether execution structure depends on secrets; binary inspection asks whether divisions with potential latency variation remain; and timing measurement asks whether a difference is observable for a specific system and input set. Even though the structural test reports no difference, **CT-KAT** retains the binary and provenance evidence. The case therefore confirms that one stage does not erase evidence produced by another.

Three other conditions in the full timing evaluation produce repeatable signals. The next subsection interprets these signals together with their input conditions.

5.5. Timing Results and the Effect of Input Conditions

An execution-time difference alone does not identify the input responsible for it. Before interpreting a result, **CT-KAT** therefore checks functional behavior, controls, system state, and result-file integrity under predefined rules. The user specifies which values are fixed or changed between input classes, and the framework records this relationship for separate review. All 28 comparisons pass these validity checks and can therefore be compared under one standard; Table 8 summarizes the results.

Three of the 28 conditions show repeatable timing differences, whereas the other 25 show no above-threshold difference. The two groups are clearly separated: the largest statistic among the 25 is approximately 2.87, while the smallest statistic among the three signals is approximately 101. The evaluation treats values above 10 as timing signals. Each value is the maximum over 102 tests, so it can exceed the statistic of a single test even when no difference exists. The comparison of equivalent inputs and the comparison of the preparation code alone contain no true difference and undergo the same procedure. Their $|t|$ did not exceed 3.79 in any of the 28 conditions (Table A3). Selecting the maximum therefore did not bring any comparison without a true difference close to the threshold of 10. For the same reason, the classification is insensitive to the choice of threshold. Any

threshold between 2.87 and 101, such as 5, 20, or 50, yields the same classification of all 28 conditions.

All three signals occur in ML-KEM comparisons. These experiments change not only the secret key but also the corresponding public ciphertext and public information embedded in the key object. All three processes show a difference in the same direction. Table 10 reports the detailed results.

Table 10. ML-KEM comparisons that change both key and ciphertext.

Implementation	$\lvert t \rvert$	Mean Cycle Difference
PQClean ML-KEM-768	256.933	approx. 8725
mlkem-native portable C	351.502	approx. 11,361
mlkem-native optimized	100.959	approx. 378

All three statistics greatly exceed the threshold of 10, so the timing differences themselves are clear. However, these mixed-input comparisons alone cannot identify which value causes the difference. **CT-KAT** records each result as a difference between classes that jointly change public and secret values, not as an effect of the secret key alone. Comparisons on the same PQClean implementation that vary only the public ciphertext or isolate part of decapsulation show no above-threshold difference.

To find the source of these differences, we ran a separate follow-up experiment on the same host. For each of the three implementations, we varied one component of the input at a time over 20,000 interleaved decapsulations and counted how many Keccak-f[1600] permutations each decapsulation executed.

When only the secret part of the key (s and z) or only the ciphertext varied, the difference disappeared in all three implementations, and the permutation count did not change. When only the public key stored in the secret-key object varied, the difference seen when the whole input varied appeared again. As Table A4 shows for mlkem-native portable C, varying only this public key gave a difference of 4106 cycles, close to the 4152 cycles obtained when the whole input varied. This was also the only case in which the permutation count changed.

The cause lies in the re-encryption step of decapsulation. This step regenerates the public matrix from the public seed ρ by rejection sampling, and the amount of rejection-sampling work depends on ρ. Comparing two fixed tuples leaves only this effect, and the resulting difference equaled the difference in permutation count multiplied by the cost of one permutation. When a fresh public key was used in every run, the two implementations that perform rejection sampling in scalar code showed an additional cost, whereas the vectorized x86-64 implementation did not. This is consistent with branch-prediction effects in the scalar rejection loop.

The timing differences in Table 10 therefore come from public data rather than from the secret key, and constant-time requirements do not prohibit such a dependence.

For signing, a variable output length may change the amount of data processed, so we also examine whether signature length explains timing. We apply the same interpretation principle to the 13 signature comparisons among the 28 conditions. Three PQClean Falcon conditions permit variable signature lengths, but the relationship between length and time is very small; the other ten use fixed lengths. Signature length is therefore not a major explanatory factor in these data.

Overall, timing analysis identifies three repeatable differences, while the recorded input relationships and signature lengths constrain how each result can be interpreted. **CT-KAT** thus presents not only the magnitude of a timing difference but also the conditions

that produced it. We next expand the target set to determine whether this workflow persists across implementation styles.

5.6. Application to Multiple Implementations

To examine whether the core findings depend on one reference implementation, we apply the same workflow to the projects introduced in the evaluation setup. We check functional behavior and buildability for every project and measure timing where possible. For OpenSSL 3.5, we invoke ML-KEM and ML-DSA through the library's public interface to test integration with a deployed library.

As Table 11 shows, four projects support the same format for functional and build results, while OpenSSL supports integration through an actual library call. The configuration and result format of **CT-KAT** are therefore not tied to a single reference implementation. The remaining question is whether results from different existing tools can likewise share one record without losing their individual meanings.

Table 11. Application of CT-KAT by project.

Target	Functional Check	Code/Build Test	Timing	Role in Evaluation
PQClean	Confirmed	Performed	Performed	Core screening and scheme-level evaluation
c-fn-dsa	Fixed vectors confirmed	Performed	Performed	Falcon-family comparison
mlkem-native	Confirmed	Build conditions checked	Performed	Portable/optimized implementation comparison
mldsa-native	Confirmed	Build conditions checked	Performed	Portable/optimized implementation comparison
OpenSSL 3.5	Public interface confirmed	Integration checked	Not performed	Deployed-library integration case

5.7. Integration of Existing Tool Results

Integration means applying multiple tools to the same target while recording separately what each tool checks and under which conditions it runs. To evaluate this function, we prepare one example whose execution path and work vary with its input and another that preserves both regardless of input. These are tool-integration cases separate from the four controls in the 25-pair screening set. We run the official dudect implementation, TIMECOP, and MicroWalk PinTracer on both examples under the same input conditions and store each output as a distinct **CT-KAT** evidence record. Table 12 summarizes the observations, control results, and execution constraints of the three tools.

Table 12. Existing tools on the same control examples.

Tool	Observation	Control Results	Main Constraint
Official dudect	Timing distributions	Problem: difference Normal: no difference	Linux/x86-64; controlled CPU/timing
TIMECOP	Secret-dependent behavior	Problem: difference Normal: no difference	Modified Valgrind
MicroWalk PinTracer	Execution and memory traces	Problem: difference Normal: no difference	Linux/x86-64; Intel Pin

All three tools find a difference in the problem case and report none in the normal case. Although the outcomes look identical, the observations differ: dudect records timing distributions, TIMECOP records the influence of secrets on program behavior, and Mi-

croWalk records execution-path and memory-access differences. **CT-KAT** preserves these as separate evidence rather than collapsing them into one pass/fail result. The experiment shows that existing tools can share the same input conditions and review workflow while retaining the meaning of each result.

Across the evaluation, expanding one build to the full matrix increases structural-review targets from 11 to 13, and adding instruction scanning expands the overall review set to 22. In KyberSlash, binary inspection retains risk evidence absent from structural analysis. For the three timing signals, recorded input relationships constrain the scope of interpretation.

Four implementation projects support the same format for functional and build evidence, and the OpenSSL case reaches an actual library interface. The three-tool comparison likewise preserves each tool's observation and execution constraints in a joint record. These results show that linking heterogeneous evidence to common targets and conditions operates consistently across the evaluation.

6. Conclusions

This paper presents **CT-KAT**, which links results from multiple constant-time tests to the conditions that produced them and supports decisions bounded by the available evidence. The evaluation shows that **CT-KAT** preserves signals that vary across build conditions and testing methods together with input relationships, measurement validity, and pending review, thereby avoiding both overstatement and oversimplification.

These findings must be interpreted within the scope of the evaluation. Timing measurements use one x86-64 Intel system and cannot be generalized to other processors; in particular, they were not taken on the Arm platforms on which the KyberSlash attacks were demonstrated, so they provide no evidence for or against that vulnerability. Each structural test executes one input, which reaches 68–95% of the branches in the executed functions, so a structural no-signal result does not cover untaken paths. Binary inspection currently focuses on division and remainder instructions, so it does not cover other instructions whose latency depends on their operands on some architectures, such as multiplication on some Arm cores or floating-point operations on subnormal values.

Future work will repeat timing experiments on multiple x86 and ARM systems and expand the target set to additional production libraries and deployment environments. Comparing **TIMECOP**, **MicroWalk**, and other tools on the same corpus will allow measurement of detection rate, false-positive rate, runtime, result-file size, manual-review effort, and execution constraints. Because the instructions whose latency depends on their operands differ across microarchitectures, a broader scan requires a per-architecture table of variable-latency instructions rather than a fixed list of mnemonics. Combining such architecture-aware instruction coverage with automated taint-based provenance tracking can further reduce manual review and the reviewer-dependent variation it introduces. Future work will also introduce consistency checks for the declared secret regions, target operations, and build settings and evaluate the robustness of **CT-KAT** under deliberately incomplete or incorrect secret annotations. With these extensions, **CT-KAT** can develop not as a substitute for proving constant-time security, but as a framework for accumulating reproducible evidence and supporting decisions grounded in sufficient evidence.

Author Contributions: Conceptualization, S.-W.L. and H.-J.S.; methodology, S.-W.L. and H.-J.S.; software, investigation, S.-W.L., M.-S.K. and S.-M.J.; writing—original draft preparation, S.-W.L., M.-S.K. and S.-M.J.; writing—review and editing, S.-W.L., M.-S.K. and S.-M.J.; supervision, H.-J.S.; project administration, H.-J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02306395, Development and Demonstration of PQC-Based Joint Certificate PKI Technology, 25%) and this work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-25394739, Development of Security Enhancement Technology for Industrial Control Systems Based on S/HBOM Supply Chain Protection, 25%) and this work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2026-25519543, Development of PQC Migration Technology for Quantum-Safe IC Chip, 50%).

Data Availability Statement: The CT-KAT source code, configurations, measurement results, and analysis scripts used in this study are available at https://github.com/Lee-Seungwon1215/CT-KAT, accessed on 30 September 2026.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Detailed Results and Review Rules

Tables A1 and A2 provide the evidence flags underlying the status counts in Table 7.

Table A1. Final statuses and evidence flags for pairs 1–15. F: functional validation (P, pass); S: structural test (F, finding; N, no finding); I: suspicious-instruction scan and operand attribution (N, no candidate; Pub, public; Sec, secret; Mix, mixed; Unr, unresolved); T: timing (S, signal; N, no signal; ^{m}, mixed-input contrast); R: review (Rev, reviewed; Pend, pending; Exp, expired); –, not run or not needed.

No.	Target and Test	F	S	I	T	R	Final Status
1	PQClean ML-KEM-512, decapsulation	P	N	Pub	–	Rev	No Issue Observed under Tested Conditions
2	PQClean ML-KEM-512, internal comparison	P	N	Pub	–	Rev	No Issue Observed under Tested Conditions
3	PQClean ML-KEM-768, key-and-ciphertext comparison	P	N	Pub	S^{m}	Rev	Risk Identified
4	PQClean ML-KEM-768, ciphertext-only comparison	P	–	–	N	Rev	No Issue Observed under Tested Conditions
5	PQClean ML-KEM-768, internal comparison	P	N	Pub	N	Rev	No Issue Observed under Tested Conditions
6	PQClean ML-KEM-1024, decapsulation	P	N	Pub	–	Rev	No Issue Observed under Tested Conditions
7	PQClean ML-KEM-1024, internal comparison	P	N	Pub	–	Rev	No Issue Observed under Tested Conditions
8	Input-dependent-work control	–	F	N	–	Rev	Risk Identified
9	Input-independent-work control	–	N	N	–	–	Inconclusive
10	Build-sensitive control	–	F	N	–	–	Risk Identified
11	Build-independent control	–	N	N	–	–	Inconclusive
12	PQClean ML-KEM-768, KyberSlash KS1	P	N	Sec	–	Rev	Risk Identified
13	PQClean ML-KEM-768, KyberSlash KS2	P	N	Sec	–	Rev	Risk Identified
14	PQClean ML-KEM-768, KyberSlash KS1/KS2	P	N	Sec	–	Rev	Risk Identified
15	Kyber reference, decapsulation	P	F	Sec	–	Rev	Risk Identified

Table A2. Final statuses and evidence flags for pairs 16–25. Abbreviations follow Table A1.

No.	Target and Test	F	S	I	T	R	Final Status
16	PQClean Falcon-512, signing	P	F	Unr	N	Pend	Further Review Required
17	PQClean Falcon-1024, signing	P	F	Unr	–	Pend	Further Review Required
18	c-fn-dsa-512, native-arithmetic signing	P	F	N	–	Pend	Further Review Required
19	c-fn-dsa-512, integer-emulated signing	P	F	N	–	Pend	Further Review Required
20	c-fn-dsa-1024, native-arithmetic signing	P	F	N	–	Pend	Further Review Required
21	c-fn-dsa-1024, integer-emulated signing	P	F	N	–	Pend	Further Review Required
22	PQClean ML-DSA-44, signing	P	F	Sec	N	Exp	Further Review Required
23	PQClean ML-DSA-65, signing	P	F	Sec	N	Exp	Further Review Required
24	PQClean ML-DSA-87, signing	P	F	Sec	N	Exp	Further Review Required
25	PQClean SLH-DSA-SHA2-128f, signing	P	F	Mix	N	Exp	Further Review Required

Table A3. Timing sensitivity for the 28 conditions in Table 8. Delays: injected positive controls; Inputs/Setup: equivalent-input/setup-only controls. Null-control $|t|$ and minimum detectable difference (MDD, cycles) are maxima across three processes. $(5/\tau)^2$ estimates the sample count for $|t| = 5$ at the observed effect. MDD and test procedures follow Section 5.1.

			Null Controls			
Group	**Target (Input Axis)**	**Delays (Ticks)**	$\|t\|$	**Inputs**	**Setup**	$(5/\tau)^2$	**MDD**
Core	PQClean Falcon-512 (secret key)	512/8192/131,072	0.99	1.69	1.48	7.07×10^5	38,420
Core	PQClean ML-DSA-44 (secret key)	65,536/262,144/1,048,576	2.01	3.79	2.97	1.29×10^5	3.46×10^5
Core	PQClean ML-DSA-65 (secret key)	65,536/262,144/1,048,576	1.82	0.10	0.72	1.53×10^5	5.57×10^5
Core	PQClean ML-DSA-87 (secret key)	65,536/262,144/1,048,576	2.35	1.55	2.07	91,596	5.94×10^5
Core	PQClean ML-KEM-768 (key and ciphertext (mixed))	64/512/16,384	256.93	0.96	1.89	9	682
Core	PQClean ML-KEM-768 (ciphertext only)	64/512/16,384	1.64	0.88	0.88	2.40×10^5	817
Core	PQClean ML-KEM-768 (internal)	64/512/16,384	2.22	2.50	1.28	1.07×10^5	700
Core	PQClean SLH-DSA-SHA2-128f (secret key)	4096/65,536/1,048,576	1.76	2.29	0.54	2.21×10^5	83,069
KyberSlash	KS1 site, patched (operand bin)	64/512/16,384	1.69	1.29	1.68	1.81×10^5	111
KyberSlash	KS1 site, vulnerable (operand bin)	64/512/16,384	2.13	1.38	1.07	1.34×10^5	111
KyberSlash	KS2 poly site, patched (operand bin)	64/512/16,384	1.15	0.64	1.52	5.46×10^5	1
KyberSlash	KS2 poly site, vulnerable (operand bin)	64/512/16,384	2.71	1.43	1.02	99,615	68
KyberSlash	KS2 polyvec site, patched (operand bin)	64/512/16,384	1.42	1.11	1.01	3.64×10^5	355
KyberSlash	KS2 polyvec site, vulnerable (operand bin)	64/512/16,384	1.93	1.08	1.39	1.35×10^5	331
KyberSlash	PQClean ML-KEM-768 (chosen ciphertext)	64/512/16,384	2.45	1.61	1.49	1.25×10^5	4465
KyberSlash	ML-KEM-768 with KS1 and KS2 (chosen ciphertext)	64/512/16,384	2.30	1.63	2.30	1.42×10^5	4544
KyberSlash	ML-KEM-768 with KS1 (chosen ciphertext)	64/512/16,384	1.89	2.02	1.33	2.09×10^5	4465
KyberSlash	ML-KEM-768 with KS2 (chosen ciphertext)	64/512/16,384	1.69	1.15	0.26	2.61×10^5	4496
Falcon/FN-DSA	c-fn-dsa-1024, integer-emulated (secret key)	65,536/262,144/1,048,576	1.68	0.95	2.35	2.65×10^5	4.09×10^5
Falcon/FN-DSA	c-fn-dsa-1024, native arithmetic (secret key)	512/8192/65,536	1.48	1.24	1.71	3.42×10^5	32,162
Falcon/FN-DSA	c-fn-dsa-512, integer-emulated (secret key)	512/8192/131,072	2.87	0.98	0.38	91,334	84,111
Falcon/FN-DSA	c-fn-dsa-512, native arithmetic (secret key)	512/8192/65,536	1.64	2.46	1.10	2.78×10^5	12,017
Falcon/FN-DSA	PQClean Falcon-1024 (secret key)	512/8192/131,072	1.81	1.28	1.17	1.52×10^5	49,837
Falcon/FN-DSA	PQClean Falcon-512 (secret key)	512/8192/131,072	2.15	0.50	0.58	1.62×10^5	40,006
mlkem/mldsa-native	mldsa-native-65, portable (secret key)	65,536/262,144/1,048,576	1.99	0.03	0.72	1.27×10^5	3.93×10^5
mlkem/mldsa-native	mldsa-native-65, optimized (secret key)	65,536/262,144/1,048,576	2.20	0.09	0.69	1.04×10^5	91,476
mlkem/mldsa-native	mlkem-native-768, portable (key and ciphertext (mixed))	64/512/16,384	351.50	1.83	2.30	5	581
mlkem/mldsa-native	mlkem-native-768, optimized (key and ciphertext (mixed))	64/512/16,384	100.96	0.83	1.21	68	359

Table A4. Exploratory root-cause follow-up for ML-KEM (Section 5.5; separate from Table 8). Class 0 uses a fixed valid tuple; class 1 cycles through 64 variants of only the listed component. Each row uses 20,000 interleaved decapsulations, excluding the slowest 5%. Δ and Δ_{perm} are class 1 minus class 0 differences in mean cycles and Keccak-f[1600] calls per decapsulation, respectively (four-way calls for the optimized build). Magnitudes depend on the fixed tuple.

Implementation	Varied Component	Δ (Cycles)	t	Δ_{perm}
PQClean ML-KEM-768	Whole tuple (as in Table 10)	2810	32.9	−0.89
	Secret part only (s, z)	16	0.2	0.00
	Embedded public key only	3081	37.4	−0.89
	Ciphertext only	138	1.6	0.00
	Another fixed tuple	−1143	−16.6	−1.00
mlkem-native-768, portable	Whole tuple (as in Table 10)	4152	84.2	−3.61
	Secret part only (s, z)	72	1.4	0.00
	Embedded public key only	4106	79.3	−3.61
	Ciphertext only	−41	−1.1	0.00
	Another fixed tuple	−3133	−77.7	−4.00
mlkem-native-768, optimized	Whole tuple (as in Table 10)	−762	−217.5	−0.91
	Secret part only (s, z)	0	−0.2	0.00
	Embedded public key only	−726	−168.2	−0.91
	Ciphertext only	18	0.9	0.00
	Another fixed tuple	−880	−469.4	−1.00

Table A5. Manual provenance-propagation rules. Unknown corresponds to Unr; parenthesized numbers identify pairs in Tables A1 and A2.

Rule	Statement	Example in This Evaluation
1. Secret source	Long-term secret-key components, signing randomness, and state computed from them are secret.	The operands of the KyberSlash divisions are polynomial coefficients derived from the secret key (#12–14).
2. Public source	The public key, the ciphertext, the message, constants, and values computed only from them are public.	The SHAKE divisions in ML-KEM use only the Keccak rate constant (#1–3, #5–7).
3. Operation	The result of an operation is secret if any operand is secret and is public only if all operands are public.	In KyberSlash, dividing a numerator computed from a secret coefficient by q yields a secret value.
4. Memory and control	A value loaded from a location that holds a secret, and a value set inside a branch whose condition is secret, are secret.	–
5. Declassification	A secret-derived value becomes public only when the algorithm includes it in its public output. Permission by the specification or a later successful signature does not declassify it.	The state of rejected ML-DSA iterations is never published and remains secret (#22–24).
6. Evidence	A public or declassification decision requires the binary location and trace for the exact build. Without them, the value is not accepted as public.	The SLH-DSA public-state hypothesis was withdrawn because its exact-build evidence was not preserved (#25).
7. Mixed and unknown	A candidate that receives both secret and public operands is mixed. An operand that cannot be traced to its source is unknown.	The Falcon-512 and Falcon-1024 candidates have yet to be traced and are unknown (#16, #17).
8. Decision	A reviewed secret or mixed label is risk evidence, and a reviewed public label is not. An unknown label or a pending or expired decision leaves the pair at Further Review Required.	The ML-DSA decisions have expired, so these pairs remain Further Review Required (#22–24).

References

1. National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. In *Federal Information Processing Standards Publication (FIPS) 203*; U.S. Department of Commerce: Washington, DC, USA, 2024. [CrossRef]

2. National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard. In *Federal Information Processing Standards Publication (FIPS) 204*; U.S. Department of Commerce: Washington, DC, USA, 2024. [CrossRef]
3. National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard. In *Federal Information Processing Standards Publication (FIPS) 205*; U.S. Department of Commerce: Washington, DC, USA, 2024. [CrossRef]
4. Nethercote, N.; Seward, J. Valgrind: A framework for Heavyweight Dynamic Binary Instrumentation. *ACM Sigplan Not*. **2007**, *42*, 89–100. [CrossRef]
5. Langley, A. Ctgrind: Checking That Functions Are Constant Time With Valgrind. 2010. Available online: https://github.com/agl/ctgrind (accessed on 30 September 2026).
6. Reparaz, O.; Balasch, J.; Verbauwhede, I. Dude, is my code constant time? In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 1697–1702. [CrossRef]
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 Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 23–26 May 2022; pp. 632–649. [CrossRef]
8. Bernstein, D.J.; Bhargavan, K.; Bhasin, S.; Chattopadhyay, A.; Chia, T.K.; Kannwischer, M.J.; Kiefer, F.; Paiva, T.B.; Ravi, P.; Tamvada, G. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2025**, *2025*, 209–234. [CrossRef]
9. Simon, L.; Chisnall, D.; Anderson, R. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P), London, UK, 24–26 April 2018; pp. 1–15. [CrossRef]
10. Kocher, P.C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the Advances in Cryptology—CRYPTO '96*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1109, pp. 104–113. [CrossRef] [PubMed]
11. Bernstein, D.J. Cache-Timing Attacks on AES. 2005. Available online: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (accessed on 30 September 2026).
12. Schneider, D.; Fuchs, P.; Lemke-Rust, K. A Comparative Evaluation of DATA and Microwalk for Detecting Constant-Time Violations in Cryptographic Libraries. Cryptology ePrint Archive, Paper 2026/611. 2026. Available online: https://eprint.iacr.org/2026/611 (accessed on 30 September 2026).
13. Neikes, M. TIMECOP: Automated Dynamic Analysis for Timing Side-Channels. 2019. Available online: https://post-apocalyptic-crypto.org/timecop/ (accessed on 30 September 2026).
14. 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 (ACSAC), San Juan, PR, USA, 3–7 December 2018; pp. 161–173. [CrossRef]
15. Almeida, J.B.; Barbosa, M.; Barthe, G.; Dupressoir, F.; Emmi, M. Verifying Constant-Time Implementations. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), USENIX Association, Austin, TX, USA, 10–12 August 2016; pp. 53–70.
16. Daniel, L.A.; Bardin, S.; Rezk, T. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–20 May 2020; pp. 1021–1038. [CrossRef]
17. Daniel, L.A.; Bardin, S.; Rezk, T. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Trans. Priv. Secur.* **2023**, *26*, 1–42. [CrossRef]
18. Zhou, Q.; Dang, S.; Zhang, D. CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming. In Proceedings of the 38th European Conference on Object-Oriented Programming (ECOOP 2024). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Leibniz International Proceedings in Informatics (LIPIcs), Vienna, Austria, 16–20 September 2024; Volume 313, pp. 46:1–46:26. [CrossRef]
19. 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 (CCS), Copenhagen, Denmark, 26–30 November 2023; pp. 1690–1704. [CrossRef]
20. Hanyecz, O.; Karenin, A.; Kirshanova, E.; Kutas, P.; Schaeffler, S. Constant time lattice reduction in dimension 4 with application to SQIsign. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2025**, *2025*, 511–534. [CrossRef]
21. Adjonyo, O.; Bardin, S.; Bellini, E.; Dione, G.N.; Al Ameen, M.F.; Merget, R.; Recoules, F.; Sellami, Y. Systematic Timing Leakage Analysis of NIST PQDSS Candidates: Tooling and Lessons Learned. *arXiv* **2025**, arXiv:2509.04010.
22. Zhang, Z.; Barthe, G. CT-LLVM: Automatic Large-Scale Constant-Time Analysis. Cryptology ePrint Archive, Paper 2025/338. 2025. Available online: https://eprint.iacr.org/2025/338 (accessed on 30 September 2026).
23. De Feo, L.; Kohel, D.; Leroux, A.; Petit, C.; Wesolowski, B. SQISign: Compact Post-Quantum Signatures from Quaternions and Isogenies. In *Proceedings of the Advances in Cryptology–ASIACRYPT 2020*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12491, pp. 64–93. [CrossRef]

24. Kannwischer, M.J.; Schwabe, P.; Stebila, D.; Wiggers, T. Improving Software Quality in Cryptography Standardization Projects. In Proceedings of the 2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Genoa, Italy, 6–10 June 2022; pp. 19–30. [CrossRef]
25. Dunsche, M.; Bastian, P.; Maehren, M.; Erinola, N.; Merget, R.; Bissantz, N.; Dette, H.; Schwenk, J. SILENT: A New Lens on Statistics in Software Timing Side Channels. *arXiv* **2025**, arXiv:2504.19821.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.