Verifiable Post-Quantum TLS Handshake: Recognizing Implementation Errors of Cryptographic Verification Steps

Runze Zhao[1], Siqi Lu[1], Yijian Liu[2,3], Zhaoxuan Li[2,3], Kaibo Liu[2,3], Hongbo Liu[2,3], Qingjun Yuan[1,4], Yongjuan Wang[1], Xianhui Lu[2,3]
[1]*Information Engineering University*
[2]*Institute of Information Engineering, Chinese Academy of Sciences*
[3]*School of Cyber Security, University of Chinese Academy of Sciences*
[4]*Xi'an Jiaotong University*

Abstract

The migration to post-quantum cryptography is becoming unavoidable for security protocols. Yet as the historical "goto fail" incident illustrates, deploying new primitives is not sufficient: subtle implementation bugs, especially in verification logic, can yield systems that remain fully interoperable while compromising security silently. Existing confirmation code techniques make certain errors observable, but they do not cover post-quantum signatures and provide limited protection in multi-stage handshakes.

Therefore, we present a migration-oriented solution that spans both the algorithm and protocol layers. At the algorithm layer, we develop two generic confirmation-code construction frameworks for Fiat–Shamir signatures and GPV-style Hash-and-Sign signatures, instantiated with ML-DSA and Falcon. For KEMs, we further propose a protocol-friendly, KEM-level confirmation-code method, instantiated with ML-KEM, HQC, and DAWN (NTRU-based KEM). We then prove confirmation-code unpredictability for each instantiation. At the protocol layer, we design PQCD-TLS, a post-quantum TLS handshake that binds confirmation codes into chained key derivation, ensuring that any verification fault induces key mismatch and causes handshake failure. We then extend a multi-stage authenticated key-exchange model with error-injection capabilities to formalize matching correctness with implementation errors, and prove that PQCD-TLS exposes such internal errors without compromising protocol security. Our implementation evaluates 48 configurations combining confirmation code-enhanced PQC algorithms and authentication modes. Results show that handshake latency overhead remains below 1% across all tests, while turning silent verification bugs into observable handshake failures.

1 Introduction

As post-quantum cryptography moves from standardization to deployment, a central challenge is the secure migration of real-world communication protocols such as TLS [21], SSH [38], and IPsec [36]. Translating formally verified algorithms into reliable implementations for the real world is a challenging process. In this process, ensuring system security requires not only theoretically sound post-quantum algorithms, but also correct implementations within protocol executions. In particular, implementation errors in cryptographic verification steps can silently invalidate security guarantees while leaving protocol functionality apparently intact. As a result, implementation security has become a fundamental concern in the post-quantum migration of security protocols.

This concern is not merely hypothetical. Apple's well-known goto fail bug [51] is a canonical example: a duplicated `"goto fail;"` statement caused certificate signature verification to be bypassed, allowing invalid certificates to be accepted. The problem is not limited to skipped verification branches. In `CVE-2022-21449` [1], Oracle Java SE accepted malformed ECDSA signatures because the implementation failed to validate signature components properly, enabling signature forgery. Nor is it limited to immature or lightly tested code. In `CVE-2025-69277` [47], `libsodium`, despite its 13-year zero-CVE record, missed an identity-point check in Ed25519 point validation, allowing certain invalid group elements to be accepted.

More recently, similar issues have also surfaced during the NIST's PQC standardization. Research has shown that an implementation error in the re-encryption check of the HQC reference implementation could enable key-recovery attacks, yet the flaw remained undetected for 19 months [2]. These errors are difficult to detect: verification flaws do not break basic functionality, so they often pass interoperability tests. Yet they silently compromise security. As security protocols migrate to post-quantum cryptography, systematically studying such implementation security has become both critical and practically urgent.

Detection of Implementation Errors. Heninger first articulated the idea of "tying security to basic functionality" in a talk at the Workshop on Attacks in Cryptography 2 (WAC 2), affiliated with Crypto 2019 [35]. Inspiring with this idea, Fis-

chlin and Günther [24] introduced "verifiable verification". They derive unpredictable confirmation codes from intermediate values in RSA-PSS, HMAC, and elliptic curve point validation. The verification errors thus lead to a mismatch in the confirmation code, and they propose using confirmation codes at the protocol level to expose implementation errors. Glabush *et al.* [31] extended this to KEMs based on the Fujisaki-Okamoto transform [28]. To address the risk of skipped re-encryption checks, they extract confirmation codes from the underlying encryption scheme and incorporate them into the KEM shared key derivation. They provided enhanced confirmation-code schemes for ML-KEM [8] and HQC [29]. These works lay the foundation for detecting implementation errors through confirmation codes.

Research Gap. While existing confirmation-code techniques have demonstrated how selected verification faults can be made observable, they remain insufficient for supporting the post-quantum migration of TLS. At the algorithm layer, existing constructions do not provide generic confirmation-code mechanisms for post-quantum signature schemes. Furthermore, existing designs focus on algorithm-specific implementation details, rendering them not well-suited for deployment in higher-layer cryptographic protocols. At the protocol layer, existing work does not specify a deployment strategy for confirmation codes in real-world protocols such as TLS 1.3. There is also no systematic method for tracing the propagation of implementation errors across handshake stages or for translating code mismatches into observable protocol failures.

Methodology. To address these gaps, we bridge algorithm-level verification and protocol-level functionality along three directions.

RQ1: How can confirmation code mechanisms be constructed for the post-quantum algorithms used in TLS, such that a verification fault produces confirmation code mismatch with provable unpredictability?

RQ2: How can confirmation codes be bound into the multi-stage key schedule of TLS 1.3 so that confirmation code mismatch induces stage-key mismatch and observable handshake failure, while maintaining regular security?

RQ3: How does deploying confirmation codes affect the TLS 1.3 handshake and overhead, and how can implementers use them to detect verification faults?

To answer RQ1, we construct confirmation codes for the signature and KEM algorithms used in TLS. For signatures, we establish new confirmation code unpredictability bounds under modeled faults such as skipped checks, malformed inputs, and parameter errors. We analyze both Fiat–Shamir [22] and GPV-style Hash-and-Sign [30] signatures, and provide generic constructions for the two paradigms. We instantiate it with ML-DSA [49] and Falcon [25]. For KEMs, we expose a protocol-friendly confirmation code interface while inheriting the unpredictability guarantees of existing constructions [31]. Additionally, we select the recently proposed DAWN [42](NTRU-based KEM, ASIACRYPT2025) to broaden the algorithm options available for TLS deployment. Our design turns silent verification implementation faults into observable confirmation code mismatches.

To answer RQ2, we design a binding mechanism that ties confirmation codes to the TLS 1.3 key schedule. We associate each confirmation code with the handshake stage that uses the corresponding primitive, and feed it as an extra input into that stage's key derivation. This design introduces no extra handshake messages and preserves the existing TLS 1.3 message flow. When an implementation error occurs, the resulting code mismatch compromises the affected stage key and the subsequent key chain. Consequently, the two parties derive different session keys, leading to a handshake failure. We formalize this mechanism in an enhanced multi-stage key-exchange model through **Matching Correctness with Implementation Errors**, and prove that the resulting protocol maintains **Match security** and **Multi-Stage security**.

To answer RQ3, we instantiate confirmation-code configurations from our theoretical bounds so that honest-but-faulty implementations trigger handshake failure with a noticeable probability in interoperability testing. We implement the augmented TLS 1.3 handshake and evaluate its overhead against the baseline by varying authentication modes and algorithm configurations. We measure handshake latency and communication traces, and use observable handshake failures as a practical means for implementers to detect verification faults.

Contribution. We make the following key contributions:

- We construct confirmation code mechanisms for post-quantum signatures and KEMs used in TLS, including generic constructions for Fiat–Shamir and Hash-and-Sign signatures, a protocol-friendly KEM-level interface, and concrete instantiations with ML-DSA, Falcon, ML-KEM, HQC, and DAWN. We prove confirmation-code unpredictability for each instantiation.
- We design PQCD-TLS, a post-quantum TLS 1.3 handshake that binds confirmation codes into the multi-stage key schedule. We formalize implementation-error detection through Matching Correctness with Implementation Errors and prove that PQCD-TLS maintains classic security.
- We implement PQCD-TLS and evaluate 48 configurations across different algorithm combinations, authentication modes, and confirmation-code binding states. The results show less than 1% handshake latency and no additional communication overhead, while turning silent verification bugs into observable handshake failures.

2 Preliminaries

Notation. With $\mathbb{N}$ we denote the natural numbers. For a set X, the notation $x \leftarrow\!\$\, X$ denotes sampling an element uniformly at random from X and assigning it to x. If $\mathcal{A}$ is a deterministic

algorithm, then $y \leftarrow \mathcal{A}(x)$ denotes executing $\mathcal{A}$ on input x and outputs y. If $\mathcal{A}$ is a probabilistic algorithm, then $y \leftarrow\!\$\ \mathcal{A}(x)$ denotes executing $\mathcal{A}$ on input x with uniformly random coins and storing the output in y. The notation $x = y$ evaluates to 1 if $x = y$, and 0 otherwise. The notation $a\|b$ denotes the concatenation of strings a and b, and $a \oplus b$ denotes the bitwise XOR of two strings.

Symmetric Primitives. In the Appendix A, We provide formal definitions for message authentication codes (MAC) and their extension with confirmation codes, along with the EUF-CMA security experiment. We also recall the definition of collision-resistant hash functions. In particular, for key confirmation we employ HMAC_C [24]. We employ the HKDF key derivation function [39], which consists of two stages: HKDF.Extract takes a salt and keying material (in our construction, we also incorporate confirmation codes), while HKDF.Expand expands a secret key using a label, a context string, and a hash value.

Confirmation-code-enhanced Schemes. We adopt the concept of confirmation codes from prior work [24, 31]. A confirmation code is derived from intermediate values generated during the execution of a cryptographic primitive. When an algorithm is correctly implemented and executed, it produces a consistent confirmation code; any deviation in the implementation typically results in a different code.

Definition 2.1 (Signature Scheme with Confirmation Codes, [24])**.** *A signature scheme with confirmation codes* $\mathsf{Sig}_\mathsf{C} = (\mathsf{KeyGen}, \mathsf{Sign}_\mathsf{C}, \mathsf{Vrfy}_\mathsf{C})$ *consists of three algorithms with message space* $\mathcal{M}$*:*

- $(pk, sk) \leftarrow\!\$\ \mathsf{KeyGen}()$ *outputs a public key* pk *and a secret key* sk*.*
- $(\sigma,\ cd\ ) \leftarrow\!\$\ \mathsf{Sign}_\mathsf{C}(sk, m)$ *outputs a signature* σ *and a confirmation code* cd*.*
- $(d,\ cd'\ ) \leftarrow \mathsf{Vrfy}_\mathsf{C}(pk, m, \sigma)$ *outputs a decision bit* $d \in \{0,1\}$ *and a recomputed confirmation code* cd'*.*

In a classic signature scheme $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$, no confirmation codes are derived or output. The security notion for signatures is *existential unforgeability under chosen-message attacks* (EUF-CMA); the formal experiment is provided in Appendix A.

Following [31], we address the re-encryption check issue in KEMs by extracting confirmation codes from the encryption algorithm of the underlying PKE scheme.

Definition 2.2 (Public-Key Encryption with Confirmation Codes, [31])**.** *A public-key encryption scheme with confirmation codes* $\mathsf{PKE}_\mathsf{C} = (\mathsf{KeyGen}, \mathsf{Enc}_\mathsf{C}, \mathsf{Dec}_\mathsf{C})$ *consists of three algorithms with message space* $\mathcal{M}$*:*

- $(pk, sk) \leftarrow\!\$\ \mathsf{KeyGen}()$ *outputs a public key* pk *and a secret key* sk*.*
- $(c,\ cd\ ) \leftarrow\!\$\ \mathsf{Enc}_\mathsf{C}(pk, m)$ *outputs a ciphertext* c *and a confirmation code* cd *for a message* $m \in \mathcal{M}$*.*
- $m' \leftarrow \mathsf{Dec}(sk, c)$ *outputs a message* $m' \in \mathcal{M} \cup \{\perp\}$*.*

We say PKE_C *is* δ*-correct if* $m' = m$ *with probability at least* $1 - \delta$*.*

Definition 2.3 (Key Encapsulation Mechanism with Confirmation Code)**.** *A key encapsulation mechanism with confirmation code* $\mathsf{KEM}_\mathsf{C} = (\mathsf{KeyGen}, \mathsf{Encap}_\mathsf{C}, \mathsf{Decap}_\mathsf{C})$ *consists of three algorithms with a key space* $\mathcal{K}$*:*

- $(pk, sk) \leftarrow\!\$\ \mathsf{KeyGen}()$ *outputs a public key* pk *and a secret key* sk*;*
- $(ct, ss,\ cd\ ) \leftarrow\!\$\ \mathsf{Encap}_\mathsf{C}(pk)$ *outputs a ciphertext* ct*, a shared secret* $ss \in \mathcal{K}$ *and a confirmation code* cd*;*
- $(ss',\ cd'\ ) \leftarrow \mathsf{Decap}_\mathsf{C}(sk, ct)$ *outputs* $ss' \in \mathcal{K} \cup \{\perp\}$ *and a recomputed confirmation code* cd'*.*

We say KEM_C *is* δ*-correct if* $ss' = ss$ *with probability at least* $1 - \delta$*.*

In a classic KEM scheme $\mathsf{KEM} = (\mathsf{KeyGen}, \mathsf{Encap}, \mathsf{Decap})$, no confirmation codes are derived or output. Security of a KEM is defined via indistinguishability under chosen-plaintext or chosen-ciphertext attacks (IND-CPA, IND-1CCA, IND-CCA); formal experiments are detailed in Appendix A.

3 Confirmation-code-enhanced Schemes

Lattice-based post-quantum signatures follow two main paradigms: Fiat-Shamir [3, 18, 19, 45, 46] and GPV-style Hash-and-Sign [20, 25, 30, 55]. Fiat-Shamir schemes build on an identification protocol and apply the Fiat–Shamir transform [22]. GPV-style Hash-and-Sign schemes generate signatures by sampling a lattice point close to the hash of the message in a randomized manner. Both paradigms involve verification logic—bound checks, norm checks, and parameter consistency tests—whose silent omission can break security while preserving apparent correctness. We therefore develop confirmation-code constructions for both paradigms, instantiate ML-DSA [49] and Falcon [25]. An enhanced unpredictability model is introduced to capture a wider range of implementation errors, and a security analysis is provided for each instantiation.

Lattice-based KEMs rely on two families of hardness assumptions: Learning With Errors (LWE) and its variants [8, 13, 32, 44], and NTRU lattices [7, 26, 42, 56]. To expose implementation errors to higher-layer protocols, we instead derive the confirmation code as an additional output at the KEM level. We extend this approach to NTRU-based KEMs, selecting the recently proposed DAWN algorithm [42] (ASIACRYPT 2025) as an instantiation target. NTRU-based KEMs typically offer smaller sizes and faster computation. Hence, we choose DAWN to provide more algorithm options for the TLS ecosystem.

3.1 Confirmation Code Unpredictability

We define *confirmation-code unpredictability* to formalize the following requirement: if an implementation deviates from the intended computation of a security-critical verification

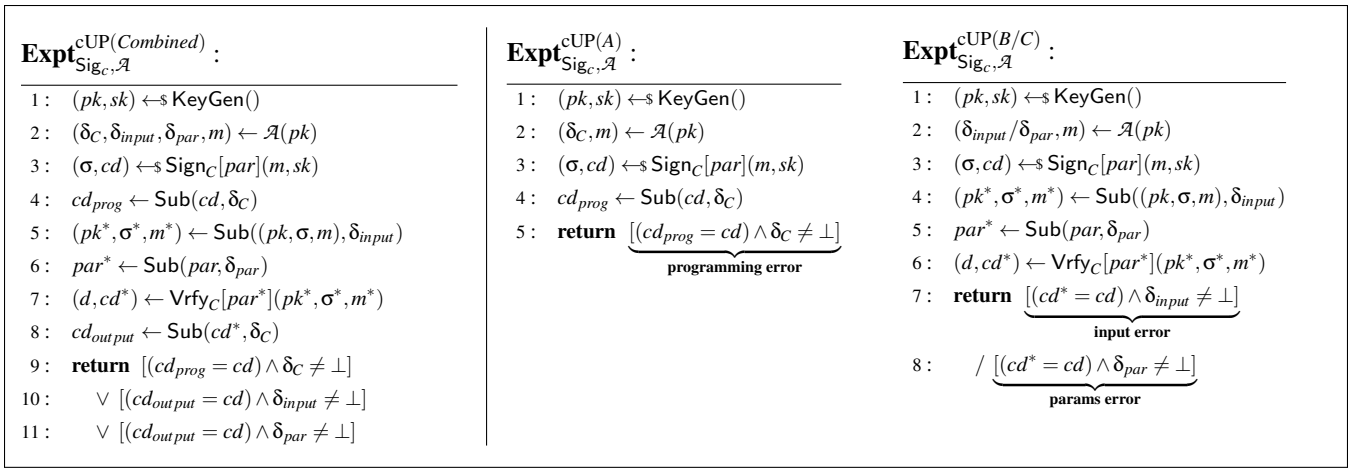

Figure 1: The confirmation code unpredictability (cUP) security experiment for signature schemes. The left side depicts the combined experiment encompassing three error types, while the right side presents three separate experiments: (A) Programming errors, (B) Input errors, (C) Parameter errors.

step, then the confirmation code produced by this faulty execution should, except with small probability, differ from the confirmation code produced by the correct execution. Thus, when the confirmation code is later bound to the surrounding protocol, such as through key derivation or an explicit consistency check, the implementation fault becomes visible as a functionality error rather than remaining silently exploitable.

Figure 1 gives the corresponding cUP experiment for signature schemes. Let $\mathsf{Sig}_\mathsf{C} = (\mathsf{KeyGen}, \mathsf{Sign}_\mathsf{C}, \mathsf{Vrfy}_\mathsf{C})$ be a signature scheme with confirmation codes. In an honest execution, Sign_C outputs a pair (σ, cd) on a message m, and Vrfy_C should reproduce the same confirmation code when run on the intended tuple (pk, σ, m) under the intended parameters par. The role of the experiment is to ask whether a faulty execution can nevertheless reproduce the same code cd. This is precisely the dangerous case for confirmation codes: if the faulty execution outputs a different code, the fault can be detected by the higher-level mechanism using the code; if it outputs the same code, the fault remains hidden.

We model implementation faults via a *non-adaptive* adversary, which fixes in advance both the locations and values of the modifications. Given the public key pk, and before the challenge message, signature, or honest confirmation code is generated, the adversary outputs $\delta = (\delta_C, \delta_{input}, \delta_{par})$.

The non-adaptiveness captures our honest-but-faulty implementation model: the implementation error is fixed in the code or configuration in advance, rather than chosen maliciously after observing the runtime values. The effect of a modification is expressed by the substitution function

$$\mathsf{Sub}(x, \delta_x) = \begin{cases} x & \text{if } \delta_x = \perp, \\ \delta_x & \text{otherwise,} \end{cases}$$

where $\perp$ denotes that no modification is made.

The combined experiment in Figure 1 covers the following three fault classes.

(A) **Programmig Errors.** These errors model faults in the implementation logic itself, such as skipped instructions [51], hard-coded values, or a bypassed computation of the confirmation code. In the experiment, this is captured by applying δ_C directly to the honest confirmation code: $cd_{prog} \leftarrow \mathsf{Sub}(cd, \delta_C)$. The adversary succeeds if $cd_{prog} = cd$ and $\delta_C \neq \perp$. This condition indicates whether programming errors will accidentally generate the correct confirmation code.

(B) **Input Errors.** These errors model faults in parsing, decoding, or reading the verification input. Instead of verifying the intended tuple (pk, σ, m), the implementation verifies a modified tuple$(pk^*, \sigma^*, m^*) \leftarrow \mathsf{Sub}((pk, \sigma, m), \delta_{input})$. The verifier then outputs (d, cd^*) on this modified input. The adversary succeeds if $cd^* = cd$ and $\delta_{input} \neq \perp$. Hence, the experiment checks whether verification on a wrong input can nevertheless yield the same confirmation code as verification on the intended input.

(C) **Parameter Errors.** These errors model missing or incorrect checks of algorithmic parameters, such as norm bounds, moduli, domains, or other configuration values. This case is especially relevant for cryptographic schemes whose security reductions typically rely on strict parameter constraints, for example bounds appearing in various security arguments. In the experiment, parameter faults are represented by $par^* \leftarrow \mathsf{Sub}(par, \delta_{par})$, and verification is run under par^*. The adversary succeeds if $cd^* = cd$ and $\delta_{par} \neq \perp$. This captures the case where an implementation uses illegal or unintended parameters but still outputs the original confirmation code. Such a fault would remain hidden unless the confir-

mation code changes. Historical vulnerabilities such as CVE-2022-21449 illustrate how missing parameter-range checks in signature verification can invalidate the intended security guarantees [1].

We emphasize that the decision bit d is not part of the success condition. Confirmation code unpredicatability measures whether the confirmation code can serve as evidence that the intended verification computation was actually carried out on the intended data and parameters. If a faulty execution changes the verification path, input, or parameters, then the confirmation code should change as well. Only when the faulty execution still produces the same confirmation code does the fault evade detection.

Definition 3.1 (Confirmation Code Unpredictability for Signature Schemes)**.** *Let* $\mathsf{Sig_C} = (\mathsf{KeyGen}, \mathsf{Sign_C}, \mathsf{Vrfy_C})$ *be a signature scheme with confirmation codes, and let* $\mathrm{Expt}^{\text{cUP-Sig}}_{\mathsf{Sig_C},\mathcal{A}}$ *be the combined cUP experiment defined in Figure 1. We define the advantage of an adversary* $\mathcal{A}$ *against the confirmation-code unpredictability of* $\mathsf{Sig_C}$ *as* $\mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{Sig_C},\mathcal{A}} := \Pr\left[\mathrm{Expt}^{\text{cUP-Sig}}_{\mathsf{Sig_C},\mathcal{A}} \Rightarrow 1\right]$.

3.2 Fiat-Shamir Signature Scheme

The signature scheme is constructed via the Fiat-Shamir transform [22] from a canonical identification scheme or a commitment-recoverable identification Scheme, augmented with a confirmation code to assist verification. During the signing phase, the signer first uses P_1 to generate a commitment w, derives the challenge $c \leftarrow \mathsf{H}(w \parallel m)$, and obtains the response z via P_2. Subsequently, the confirmation code cd is computed as $\mathsf{Code}(c)$ (or as $\mathsf{Code}(w)$ in the recoverable commitment variant). The signature output is $\sigma \leftarrow \big((w,z) \text{ or } (c,z),\ cd\big)$. Here, $\mathsf{Code}(\cdot)$ denotes the confirmation code derivation function, which, in principle, does not introduce significant additional computation. The verifier either directly recomputes $c \leftarrow \mathsf{H}(w \parallel m)$, or, in the recoverable commitment variant, recovers $w' \leftarrow \mathsf{Rec}(pk, c, z)$. The same derivation function is then applied to obtain $cd' \leftarrow \mathsf{Code}(c)$ (or $\mathsf{Code}(w')$). The verification predicate d is determined by executing $\mathsf{Ver}(pk, w, c, z)$ or by checking $\mathsf{H}(w' \parallel m) = c$. The algorithm ultimately returns the pair (d, cd').

FS-Sign$_c(sk, m)$:

1: $w \leftarrow \mathsf{P}_1(sk)$
2: $c \leftarrow \mathsf{H}(w \| m)$
3: $z \leftarrow \mathsf{P}_2(sk, W, c)$
4: $cd \leftarrow \mathsf{Code}(c)/\mathsf{Code}(w)$
5: **return** $\sigma \leftarrow (w,z)/(c,z),\ cd$

FS-Vrfy$_c(pk, m, \sigma)$:

1: $c \leftarrow \mathsf{H}(w \| m)/w' \leftarrow \mathsf{Rec}(pk, c, z)$
2: $cd' \leftarrow \mathsf{Code}(c)/\mathsf{Code}(w')$
3: $d \leftarrow \mathsf{Ver}(pk, w, z, c)/\mathsf{H}(w' \| m) = c$
4: **return** $d,\ cd'$

Figure 2: The framework for constructing confirmation codes from Fiat-Shamir signatures.

We derive confirmation codes from c or w because c captures the challenge-binding step in the standard form, while w captures the reconstructed commitment state in the recoverable form, and deviations in the corresponding verification steps are therefore reflected in the derived code. In the following, we select ML-DSA [49] as our target for instantiating this confirmation code construction for Fiat-Shamir signature algorithms.

Algorithm 1 ML-DSA$_C$.Sign$_C(sk, M)$

$\hat{\mathbf{s}}_1 \leftarrow \mathrm{NTT}(\mathbf{s}_1), \hat{\mathbf{s}}_2 \leftarrow \mathrm{NTT}(\mathbf{s}_2), \hat{\mathbf{t}}_0 \leftarrow \mathrm{NTT}(\mathbf{t}_0)$
$\hat{\mathbf{A}} \leftarrow \mathrm{ExpandA}(\rho)$
$\mu \leftarrow \mathrm{H}(tr \| M, 512)$
$rnd \leftarrow \{0,1\}^{256}, \kappa \leftarrow 0, (\mathbf{z}, \mathbf{h}) \leftarrow \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} \leftarrow \mathrm{ExpandMask}(\rho', \kappa)$
  $\mathbf{w} \leftarrow \mathrm{NTT}^{-1}\big(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{y})\big)$
  $\mathbf{w}_1 \leftarrow \mathrm{HighBits}(\mathbf{w})$
  $\tilde{c} \leftarrow \mathrm{H}\big(\mu \| \mathrm{w1Encode}(\mathbf{w}_1), 2\lambda\big)$ ▷Commitment
  $(\tilde{c}_1, \tilde{c}_2) \in \{0,1\}^{256} \times \{0,1\}^{2\lambda-256} \leftarrow \tilde{c}$
  $\hat{c} \leftarrow \mathrm{SampleInBall}(\tilde{c}_1)$
  $\langle\langle c\mathbf{s}_1 \rangle\rangle \leftarrow \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_1), \langle\langle c\mathbf{s}_2 \rangle\rangle \leftarrow \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2)$
  $\mathbf{z} \leftarrow \mathbf{y} + \langle\langle c\mathbf{s}_1 \rangle\rangle$ ▷Response
  $\mathbf{r}_0 \leftarrow \mathrm{LowBits}\big(\mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle\big)$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
  else
    $\langle\langle c\mathbf{t}_0 \rangle\rangle \leftarrow \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)$
    $\mathbf{u} \leftarrow \mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle + \langle\langle c\mathbf{t}_0 \rangle\rangle$
    $\mathbf{h} \leftarrow \mathrm{MakeHint}\big(-\langle\langle c\mathbf{t}_0 \rangle\rangle, \mathbf{u}\big)$
    if $\|\langle\langle c\mathbf{t}_0 \rangle\rangle\|_\infty \geq \gamma_2$ **or** $\mathrm{weight}(\mathbf{h}) > \omega$ **then**
      $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
    end if
  end if
  $\kappa \leftarrow \kappa + \ell$
end while
$\sigma \leftarrow \mathrm{sigEncode}\big(\tilde{c}, \mathbf{z} \bmod^{\pm} q, \mathbf{h}\big)$
$cd_0 \leftarrow \big(\mathbf{w}_1[i_1][j_1] \bmod 2^4\big) \| \cdots \| \big(\mathbf{w}_1[i_\ell][j_\ell] \bmod 2^4\big)$
$cd \leftarrow cd_0 \oplus \big((\gamma_1 - \beta) \bmod 2^{4\ell}\big)$ ▷Generate confirmation code
return (σ, cd)

ML-DSA: A commitment-recoverable signature scheme based on the Fiat-Shamir with Aborts paradigm. The signature consists of $(\tilde{c}, \mathbf{z}, \mathbf{h})$, where $\tilde{c}$ encodes the challenge, $\mathbf{z}$ is the response, and $\mathbf{h}$ is a hint vector that enables the verifier to reconstruct $\mathbf{w}'_1$, the rounded high-order component of the commitment $\mathbf{w} = \mathbf{Ay}$. We construct the confirmation code for ML-DSA based on this reconstructed commitment component. Specifically, after recovering $\mathbf{w}'_1$ during verification, we compute $cd'_0 \leftarrow \big(\mathbf{w}'_1[i_1][j_1] \bmod 2^4\big) \| \ldots \| \big(\mathbf{w}'_1[i_\ell][j_\ell] \bmod 2^4\big), cd' \leftarrow cd'_0 \oplus \big((\gamma_1 - \beta) \bmod 2^{4\ell}\big)$, where $\mathbf{w}'_1[i][j]$ denotes a coefficient of the reconstructed commitment component. The XOR with the low-order bits of the norm bound $\gamma_1 - \beta$ makes misconfig-

Algorithm 2 ML-DSA$_{\mathrm{C}}$.Vrfy$_{\mathrm{C}}(pk, M, \sigma)$

$(\rho, \mathbf{t}_1) \leftarrow \mathrm{pkDecode}(pk)$
$(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow \mathrm{sigDecode}(\sigma)$
$\hat{\mathbf{A}} \leftarrow \mathrm{ExpandA}(\rho)$
$tr \leftarrow \mathrm{H}(\mathrm{BytesToBits}(pk), 512)$
$\mu \leftarrow \mathrm{H}(tr \| M, 512)$ ▷Compute message representative μ
$(\tilde{c}_1, \tilde{c}_2) \in \{0,1\}^{256} \times \{0,1\}^{2\lambda-256} \leftarrow \tilde{c}$
$c \leftarrow \mathrm{SampleInBall}(\tilde{c}_1)$
$\mathbf{w}'_{\mathrm{Approx}} \leftarrow \mathrm{NTT}^{-1}\big(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{z}) - \mathrm{NTT}(c) \circ \mathrm{NTT}(\mathbf{t}_1 \cdot 2^d)\big)$
$\mathbf{w}'_1 \leftarrow \mathrm{UseHint}(\mathbf{h}, \mathbf{w}'_{\mathrm{Approx}})$ ▷Reconstruction commitment
$\tilde{c}' \leftarrow \mathrm{H}\big(\mu \| \mathrm{w1Encode}(\mathbf{w}'_1), 2\lambda\big)$
if $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ **and** $\tilde{c} = \tilde{c}'$ **and** $\mathrm{weight}(\mathbf{h}) \le \omega$ **then**
  $cd'_0 \leftarrow \big(\mathbf{w}'_1[i_1][j_1] \bmod 2^4\big) \| \cdots \| \big(\mathbf{w}'_1[i_\ell][j_\ell] \bmod 2^4\big)$
  $cd' \leftarrow cd'_0 \oplus \big((\gamma_1 - \beta) \bmod 2^{4\ell}\big)$
  return $(d, cd') \leftarrow (1, cd')$
else
  return $(d, cd') \leftarrow (0, \perp)$
end if

uration of this critical parameter observable with noticeable probability.

The parameter ℓ determines the confirmation-code length and the resulting collision bound. Specifically, the construction extracts four low-order bits from each of ℓ fixed public coefficient positions of $\mathbf{w}'_1$, yielding a 4ℓ-bit value before masking with $(\gamma_1 - \beta) \bmod 2^{4\ell}$. Increasing ℓ therefore increases the code length linearly and decreases the probability that a faulty implementation accidentally matches the honest confirmation code. The practical role of the confirmation code is to make otherwise silent verification errors observable. In our ML-DSA instantiation, faults such as skipped or incorrect commitment reconstruction, input-handling errors, or misconfiguration of $\gamma_1 - \beta$ change the reconstructed verification state, and therefore lead to a confirmation-code mismatch. The following theorem formalizes this unpredictability guarantee for ML-DSA$_{\mathrm{C}}$. The proof is given in Appendix C.2.

Theorem 3.1. *Let ML-DSA$_C$ denote the ML-DSA signature scheme equipped with a confirmation code, as defined in Algorithms 1 and 2, where $i \in \{0, \ldots, \ell\}$, $j \in \{0, \ldots, 255\}$. Let $\mathcal{A}$ be a non-adaptive adversary in the* cUP-Sig *experiment for the confirmation code. Then we have:* $\mathsf{Adv}^{\text{cUP-Sig}}_{\text{ML-DSA}_{\mathrm{C}},\mathcal{A}} \le 2 \cdot (P_{\max})^{\ell} + 1/2^{4\ell}$*, where $P_{\max} = 3/44$ for ML-DSA$_C$-44 and $P_{\max} = 1/16$ for ML-DSA$_C$-65/87.*

3.3 GPV-Style Hash-and-Sign Signature Scheme

Hash-and-sign signatures are typically based on the GPV framework [30], which utilizes the preimage sampling algorithm to construct trapdoor functions based on lattice hard problems. It has been proven that the security of these trapdoor functions is equivalent to that of trapdoor permutations. The construction of key generation, signing, and verification is as follows:

- **Key generation:** Given initial parameters m, n, q, the public key is a full-rank matrix $A \in \mathbb{Z}_q^{m \times n}$ (with $m > n$) modulo q, and the private key is a short basis $B \in \mathbb{Z}_q^{n \times n}$ of the lattice satisfying $BA^\top = 0$.
- **Sign:** For a message $M \in \mathcal{M}$ and a hash function H, first solve the linear system $c_0 A^\top = H(M) \pmod q$ to obtain a vector $c_0 \in \mathbb{Z}_q^n$ (which need not be short). Then, using the private key B, find a lattice vector $c \in \mathcal{L}(B)$ close to c_0. The short vector $s = c_0 - c$ serves as the signature for M.
- **Vrfy:** To verify the signature, it check that s is short and the equality $sA^\top = H(M) \pmod q$ holds.

In GPV-style hash-and-sign signatures, verification consists of two essential checks: the linear relation between the signature and the hash target, and the shortness of the signature vector. Following the confirmation-code principle, the code should be derived from an intermediate value that is actually computed along the verification path and reflects the critical verification condition. Directly using the public hash target $H(M)$ is therefore unsuitable: it can be recomputed from the message alone and does not indicate whether the verifier has performed the norm check. We instead derive the confirmation code from the norm-check slack, which ties the code to the acceptance condition $\|\mathbf{s}\|^2 \le \beta^2$.

Falcon: A concrete instantiation of the GPV framework, achieving compact signatures and high operational efficiency through the use of NTRU lattices and the Fast-Fourier sampling. The scheme operates over the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. The private key comprises a tuple of short polynomials $(f, g, F, G) \in \mathcal{R}^4$ satisfying the NTRU equation $fG - gF = q \bmod (x^n + 1)$, while the public key is $h = gf^{-1} \in \mathcal{R}_q$. To sign a message m, the signer first hashes a random salt r together with m to generate a target polynomial $c \in \mathcal{R}_q$. Subsequently, employing a precomputed Falcon tree, a fast Fourier sampling algorithm produces a short signature vector $(s_1, s_2) \in \mathcal{R}^2$ such that $s_1 + s_2 h \equiv c \pmod q$. The final signature is the pair (r, s), where s is a compressed encoding of s_2. Verification entails recomputing c from (r, m), decompressing s to recover s_2, computing $s_1 = c - s_2 h \bmod q$, and accepting the signature if and only if $\|(s_1, s_2)\|^2 \le \beta^2$ for the scheme's predefined norm bound β^2.

We construct the confirmation code for Falcon based on the norm difference computed during verification. Specifically, after computing the signature vector $\mathbf{s} = (s_1, s_2)$, we compute the confirmation code as $cd \leftarrow \big(\lfloor \beta^2 \rfloor - \|\mathbf{s}\|^2\big) \bmod 2^\ell$, where $\mathbf{s} = (s_1, s_2)$ and $l \le \lfloor \log_2 \beta^2 \rfloor$. This construction directly encodes the verification condition $\|\mathbf{s}\|^2 \le \beta^2$.

Our design is motivated by the need to detect misconfigurations of critical verification parameters. Since the norm $\|\mathbf{s}\|^2$ of a valid signature has sufficient entropy within the range bounded by β^2, a non-adaptive adversary cannot predict the value of the confirmation code. Moreover, by embedding the

Algorithm 3 $\text{Falcon}_C.\text{Sign}_C(sk, m)$

$r \leftarrow \{0,1\}^{256}$
$c \leftarrow \text{HashToPoint}(r, m, q, n)$
$\mathbf{t} \leftarrow \left(-\frac{1}{q}\text{FFT}(c) \odot \text{FFT}(F), \frac{1}{q}\text{FFT}(c) \odot \text{FFT}(f)\right)$
repeat
  repeat
    $\mathbf{z} \leftarrow \text{ffSampling}_n(\mathbf{t}, \mathbf{T})$ ▷ Fast Fourier Sampling
    $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
  until $\|\mathbf{s}\|^2 \leq \lfloor \beta^2 \rfloor$
  $cd \leftarrow \left(\lfloor \beta^2 \rfloor - \|\mathbf{s}\|^2\right) \bmod 2^l$ ▷ Generate cd
  $(s_1, s_2) \leftarrow \text{invFFT}(\mathbf{s})$
  $s \leftarrow \text{Compress}(s_2)$
until $s \neq \perp$
$\sigma \leftarrow (r, s)$
return (σ, cd)

Algorithm 4 $\text{Falcon}_C.\text{Vrfy}_C(\sigma, m, pk)$

$(r, s) \leftarrow \sigma$
$c \leftarrow \text{HashToPoint}(r, m, q, n)$
$s_2 \leftarrow \text{Decompress}(s)$
if $s_2 = \perp$ **then**
  return $(0, \perp)$ ▷ Reject invalid encodings
end if
$s_1 \leftarrow c - s_2 h \bmod q$
if $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ **then**
  $d \leftarrow 1$, $cd' \leftarrow \left(\lfloor \beta^2 \rfloor - \|(s_1, s_2)\|^2\right) \bmod 2^\ell$
else
  $d \leftarrow 0$, $cd' \leftarrow \perp$ ▷ Verification failed
end if
return (d, cd')

low-order bits of the norm bound β^2 directly into the confirmation code, any modification to β^2 will cause a deviation in the confirmation code, which can be detected with high probability. Confirmation code unpredictability follows from Theorem 3.2 and the proof is given in Appendix C.3.

Theorem 3.2. *Let Falcon_C denote the Falcon signature scheme augmented with a confirmation code, as defined in Algorithms 3 and 4, Let $\mathcal{A}$ be a non-adaptive adversary in the* cUP-Sig *experiment for the confirmation code. Then we have:* $\text{Adv}^{\text{cUP-Sig}}_{\text{Falcon}_C,\mathcal{A}} \leq 3 \cdot 2^{-\ell}$.

3.4 KEM Schemes in Upper-Layer Protocols

To detect potential implementation errors in the underlying Key Encapsulation Mechanism (KEM) at the upper-layer protocol level(e.g. TLS 1.3). We directly output the confirmation code as an additional KEM-level output as Figure 3. This ensures that even when the shared secret is identical, a mismatch in the confirmation code can assist implementers in locating the error, achieving the effect of "same secret, different confirmation code".

Encap(pk):	**Decap**(sk, c):
$m \leftarrow\!\!\$\ \mathcal{M}$	$m' \leftarrow \text{Dec}(sk, c)$
$(c, \boxed{cd}) \leftarrow \text{Enc}_C(pk, m)$	$(c', \boxed{cd'}) \leftarrow \text{Enc}(pk, m')$
$K \leftarrow \text{KDF}(m, pk)$	**if** $c' \neq c$ **then return** $(\perp, \perp)$
return $(K, c, \boxed{cd})$	$K' \leftarrow \text{KDF}(m', pk)$
	return $(K', \boxed{cd'})$

Figure 3: KEM encapsulation and decapsulation with confirmation codes.

We need to analyze whether the confirmation-code unpredictability definition for the underlying PKE scheme can be extended to the KEM layer that incorporates re-encryption checks. We prove that confirmation-code unpredictability of the underlying PKE lifts to the KEM layer even with re-encryption checks, and the KEM preserves its original security. The proof is detailed in Appendix C.4.

Corollary 3.1 (Confirmation Code Unpredictability for KEM with Re-encryption Checks)**.** *Let $\text{KEM}_C = (\text{KeyGen}, \text{Encap}_C, \text{Decap}_C)$ be a confirmation code augmented key encapsulation scheme that internally uses a PKE scheme PKE_C and performs a re-encryption check during decapsulation. The confirmation code cd output by Encap_C and Decap_C is exactly the code produced by the underlying $\text{PKE}_C.\text{Enc}_C$. Let $\mathcal{B}$ be an adversary against the confirmation code unpredictability of KEM_C in the experiment adapted for KEMs. Then there exists an adversary $\mathcal{A}$ against the confirmation-code unpredictability of the underlying PKE_C scheme such that:*

$$\text{Adv}^{\text{cUP-KEM}}_{\text{KEM}_C,\text{SampleE},max}(\mathcal{B}) \leq \text{Adv}^{\text{cUP-PKE}}_{\text{PKE}_C,\text{SampleE},max}(\mathcal{A}).$$

Based on the KEM-level interface above, we instantiate confirmation-code-augmented KEMs for the post-quantum algorithms used in our protocol. For ML-KEM and HQC, we adopt the confirmation-code-enhanced encryption algorithms Enc_C proposed in [31], and lift them to the KEM level using the encapsulation and decapsulation interface in Figure 3. This yields protocol-friendly KEM variants whose confirmation codes are local auxiliary outputs and can be directly bound into the TLS key schedule without changing the handshake message flow. To provide more deployment choices for TLS, we additionally instantiate the interface with the NTRU-based DAWN KEM (ASIACRYPT2025, [43]). The DAWN-specific construction, confirmation-code extraction rule, and cUP bound are given in Appendix B and C.5.

4 The PQCD-TLS Protocol

The PQCD-TLS protocol enhances the TLS 1.3 framework [52] by introducing confirmation-code-augmented cryptographic schemes in Figure 4. These modifications establish

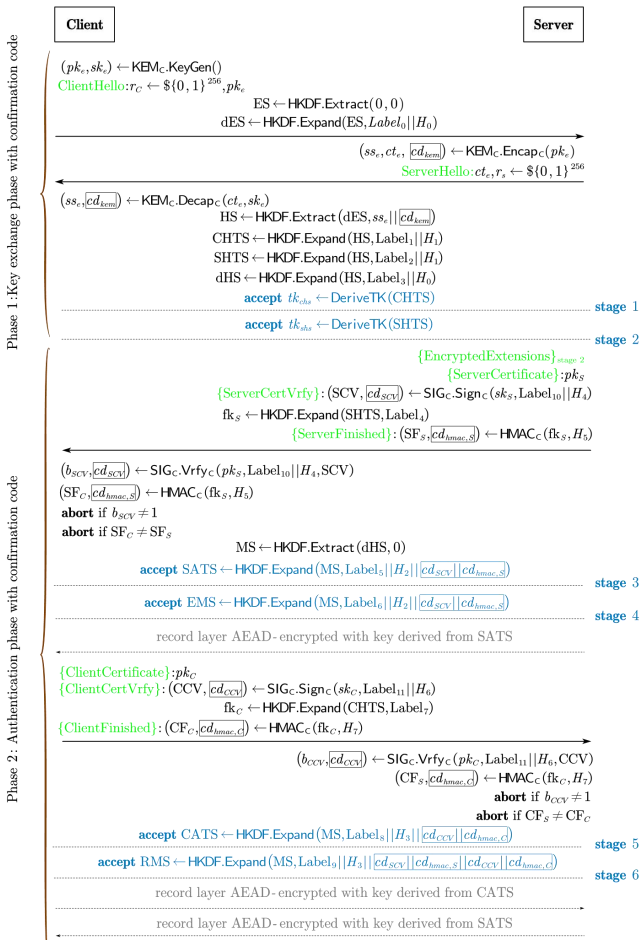

Figure 4: The PQCD-TLS handshake protocol

stronger key integrity guarantees and finer-grained key separation, while also enabling effective exposure of implementation errors. The protocol execution consists of two core phases: the ephemeral key exchange phase and the authentication phase [6, 9, 15, 17, 40, 41, 50].[1]

4.1 Ephemeral Key Exchange Phase

The client first generates a KEM key pair (pk_e, sk_e) and samples a random nonce r_C. It then sends pk_e and r_C to the server in a `ClientHello` message. Upon receipt, the server executes the enhanced key encapsulation algorithm $\mathsf{KEM_C.Encap_C}(pk_e)$, which produces a shared secret ss_e, a ciphertext ct_e, and a local confirmation code cd_{kem}. The server responds with a `ServerHello` message containing ct_e and its own random nonce r_S. The confirmation code cd_{kem} is not included in the `ServerHello`. The client then executes $\mathsf{KEM_C.Decap_C}(ct_e, sk_e)$ to recover ss_e and its local confirmation code cd_{kem}. Both parties proceed to key derivation using the shared secret ss_e and their locally computed KEM confirmation codes.

4.2 Authentication Phase

All messages in the authentication phase are encrypted using the handshake traffic keys and consist of server authentication and optional client authentication.

Server Authentication. The server first sends its certificate $\mathsf{pk_S}$, then signs the hash of all handshake messages up to this point using its long-term private key $\mathsf{sk_S}$. This produces a signature SCV and a server signature confirmation code cd_{scv}. The server then derives the finished key $\mathsf{fk_S}$ from SHTS and computes the authentication tag SF_s for the finished message, along with a confirmation code $cd_{hmac,s}$. The client verifies the signature and MAC tag, and computes its local copies of cd_{scv} and $cd_{hmac,s}$.

(Optional) Client Authentication. If the server requests client authentication, the client sends its certificate $\mathsf{pk_C}$ and generates a signature CCV along with a client signature confirmation code cd_{ccv}. The client then derives the finished key $\mathsf{fk_C}$ from CHTS and computes the authentication tag CF_c for the finished message, along with a confirmation code $cd_{hmac,c}$. The server verifies the signature and MAC tag, and computes its local copies of cd_{ccv} and $cd_{hmac,c}$.

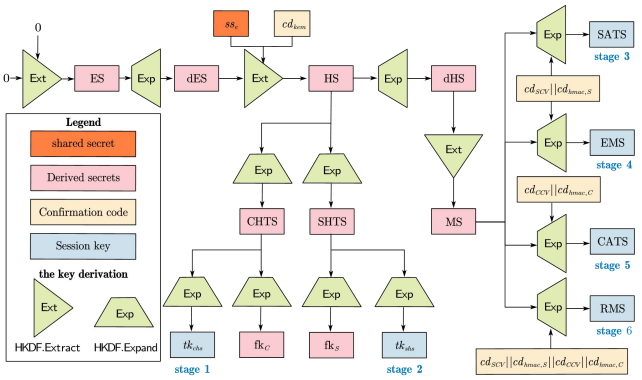

Figure 5: Key derivation schedule with confirmation code.

4.3 Key Derivation Schedule

In TLS 1.3, the handshake secret HS is derived via HKDF.Extract using the Diffie-Hellman shared secret g^{xy}. Following [54], the PQCD-TLS protocol employs a KEM exchange and incorporates the confirmation code cd_{kem} as an additional input in Figure 5. This ensures that the correct execution of the key encapsulation mechanism (implementation

[1] The confirmation-code-enhanced MAC extracts the internal hash value of HMAC as confirmation codes [24].

errors cause a confirmation code mismatch) is a prerequisite for all subsequent keying material, effectively preventing KEM implementation errors.

Following this, mutual authentication is performed. A key design choice is the **delayed generation** of CATS (Client Application Traffic Secret), as illustrated in Fig. 4. Specifically, CATS is derived only after the client has successfully authenticated the server and verified its Finished. After server authentication is completed, both parties proceed to derive the master secret MS in the standard manner. Subsequently, SATS (Server Application Traffic Secret) and EMS (Exporter Master Secret) are derived, where the generation of SATS and EMS incorporates the relevant server-side confirmation codes $(cd_{SCV}, cd_{hmac,S})$.

When mutual authentication is enabled, the protocol then continues with client authentication. Only after the server has successfully authenticated the client and verified the `ClientFinished` do both parties compute CATS and RMS (Resumption Master Secret). In this case, CATS incorporates the client-side confirmation codes $(cd_{CCV}, cd_{hmac,C})$, while RMS incorporates all authentication-stage confirmation codes $(cd_{SCV}, cd_{hmac,S}, cd_{CCV}, cd_{hmac,C})$.

In the server-authentication-only mode, the client authentication is omitted. Consequently, cd_{CCV} are not generated and are not used in the derivations of CATS or RMS; in this case, RMS incorporates only the server-side confirmation codes $(cd_{SCV}, cd_{hmac,S}, cd_{hmac,c})$.

Rationale for the Key Derivation Schedule with Confirmation Code. An intuitive alternative would be to incorporate all confirmation codes directly into the derivation of the master secret MS, causing all subsequent keys to fail upon any error. However, a significant drawback of this approach is that it *obscures error localization*. If the handshake fails, implementers would find it difficult to pinpoint the exact stage or party where the implementation flaw occurred.

Our staged incorporation of confirmation codes addresses this. For instance, if an implementation error exists on the client side, the encryption key used by the server to send messages to the client (SATS-derived) will become inconsistent, causing the client to fail in decrypting received messages. The converse holds true for server-side errors. This design enables developers to localize the source of an error through basic interoperability testing.

The inclusion of all authentication stage confirmation codes in the derivation of RMS serves a specific purpose: it safeguards against scenarios where interoperability tests are not performed immediately after the initial handshake. If an application directly proceeds to use RMS for deriving session tickets, any prior undetected authentication error will now cause a mismatch, preventing the generation of identical session tickets. This ensures that errors cannot remain latent and affect resumed sessions.

5 Security Analysis

Our approach adapts the security models proposed by DFGS model [15, 16] for the handshake of Post-Quantum TLS with Confirmation Codes. Building upon this foundation, we aim to analyze the protocol's capability to capture implementation errors.

Security goal. We demonstrate that the introduced confirmation code mechanism preserves all established security properties of the standard TLS 1.3 handshake protocol under the multi-stage key exchange model. These include: (1) **Matching security**, which ensures that both parties with the same session identifier compute consistent keys; and (2) **Multi-stage security** for each stage's session keys, meaning that in a fresh session, an adversary cannot distinguish the real session key from a random string of equal length. Furthermore, we prove that the design can detect and block security failures caused by implementation flaws at the protocol level. Hence, we introduce a new security goal in the expanded multi-stage security model: (3) **Matching Correctness with Implementation Errors(MCIE)**. This property formally captures the following guarantee: for any implementation flaw within a predefined set, if such a flaw occurs in a session, the two parties cannot generate identical stage keys. The resulting key mismatch will cause subsequent handshake verification to fail or prevent the establishment of application-layer communication, thereby transforming potential security vulnerabilities into visible connection failures.

5.1 Security Model

We build on the DFGS multi-stage AKE model [15, 16] and inherit its participants, long-term keys, PKI assumptions, session identifiers, partnering notions, adversarial queries, and freshness conditions. We only describe the extensions needed to reason about confirmation-code-based implementation-error detection; the full formal syntax is deferred to Appendix D.

Error Records. Each local session is augmented with two additional records: an error record ERROR and a confirmation-code record CD. The error record is organized according to the protocol location where a silent verification fault may occur. In PQCD-TLS, we consider three such locations: the KEM decapsulation step in the ephemeral key exchange, the signature verification step in authentication, and the HMAC verification step in Finished message. Accordingly, we write $\mathsf{ERROR} \subset \{\mathsf{KEM.error}, \mathsf{SIG.error}, \mathsf{HMAC.error}\}$

It is important to emphasize the scope of this modeling. The confirmation-code unpredictability (cUP) proofs only guarantee that, when an error defined in ERROR occurs, the two parties generate different confirmation codes except with the bounded probability given by the corresponding cUP theorem. Once these confirmation codes are bound into the key schedule, such a mismatch propagates through the key-derivation

chain and causes the affected stage keys to differ. This guarantee does not apply to implementation errors outside ERROR, although such errors may still be exposed in practice. In this work, we instantiate the possible fault classes as follows:

- $\mathsf{KEM.error} \triangleq \{\mathsf{KEM.bypass}\}$: denotes the KEM re-encryption bypass error, i.e., skipped or tampered with the KEM decapsulation process (corresponding to the cUP proof in Definition B.1)
- $\mathsf{SIG.error} \triangleq \{\mathsf{SIG.program}, \mathsf{SIG.input}, \mathsf{SIG.param}\}$: capture faults in signature verification. SIG.program models programming errors such as skipped or hard-coded computations; SIG.input models parsing, decoding, or input-handling faults; and SIG.param models incorrect parameters. These are the fault classes covered by the signature cUP experiment in Definition 3.1.
- $\mathsf{HMAC.error} \triangleq \{\mathsf{HMAC.program}, \mathsf{HMAC.input}\}$: capture faults in HMAC verification. HMAC.program models MAC checking skipped, while HMAC.input models faults in the input to the MAC verification procedure. These correspond to the HMAC fault classes considered in [24, Theorem 4.2].

Confirmation Code Records. For each local session, the model also records the confirmation codes generated during the execution of the corresponding cryptographic operations. We denote this local record by CD. In PQCD-TLS, confirmation codes are associated with the same three protocol locations as the error record: the KEM-based ephemeral key exchange, the signature-based authentication step, and the HMAC-based Finished verification step. Concretely, the confirmation code record contain $\mathsf{CD} \subset \{cd_{\mathsf{kem}}, cd_{\mathsf{SCV}}, cd_{\mathsf{hmac,S}}, cd_{\mathsf{CCV}}, cd_{\mathsf{hmac,C}}\}$.

Here, cd_{kem} is generated during KEM encapsulation and decapsulation. cd_{SCV} is generated during the `ServerCertVrfy` signing and verification procedures. $cd_{\mathsf{hmac,S}}$ is generated during the computation and verification of the `ServerFinished`. cd_{CCV} is generated during the optional `ClientCertVrfy` signing and verification procedures. and $cd_{\mathsf{hmac,C}}$ is generated during the computation and verification of the `ClientFinished`.

A confirmation code is always local to the party that generates it. It is never sent as a protocol message and is never directly compared with the peer's value. Instead, PQCD-TLS binds the locally generated confirmation code into the corresponding stage of the key schedule. Therefore, if two intended peers generate different confirmation codes for the same cryptographic operation, the mismatch is propagated into the derived stage keys.

In an honest-and-correct execution, matching sessions generate identical confirmation codes for the same operation and thus derive matching stage keys. If an injected error from ERROR affects one of the corresponding cryptographic operations, the cUP guarantee of the underlying primitive implies that the locally generated confirmation code differs from the honest one except with the bounded probability. Since this code is used as input to key derivation, the affected stage keys will mismatch, causing handshake failed. If a confirmation code is not generated in a given execution path, its entry is set to $\perp$ and is not used in the corresponding key-derivation step. For example, in the server-authentication-only mode, the client authentication flight is omitted and cd_{CCV} is not generated.

Adversary interaction. The adversary has full network control as in the DFGS model and can access the standard queries NewSession, Send, Reveal, Corrupt, and Test. To model implementation faults, we add one query:

- $\mathsf{Inject}(U, i, j, e)$: marks user U as faulty with error e during stages i through j, where $e \in$ ERROR. For any affected session of U, the corresponding primitive execution follows the fault class captured by the relevant cUP experiment, and the locally generated confirmation code is recorded in CD.

The Inject query is not a network attack and does not model a malicious party. It represents an honest-but-faulty implementation whose verification logic deviates from the intended computation. The injected error is restricted to the predefined classes in ERROR, which are exactly the classes covered by the cUP guarantees of the underlying primitives.

The standard Multi-stage security game is evaluated only on sessions without injected implementation errors, since this game captures whether PQCD-TLS preserves the usual key indistinguishability guarantees of TLS 1.3. Sessions affected by Inject are analyzed separately in the MCIE game. In MCIE, the adversary wins if, after injecting an error from ERROR, two intended peer sessions still derive matching affected stage keys. Thus, the two games separate ordinary protocol security from implementation-error exposure.

5.2 Security Properties

The AKE protocol security is typically captured by two fundamental properties: *Matching-Security* and the traditional notion of *key indistinguishability* [5, 10, 11, 23]. Matching-Security, formalized through the modeling of session identifiers, guarantees that intended communication partners compute identical session keys. To analyze whether the PQCD-TLS protocol can expose errors when they occur, we introduce a new definition of *Matching Correctness with Implementation Errors*, which extends the established concept of *Matching-Security*. This property formally characterizes whether session participants can still generate matching session keys when certain implementation errors are present.

5.2.1 Matching Security.

This is a central security goal for multi-stage key exchange protocols, ensuring session consistency between honest parties. It requires that any two sessions sharing the same session identifier (sid) in a given stage must agree on all essential

state elements, including confirmation codes, session keys, authentication types, contributive identifiers, roles, and that no third session may hold the same sid. The model captures all feasible matching failures under the assumption of a correct protocol implementation. The proof is given in D.3.

Theorem 5.1 (Matching-Security of PQCD-TLS)**.** *The PQCD-TLS protocol is* $\mathsf{Matching-secure}$*, for any efficient adversary* $\mathcal{A}$ *we have*

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{PQCD\text{-}TLS},\mathcal{A}} \leq n_s\delta_e + n_s^2/2^{|\mathsf{nonce}|}$$

where n_s *is the number of sessions,* $|nonce|$ *is the length of the nonces* r_C *and* r_S*, the ephemeral KEM is assumed to be* δ_e*-correct.*

5.2.2 Multi-Stage Security.

This is a security goal for modern key exchange protocols, aiming to provide independent security guarantees for keys derived at different stages of the handshake. This model ensures that even if some stage keys are compromised, the confidentiality of keys in other stages remains intact. More specifically, the protocol must satisfy key independence, forward secrecy, and resistance against active attacks. We prove the overall security of the protocol via a sequence of game hops, gradually reducing the advantage of an adversary $\mathcal{A}$ in the original security game to the security of underlying cryptographic primitives, such as the $\mathsf{KEM_C}$, HKDF, and signature scheme. The resulting security bound is stated in the theorem above. The reductionist proof is given in Appendix D.4.

Theorem 5.2 (Multi-Stage Security of PQCD-TLS)**.** *The PQCD-TLS is Multi-Stage secure with* $(\mathsf{M},\mathsf{AUTH},\mathsf{FS},\mathsf{USE})$*. For any PPT adversary* $\mathcal{A}$ *against the Multi-Stage security there exist efficient algorithms* $\mathcal{B}_1,\ldots,\mathcal{B}_7$ *such that*

$$\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{PQCD\text{-}TLS},\mathcal{A}} \leq \frac{n_s^2}{2^{|\mathsf{nonce}|}} + \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1} + n_s\delta_e + 6n_s \cdot \begin{pmatrix} n_s \cdot \begin{pmatrix} 2\mathsf{Adv}^{\text{IND-1CCA}}_{\mathsf{KEM_C},\mathcal{B}_2} + \frac{q_G}{|\mathcal{K}|} + 2\mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_3} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_4} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_5} \end{pmatrix} \\ +n_u \cdot \mathsf{Adv}^{\text{EUF-CMA}}_{\mathsf{SIG_C},\mathcal{B}_6} \end{pmatrix}.$$

where n_s *is the number of sessions,* n_u *is the number of users,* $\mathcal{K}$ *denotes the KEM shared secret space, the ephemeral KEM is assumed to be* δ_e*-correct,* q_G *is the maximum number of queries to the random oracle G.*

5.2.3 MCIE Security.

This is a key measure of the robustness of multi-stage key exchange protocols. It requires that when implementation errors are present, the protocol prevents the two parties from deriving the same session key, thereby avoiding hidden security vulnerabilities.

Theorem 5.3 shows how the confirmation codes function in the PQCD-TLS protocol through the key derivation chain. An error at any stage results in a mismatch of the corresponding confirmation code, which then propagates through all subsequent key derivations, ultimately preventing communication from proceeding. Specifically, an error in the early handshake (Stages 1/2) prevents the derivation of a consistent HS, halting the transmission of all subsequent authentication messages. An error during the mid-authentication phase (Stages 3/4) causes MS to diverge, thereby preventing the alignment of all application traffic keys. Finally, an error in the final handshake stages (Stages 5/6) obstructs the generation of a consistent CATS. This not only renders the server incapable of decrypting client messages but also ensures that the RMS differs between parties, consequently invalidating the establishment of any session ticket for connection resumption. The proof is given in Appendix D.5, together with concrete deployment recommendations for implementers that respect the theoretical bounds.

Theorem 5.3 (MCIE Security of PQCD-TLS)**.** *The PQCD-TLS protocol is MCIE-secure with properties* $(\mathsf{M},\mathsf{AUTH},\mathsf{FS},\mathsf{USE})$*. For any PPT adversary* $\mathcal{A}$ *against the MCIE security of PQCD-TLS, there exist efficient algorithms* $\mathcal{B}_1,\ldots,\mathcal{B}_6$ *such that*

$$\mathsf{Adv}^{\mathsf{MCIE}}_{\mathsf{PQCD-TLS},\mathcal{A}} \leq \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1} + n_s \cdot \delta_e + n_e \cdot \begin{pmatrix} \mathsf{Adv}^{\text{cUP-KEM}}_{\mathsf{KEM_C},\mathcal{B}_2} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_3} \\ +\mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_4} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_5} \\ +\mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_6} \end{pmatrix}.$$

where n_s *is the number of sessions,* n_e *is the number of users whose implementations contain errors, and* δ_e *is the correctness error probability of the ephemeral KEM.*

6 Experiments and Results.

This section presents the implementation details and comprehensive evaluation of PQCD-TLS, our proposed TLS handshake protocol enhanced with confirmation codes for post-quantum cryptographic schemes. Our primary objective is to systematically quantify the performance overhead introduced by confirmation codes under various algorithm combinations and authentication modes.

6.1 Choice of Primitives

In our experimental evaluation, we systematically select a comprehensive suite of post-quantum cryptographic primitives to assess the efficacy of confirmation code mechanisms in enhancing protocol security. The chosen algorithms span three critical cryptographic categories: Key Encapsulation Mechanisms (KEM), digital signatures (SIG), and Message

Authentication Codes (MAC), each augmented with confirmation codes whose parameters are detailed in Table 1.

KEM Selection. We incorporate six KEM algorithms: three parameterized variants of ML-KEM (ML-KEM$_C$-512/768/1024) based on the Module Learning with Errors (MLWE) problem, two parameterized variants of DAWN-α (DAWN$_C$-α-512/1024) based on NTRU, and HQC$_C$-128 (code-based). The confirmation codes for ML-KEM$_C$ and HQC$_C$ are generated according to [31], but are directly output as error detection results at the KEM level, analogous to the implementation of DAWN$_C$ in Section 3.4. **Signature Selection.** For signatures, we evaluate ML-DSA$_C$-44 and Falcon$_C$-512. Both algorithms utilize a 1-byte confirmation code, consistent with the compactness requirements of signature schemes. **MAC Selection.** For message authentication, we employ HMAC$_C$-SHA3-256, which uses the SHA-3(Keccak) [48] construction. The confirmation code is extracted as the first byte of the internal hash, following the approach of [24]. The upper bound on its unpredictability is guaranteed by Theorem 4.2 of the same reference.

In Table 1, the unpredictability bound for each algorithm quantifies the adversarial advantage in predicting or forging valid confirmation codes. These primitives will be integrated into a TLS implementation for practical evaluation.

6.2 Evaluation

Building on the implementation platform described above, we quantify the overhead introduced by confirmation code (cd) binding in post-quantum TLS. Our evaluation follows a controlled-variable design: for each cryptographic configuration, we compare (i) a baseline PQ-TLS handshake and (ii) an otherwise identical handshake with confirmation code binding enabled. This pairing isolates the performance impact of cd binding from other factors.

Comparison Methods and Metrics. We instantiate PQCD-TLS with the post-quantum primitives in Table 1. For each algorithm combination, we evaluate both unidirectional and mutual authentication under TLS 1.3. This yields 24 comparison groups, each with confirmation code binding enabled and disabled, for a total of 48 configurations. For each configuration, we measure the end-to-end TLS 1.3 handshake latency, defined as the time elapsed from handshake initiation to the completion of key establishment and message verification. Timing is collected using a high-resolution clock and reported in milliseconds. Each test case is repeated 100 times. We perform a warm-up phase and execute configurations in randomized order to mitigate ordering bias.

Handshake Times. For each of the 24 paired comparison groups, we compute the relative overhead $\Delta = (T_{\text{cd}} - T_{\text{base}})/T_{\text{base}}$, where T_{cd} and T_{base} denote the average handshake latency with and without confirmation code binding, respectively. The results are shown in Table 2. Across all groups, confirmation code binding introduces less than 1% additional handshake latency in every case, indicating practical deployability with negligible runtime cost.

6.3 Implementation Error Detection

To evaluate the practical error-detection capability of confirmation-code binding, we inject five representative implementation faults into the TLS handshake, covering signature verification bypass, signature parameter/input errors, KEM re-encryption-check bypass, and Finished-message HMAC verification bypass. For each fault, we compare three execution modes: a correct implementation, a faulty implementation without confirmation code, and the same faulty implementation with confirmation code binding enabled.

Our experiments show that several faults remain silent in the baseline handshake, allowing protocol execution to continue despite incorrect verification behavior. In contrast, when confirmation-code binding is enabled, the faulty execution typically produces a confirmation code inconsistent with that of the correct peer. This mismatch is propagated through the corresponding stage of the key schedule, leading to divergent stage secrets or traffic secrets and eventually causing an observable protocol failure, such as Finished verification failure, AEAD decryption failure, or unsuccessful data exchange. These results demonstrate that confirmation-code binding can effectively transform otherwise silent verification faults into detectable interoperability failures. Detailed fault-injection settings and per-case results are provided in Appendix E.

7 Discussion

7.1 Deployment Considerations

PQCD-TLS is designed to integrate into the existing TLS deployment model with minimal changes. Confirmation codes remain local to each endpoint: they are not transmitted as protocol messages and do not change the TLS wire format. The required modifications are limited to cryptographic interfaces that expose confirmation codes and to the local key schedule that consumes them.

This design supports two deployment strategies. One option is to keep confirmation code processing enabled during and after the post-quantum transition as a permanent safety net, so that later implementation changes can still expose silent verification errors. PQCD-TLS can also be used as a *pluggable testing mechanism* for detecting implementation errors. In this mode, confirmation-code processing is negotiated through a dedicated TLS extension in `ClientHello` and `ServerHello`. When the extension is enabled by both endpoints, the handshake carries and checks confirmation-code information to expose mismatches caused by faulty verification logic. This allows implementers to selectively enable PQCD-TLS during development, integration testing, or interoperability testing.

Table 1: Cryptographic Algorithms with Confirmation Codes

KEM Algorithms				KEM Algorithms				SIG & MAC Algorithms			
Type	**Algorithm**	**cd (bytes)**	**Unpred. Bound**	**Type**	**Algorithm**	**cd (bytes)**	**Unpred. Bound**	**Type**	**Algorithm**	**cd (bytes)**	**Unpred. Bound**
KEM	ML-KEM$_C$-512	12	0.029	KEM	DAWN$_C$-α-512	13	0.042	SIG	ML-DSA$_C$-44	1	0.013
	ML-KEM$_C$-768	16	0.005		DAWN$_C$-α-1024	20	0.011		Falcon$_C$-512	1	0.012
	ML-KEM$_C$-1024	20	0.028		HQC$_C$-128	1	0.004	MAC	HMAC$_C$-SHA3-256	1	0.008

Note: cd = Confirmation Code. **Unpred. Bound** represents the upper bound of the adversary's advantage against the unpredictability of the confirmation code.

Table 2: Performance Evaluation Results for TLS Confirmation Code Binding (24 Comparison Groups)

Group	KEM	Signature	Auth	w/o cd (ms)	with cd (ms)	Diff. (%)
1	ML-KEM$_C$-512	ML-DSA$_C$-44	Uni	311.867	312.834	+0.310
2	ML-KEM$_C$-512	ML-DSA$_C$-44	Mut	312.856	313.673	+0.261
3	ML-KEM$_C$-512	Falcon$_C$-512	Uni	319.853	322.105	+0.704
4	ML-KEM$_C$-512	Falcon$_C$-512	Mut	319.919	321.251	+0.417
5	ML-KEM$_C$-768	ML-DSA$_C$-44	Uni	310.392	310.711	+0.103
6	ML-KEM$_C$-768	ML-DSA$_C$-44	Mut	311.725	312.482	+0.243
7	ML-KEM$_C$-768	Falcon$_C$-512	Uni	320.392	321.786	+0.435
8	ML-KEM$_C$-768	Falcon$_C$-512	Mut	320.636	320.849	+0.066
9	ML-KEM$_C$-1024	ML-DSA$_C$-44	Uni	311.002	311.767	+0.246
10	ML-KEM$_C$-1024	ML-DSA$_C$-44	Mut	314.377	315.471	+0.348
11	ML-KEM$_C$-1024	Falcon$_C$-512	Uni	322.529	323.281	+0.233
12	ML-KEM$_C$-1024	Falcon$_C$-512	Mut	325.269	326.274	+0.309
13	HQC$_C$-128	ML-DSA$_C$-44	Uni	317.696	318.127	+0.136
14	HQC$_C$-128	ML-DSA$_C$-44	Mut	318.876	319.856	+0.307
15	HQC$_C$-128	Falcon$_C$-512	Uni	324.530	326.058	+0.471
16	HQC$_C$-128	Falcon$_C$-512	Mut	325.322	326.120	+0.245
17	DAWN$_C$-α-512	ML-DSA$_C$-44	Uni	313.524	313.585	+0.020
18	DAWN$_C$-α-512	ML-DSA$_C$-44	Mut	315.880	316.002	+0.038
19	DAWN$_C$-α-512	Falcon$_C$-512	Uni	320.292	320.904	+0.191
20	DAWN$_C$-α-512	Falcon$_C$-512	Mut	320.522	321.305	+0.244
21	DAWN$_C$-α-1024	ML-DSA$_C$-44	Uni	311.834	312.228	+0.126
22	DAWN$_C$-α-1024	ML-DSA$_C$-44	Mut	312.728	312.849	+0.039
23	DAWN$_C$-α-1024	Falcon$_C$-512	Uni	322.361	323.745	+0.429
24	DAWN$_C$-α-1024	Falcon$_C$-512	Mut	324.327	325.913	+0.489

Note: Performance difference (Diff.) is calculated as: $\frac{\text{with cd} - \text{w/o cd}}{\text{w/o cd}} \times 100\%$. Auth: Uni = Unidirectional, Mut = Mutual. All latency values are in milliseconds

Other TLS variants require more targeted analysis. For hybrid TLS handshakes [21], the confirmation-code mechanism can be applied at the protocol level. However, extending it to the classical component requires additional analysis of possible verification errors in classical algorithms and suitable confirmation-code extraction points [14, 52]. For the record layer, the main challenge is silent AEAD verification failure, which would require extracting confirmation codes from AEAD verification and identifying suitable binding points in traffic-key updates or record-processing state. The approach is also compatible with KEMTLS [54] and DTLS [53].

7.2 Limitations

PQCD-TLS targets honest-but-faulty implementations of cryptographic verification logic. We do not claim to cover all possible implementation errors; instead, our formal analysis focuses on the predefined classes of silent verification faults modeled in this work. Moreover, PQCD-TLS does not address malicious implementations, which are outside our scope. A malicious implementation may compute confirmation codes consistently while deliberately bypassing the corresponding verification logic.

PQCD-TLS exposes errors by making confirmation codes affect key derivation. Therefore, the mechanism cannot detect implementation errors that do not affect the verification step computation. For example, weak keys, randomness failures or incorrect key generation may compromise security without necessarily changing the verification intermediate values from which confirmation codes are extracted. Similarly, if both endpoints use the same faulty implementation and compute the same erroneous confirmation code, the handshake may still derive matching keys. Implementer should therefore be deployed together with standard defenses such as constant-time implementations, fuzzing testing and formal verification.

8 Conclusion

We presented a two-layer framework for detecting implementation errors during post-quantum migration. At the algorithm layer, we constructed generic confirmation-code mechanisms for both Fiat–Shamir and GPV-style Hash-and-Sign signatures, and gave upper-layer adapted confirmation-code KEMs, with formal unpredictability guarantees. At the protocol layer, we designed PQCD-TLS, which binds these confirmation codes into the TLS 1.3 key schedule so that verification faults propagate into stage-key mismatches and observable handshake failures. Our implementation shows that this mechanism introduces less than 1% handshake latency overhead. Beyond TLS, the approach of tying confirmation codes to protocol key derivation can be extended to other security protocols such as SSH [38], IPsec [27], IKEv2 [36] and Signal [34], offering a unified strategy for surfacing silent implementation errors across the post-quantum protocol stack.

References

[1] CVE-2022-21449: Vulnerability in oracle java SE and oracle graalvm enterprise edition. Common Vulnerabilities and Exposures (CVE) Record, April 2022. Published: April 19, 2022. URL: https://www.cve.org/CVERecord?id=CVE-2022-21449.

[2] CVE-2025-48946: Iibogs affected by theoretical design flaw in HQC. Common Vulnerabilities and Exposures (CVE) Record, May 2025. URL: https://www.cve.org/CVERecord?id=CVE-2025-48946.

[3] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 2014. URL: https://doi.org/10.1007/978-3-319-04852-9_2.

[4] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology - CRYPTO 2006, Santa Barbara, California, USA, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, August 2006. URL: https://doi.org/10.1007/11818175_36.

[5] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO93, Santa Barbara, California, USA, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1993. URL: https://doi.org/10.1007/3-540-48329-2_21.

[6] Mihir Bellare and Björn Tackmann. The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Santa Barbara, CA, USA, August 14-18, 2016,Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 247–276, 2016. URL: https://doi.org/10.1007/978-3-662-53018-4_10.

[7] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime. *IACR Cryptol. ePrint Arch.*, page 461, 2016. URL: http://eprint.iacr.org/2016/461.

[8] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In *EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018. URL: https://doi.org/10.1109/EuroSP.2018.00032.

[9] Jacqueline Brendel, Marc Fischlin, and Felix Günther. Breakdown resilience of key exchange protocols: Newhope, TLS 1.3, and hybrids. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 521–541. Springer, 2019. URL: https://doi.org/10.1007/978-3-030-29962-0_25.

[10] Christina Brzuska. *On the Foundations of Key Exchange*. PhD thesis, Darmstadt University of Technology, Germany, 2013. URN: urn:nbn:de:tuda-tuprints-34147. URL: http://tuprints.ulb.tu-darmstadt.de/3414/.

[11] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of bellare-rogaway key exchange protocols. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 51–62. ACM, October 2011. URL: https://doi.org/10.1145/2046707.2046716.

[12] Jinrong Chen, Biming Zhou, Rongmao Chen, Haodong Jiang, Yi Wang, Xinyi Huang, Yunlei Zhao, and Moti Yung. Post-quantum TLS 1.3 handshake from cpa-secure kems with tighter reductions. In *EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, Proceedings, Part II*, Lecture Notes in Computer Science, pages 499–529. Springer. URL: https://doi.org/10.1007/978-3-032-25317-0_17.

[13] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*, volume 10831 of *Lecture Notes in Computer Science*, pages 282–305. Springer, 2018. URL: https://doi.org/10.1007/978-3-319-89339-6_16.

[14] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.1. RFC 4346, IETF, April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507, 7919. URL: https://www.rfc-editor.org/rfc/rfc4346.txt.

[15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1197–1210. ACM, October 2015. URL: https://doi.org/10.1145/2810103.2813653.

[16] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 34(4):1–37, October 2021. URL: https://doi.org/10.1007/s00145-021-09384-1.

[17] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Paper 2016/081, 2016. URL: https://eprint.iacr.org/2016/081.

[18] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013. URL: https://doi.org/10.1007/978-3-642-40041-4_3.

[19] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, March 2018. URL: https://doi.org/10.13154/tches.v2018.i1.238-268.

[20] Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A simpler, parallelizable, maskable variant of falcon. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part III*, volume 13277 of *Lecture Notes in Computer Science*, pages 222–253. Springer, 2022. doi:10.1007/978-3-031-07082-2_9.

[21] Stephen Farrell. Post-Quantum Guidance for TLS. Internet-Draft draft-farrell-tls-pqg-00, IETF, December 2024. Work in progress. URL: https://www.ietf.org/archive/id/draft-farrell-tls-pqg-00.html.

[22] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, August 1986. URL: https://doi.org/10.1007/3-540-47721-7_12.

[23] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of google's QUIC protocol. In *CCS 2014, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1193–1204. ACM, November 2014. URL: https://doi.org/10.1145/2660267.2660308.

[24] Marc Fischlin and Felix Günther. Verifiable verification in cryptographic protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 3239–3253. ACM, November 2023. doi:10.1145/3576915.3623151.

[25] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON: Fast-fourier lattice-based compact signatures over NTRU. Specification v1.2, 2020. URL: https://falcon-sign.info/.

[26] Pierre-Alain Fouque, Paul Kirchner, Thomas Pornin, and Yang Yu. BAT: small and fast KEM over NTRU lattices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):240–265, 2022. URL: https://doi.org/10.46586/tches.v2022.i2.240-265.

[27] Sheila Frankel and Suresh Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071, February 2011. Informational; obsoletes RFC 2411. URL: https://www.rfc-editor.org/info/rfc6071, doi:10.17487/RFC6071.

[28] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, March 2013. URL: https://doi.org/10.1007/s00145-011-9114-1.

[29] Philippe Gaborit, Carlos Aguilar Melchor, Nicolas Aragon, and et al. Hamming quasi-cyclic (HQC), fourth round version. Algorithm specification, NIST Post-Quantum Cryptography Standardization Project, February 2025. Updated version 19/02/2025. URL: https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf.

[30] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 197–206.

ACM, May 2008. URL: https://doi.org/10.1145/1374376.1374407.

[31] Lewis Glabush, Felix Günther, Kathrin Hövelmanns, and Douglas Stebila. Verifiable decapsulation: Recognizing faulty implementations of post-quantum KEMs. In *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part III*, volume 16002 of *Lecture Notes in Computer Science*, pages 543–574. Springer, August 2025. doi:10.1007/978-3-031-01881-6_17.

[32] Lewis Glabush, Patrick Longa, Michael Naehrig, Chris Peikert, Douglas Stebila, and Fernando Virdia. FrodoKEM: A CCA-secure learning with errors key encapsulation mechanism. Cryptology ePrint Archive, Paper 2025/1861, 2025. URL: https://eprint.iacr.org/2025/1861, doi:10.62056/ayivom2hd.

[33] Felix Günther. Modeling advanced security aspects of key exchange and secure channel protocols. *it Inf. Technol.*, 62(5-6):287–293, 2020. URL: https://doi.org/10.1515/itit-2020-0029.

[34] Keitaro Hashimoto, Shuichi Katsumata, and Thom Wiggers. Bundled authenticated key exchange: A concrete treatment of signal's handshake protocol and {Post-Quantum} security. In *USENIX Security 25*, pages 6777–6796, 2025.

[35] Nadia Heninger. Biased nonce sense: Lattice attacks against weak ecdsa signatures in the wild. Talk at the Workshop on Attacks in Cryptography 2 (WAC2), Crypto 2019, August 2019. URL: https://crypto.iacr.org/2019/affevents/wac/medias/Heninger-BiasedNonceSense.pdf.

[36] Jun Hu, Yasufumi Morioka, and Guilin Wang. Post-Quantum Traditional (PQ/T) Hybrid PKI Authentication in the Internet Key Exchange Version 2 (IKEv2). Internet-Draft draft-hu-ipsecme-pqt-hybrid-auth-04, IETF, February 2026. URL: https://datatracker.ietf.org/doc/draft-hu-ipsecme-pqt-hybrid-auth/.

[37] Loïs Huguenin-Dumittan and Serge Vaudenay. On ind-qcca security in the ROM and its applications - CPA security is sufficient for TLS 1.3. In *EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, Proceedings, Part III*, Lecture Notes in Computer Science, pages 613–642. Springer. URL: https://doi.org/10.1007/978-3-031-07082-2_22.

[38] Panos Kampanakis, Douglas Stebila, and Tony Hansen. Post-quantum Hybrid Key Exchange in SSH. Internet-Draft draft-kampanakis-curdle-ssh-pq-ke-01, IETF, April 2023. URL: https://www.ietf.org/archive/id/draft-kampanakis-curdle-ssh-pq-ke-01.html.

[39] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO 2010, August 15-19, 2010, Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, August 2010. URL: https://doi.org/10.1007/978-3-642-14623-7_34.

[40] Hugo Krawczyk. A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *CCS 2016, Vienna, Austria*, pages 1438–1450. ACM, 2016. URL: https://doi.org/10.1145/2976749.2978325.

[41] Xinyu Li, Jing Xu, Zhenfeng Zhang, Dengguo Feng, and Honggang Hu. Multiple handshakes security of TLS 1.3 candidates. In *IEEE Symposium on Security and Privacy 2016, San Jose, CA, USA, May 22-26, 2016*, pages 486–505. IEEE Computer Society, 2016. URL: https://doi.org/10.1109/SP.2016.36.

[42] Yijian Liu, Yu Zhang, Xianhui Lu, Yao Cheng, and Yongjian Yin. DAWN: Smaller and faster NTRU encryption via double encoding. *IACR Cryptology ePrint Archive*, page 1520, September 2025. URL: https://eprint.iacr.org/2025/1520.

[43] Yijian Liu, Yu Zhang, Xianhui Lu, Yao Cheng, and Yongjian Yin. DAWN: smaller and faster NTRU encryption via double encoding. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology - ASIACRYPT 2025 - 31st International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, VIC, Australia, December 8-12, 2025, Proceedings, Part III*, Lecture Notes in Computer Science, pages 396–427. Springer, 2025. doi:10.1007/978-981-95-5099-9_13.

[44] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. LAC: practical ring-lwe based public-key encryption with byte-level modulus. *IACR Cryptol. ePrint Arch.*, page 1009, 2018. URL: https://eprint.iacr.org/2018/1009.

[45] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009. doi:10.1007/978-3-642-10366-7_35.

[46] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012, Cambridge, UK. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012. URL: https://doi.org/10.1007/978-3-642-29011-4_43.

[47] MITRE Corporation. Cve-2025-69277. Common Vulnerabilities and Exposures (CVE). [Online; accessed 13-Apr-2026]. URL: https://www.cve.org/CVERecord?id=CVE-2025-69277.

[48] National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical Report 202, August 2015. URL: https://doi.org/10.6028/NIST.FIPS.202.

[49] National Institute of Standards and Technology. FIPS 204: Module-lattice-based digital signature standard. Federal Information Processing Standards (FIPS) Publication, August 2023. URL: https://doi.org/10.6028/NIST.FIPS.204.

[50] Christopher Patton and Thomas Shrimpton. Partially specified channels: The TLS 1.3 record layer without elision. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *CCS 2018, Toronto, ON, Canada*, pages 1415–1428. ACM, 2018. URL: https://doi.org/10.1145/3243734.3243789.

[51] Kevin Poulsen. Behind iPhone's critical security bug, a single bad "goto". Wired, February 2014. Accessed: April 27, 2023. URL: https://www.wired.com/2014/02/gotofail/.

[52] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, Internet Engineering Task Force (IETF), August 2018. Obsoletes RFCs 5077, 5246, 6961; Updates RFCs 5705, 6066. URL: https://www.rfc-editor.org/rfc/rfc8446.txt.

[53] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147, RFC Editor, April 2022. URL: https://www.rfc-editor.org/rfc/rfc9147.html, doi:10.17487/RFC9147.

[54] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS 2020, Virtual Event, USA, November 9-13, 2020*, pages 1461–1480. ACM, 2020. URL: https://doi.org/10.1145/3372297.3423350.

[55] Yang Yu, Huiwen Jia, Leibo Li, Delong Ran, Zhiyuan Qiu, Shiduo Zhang, Xiuhan Lin, and Xiaoyun Wang. Hufu: Hash-and-sign signatures from powerful gadgets. Algorithm Specifications and Supporting Documentation, 2023. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/HuFu-spec-web.pdf.

[56] Jiang Zhang, Dengguo Feng, and Di Yan. NEV: faster and smaller NTRU encryption using vector decoding. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part VII*, volume 14444 of *Lecture Notes in Computer Science*, pages 157–189. Springer, 2023. doi:10.1007/978-981-99-8739-9_6.

[57] Biming Zhou, Haodong Jiang, and Yunlei Zhao. Cpa-secure kems are also sufficient for post-quantum TLS 1.3. In *ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, Proceedings, Part III*, Lecture Notes in Computer Science, pages 433–464. Springer. URL: https://doi.org/10.1007/978-981-96-0891-1_14.

A Cryptographic Definitions

Definition A.1 (Hash function and collision resistance)**.** *Let $H : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function that maps arbitrary-length messages to fixed-length outputs. We define the* collision resistance *of H as the advantage of an adversary $\mathcal{A}$ in finding two distinct messages that evaluate to the same hash value, and the according advantage function as:*

$$\mathsf{Adv}^{\mathrm{COLL}}_{H,\mathcal{A}} := \Pr\left[(m,m') \leftarrow\$ \mathcal{A} : (m \neq m') \wedge (H(m) = H(m'))\right]$$

Definition A.2 (Pseudorandom Functions)**.** *A pseudorandom funtion $\mathsf{F} : \mathcal{K} \times \mathcal{L} \to O$ maps a key $k \in \mathcal{K}$ and a label $l \in \mathcal{L}$ to an output $o \in O$. We define the PRF-security of a pseudorandom funtion F as*

$$\mathsf{Adv}^{\mathsf{PRF-sec}}_{\mathsf{F},\mathcal{A}} := \left|\Pr\left[k \leftarrow\$ \mathcal{K} : \mathcal{A}^{\mathsf{F}(k,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\mathsf{R}(\cdot)} \Rightarrow 1\right]\right|$$

where $\mathsf{R} : \mathcal{L} \to O$ is a truly random function.

In certain cases, relying solely on the PRF assumption is insufficient. Consequently, it is necessary to introduce stronger security assumptions to ensure the validity of the proof. Following the formulation presented in [4], we assume that the HMAC employed herein can be treated as a dual PRF.

Definition A.3 (Dual PRF security)**.** *A dual pseudorandom funtion $\mathsf{f} : \mathcal{K} \times \mathcal{L} \to O$ maps a key $k \in \mathcal{K}$ and a label $l \in \mathcal{L}$ to an output $o \in O$. We define the dual PRF-security of f as the PRF-security of $\mathsf{f}^{\mathsf{swap}}(k,l) := \mathsf{f}(l,k)$ and the advantage as $\mathsf{Adv}^{\mathsf{dual-PRF-sec}}_{\mathsf{f},\mathcal{A}} := \mathsf{Adv}^{\mathsf{PRF-sec}}_{\mathsf{f}^{\mathsf{swap}},\mathcal{A}}$*

Definition A.4 (Existential unforgeability for signature scheme)**.** *Let* $\mathsf{SIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$ *be a signature scheme and let* $\mathsf{Expt}^{\mathsf{EUF-CMA}}_{\mathsf{Sig},\mathcal{A}}$ *for an adversary* $\mathcal{A}$ *be defined as in Figure 6. We define the advantage of an adversary* $\mathcal{A}$ *against the existential unforgeability under chosen-message attacks of* SIG *as* $\mathsf{Adv}^{\mathsf{EUF-CMA}}_{\mathsf{SIG},\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{EUF-CMA}}_{\mathsf{Sig},\mathcal{A}} \Rightarrow 1\right]$

Definition A.5 (IND-atk security for key encapsulation mechanism)**.** *Let* $\mathsf{KEM} = (\mathsf{KeyGen}, \mathsf{Encap}, \mathsf{Decap})$ *be a key encapsulation mechanism, and let* $\mathsf{Expt}^{\mathsf{IND-atk}}_{\mathsf{KEM},\mathcal{A}}$ *for an adversary* $\mathcal{A}$ *be defined as in Figure 7, where* $\mathsf{atk} \in \{\mathsf{CPA}, \mathsf{1CCA}, \mathsf{CCA}\}$. *We define the advantage of an adversary* $\mathcal{A}$ *against the indistinguishability under* atk *attacks of* KEM *as* $\mathsf{Adv}^{\mathsf{IND-atk}}_{\mathsf{KEM},\mathcal{A}} := \left|\Pr\left[\mathsf{Expt}^{\mathsf{IND-atk}}_{\mathsf{KEM},\mathcal{A}} \Rightarrow 1\right] - \frac{1}{2}\right|$. *We say that* KEM *is* IND-atk *secure if* $\mathsf{Adv}^{\mathsf{IND-atk}}_{\mathsf{KEM},\mathcal{A}}$ *is negligible for every probabilistic polynomial-time adversary* $\mathcal{A}$.

$\mathsf{Expt}^{\mathsf{EUF-CMA}}_{\mathsf{Sig},\mathcal{A}}$

1 : $(sk, pk) \leftarrow\$ \mathsf{KeyGen}()$
2 : $Q \leftarrow \emptyset$
3 : $(m^*, \sigma^*) \leftarrow\$ \mathcal{A}^{O_{\mathsf{Sign}}}(pk)$
4 : $d^* \leftarrow \mathsf{Vrfy}(pk, m^*, \sigma^*)$
5 : **return** $[d^* \wedge m^* \notin Q]$

$O_{\mathsf{Sign}}(m)$

1 : $\sigma \leftarrow\$ \mathsf{Sign}(m, sk)$
2 : $Q \leftarrow Q \cup \{m\}$
3 : **return** σ

Figure 6: EUF-CMA security experiment for signature schemes.

$\mathsf{Expt}^{\mathsf{IND-atk}}_{\mathsf{KEM},\mathcal{A}}$

1 : $(pk, sk) \leftarrow\$ \mathsf{KeyGen}()$
2 : $q_O \leftarrow 0$
3 : $b \leftarrow\$ \{0,1\}$
4 : $(ss_0, ct^*) \leftarrow\$ \mathsf{Encap}(pk)$
5 : $ss_1 \leftarrow\$ \mathcal{K}$
6 : $b' \leftarrow \mathcal{A}^{O_{\mathsf{atk}}}(pk, ct^*, ss_b)$
7 : **return** $[b' = b]$

$O_{\mathsf{CPA}}(ct)$

1 : **return** $\perp$

$O_{\mathsf{1CCA/CCA}}(ct)$

1 : $q_O \leftarrow q_O + 1$
2 : **if** $ct = ct^*$ **then**
3 : **return** $\perp$
4 : **if** $\mathsf{atk} = \mathsf{1CCA}$ **then**
5 : **if** $q_O > 1$ **then**
6 : **return** $\perp$
7 : **return** $\mathsf{Dec}(sk, ct)$

Figure 7: IND-atk security experiment for key encapsulation mechanisms.

We present the confirmation-code-augmented HMAC scheme from [24].The standard HMAC scheme is defined as $\mathsf{HMAC}(K, m) := \mathsf{H}\big(K \oplus \mathsf{opad} \| \mathsf{H}(K \oplus \mathsf{ipad} \| m)\big)$. In enhanced scheme, the intermediate hash value $\mathsf{H}(K \oplus \mathsf{ipad} \| m)$ is output as the confirmation code, and the same value is recomputed during verification. Concretely, in our instantiation, we choose SHA-3 as H and extract only the first byte as the confirmation code. The Unpredictability Analysis of Confirmation Codes is provided in [24].

B Enhanced Version of DAWN

Recall that the DAWN algorithm operates over the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where n is a power of two and q is a prime. In the underlying public-key encryption scheme of DAWN, the encryption process pseudorandomly samples noise terms $s, e \in R_q$ and uses them to compute a polynomial $c \in R_q$. The coefficients of c are then compressed into a smaller range via a modulo reduction and rounding operation to form the final ciphertext. We leverage the information loss inherent in this compression step to construct a confirmation-code-enhanced version of DAWN, denoted DAWN_C. We use the sampling function to generate the noise term $\mathbf{e}$ is treated as the constrained function F within the cUP security notion.[2]

Algorithm 5 $\mathrm{DAWN_C.PKE_C.Enc_C}(pk, m, \rho)$

1: $\mathbf{h} \leftarrow pk$
2: $(\rho_s, \rho_e) \leftarrow \rho$
3: $\mathbf{s} \leftarrow \mathrm{SampleS}(\mathcal{T}^n_{n,k_s}, \rho_s)$
4: $\mathbf{e} \leftarrow \mathrm{SampleE}(\mathcal{T}^n_{n,k_e}, \rho_e)$
5: $u \leftarrow \left(\mathbf{hs} + \mathbf{e} + (x^{n/4} + 1)^{-1} m\right) \bmod (x^n + 1, \mathbb{Z}_q)$
6: $\mathrm{cd} \leftarrow \big(u[s_1], \ldots, u[s_k]\big) Generateconfirmationcode$
7: $c \leftarrow \lfloor u \rceil_{d_c}$
8: **return** (c, cd)

Definition B.1 (Confirmation Code Unpredictability Experiment for PKE Schemes, [31])**.** *Following [31], we define the confirmation code predictability experiment for a public-key encryption scheme with confirmation codes* $\mathsf{PKE}_C = (\mathsf{KeyGen}, \mathsf{Enc}_C, \mathsf{Dec})$. *Let* F *be a function and* $\mathcal{A}$ *be an adversary that makes at most* max *queries to* F. *The experiment* $\mathsf{Expt}^{\mathsf{cUP-PKE}}_{\mathsf{PKE}_C,F,\max,\mathcal{A}}$ *is defined as follows:*

1. $(pk, sk) \leftarrow\$ \mathsf{KeyGen}^F()$
2. $(m, r) \leftarrow\$ \mathcal{M} \times \mathcal{R}$
3. $(c, cd) \leftarrow \mathsf{Enc}^F_C(pk, m; r)$
4. $cd' \leftarrow \mathcal{A}^{\tilde{F}}(pk, sk, c, m, r)$
5. *Return* $[cd = cd']$

We define the advantage of $\mathcal{A}$ *in predicting confirmation codes as:*

$$\mathsf{Adv}^{\mathsf{cUP-PKE}}_{\mathsf{PKE}_C,F,\max,\mathcal{A}} := \Pr\left[\mathsf{Expt}^{\mathsf{cUP-PKE}}_{\mathsf{PKE}_C,F,\max,\mathcal{A}} \Rightarrow 1\right]$$

In this experiment, the adversary $\mathcal{A}$ *is allowed to query the function* F *at most* max *times.*

[2] For simplicity, our analysis considers only the constrained invocation of the sampling function for $\mathbf{e}$, excluding constraints on other functions. This setting already captures scenarios such as the bypassing of re-encryption checks.

It is crucial to emphasize that the models for simulating implementation faults differ between signature schemes and KEMs. This discrepancy arises because signature verification outputs only a decision bit, providing limited feedback on execution details and making faults more prone to remain latent. In contrast, KEMs essentially perform shared key agreement, where faults are more likely to be exposed. However, the re-encryption check in this context can be bypassed, and such vulnerabilities have been demonstrated in prior work. Therefore, we adapt the fault simulation methodology from Glabush *et al.* [31] to model faults in KEMs..

Theorem B.1. *Let DAWN.PKE$_C$ denote the DAWN public-key encryption scheme equipped with a confirmation code, as defined in Algorithm 5, where $s_1,...,s_k \in \{0,...,n-1\}$. Let $\mathcal{A}$ be a non-adaptive adversary in the unpredictability (cUP) experiment for the confirmation code. Then we have:*

$$\mathsf{Adv}^{\text{cUP-PKE}}_{DAWN_C,SampleE,0,\mathcal{A}} \leq (P_{\text{succ}})^k,$$

where $\mathsf{max} = 0$ *means $\mathcal{A}$ cannot query the* SampleE *function and the single-coefficient success probabilities are $P_{\text{succ}} \approx 0.730$ for DAWN-α-512 and $P_{\text{succ}} \approx 0.754$ for DAWN-α-1024.*

According to Corollary 3.1, the KEM scheme of DAWN (denoted DAWN.KEM$_\mathsf{C}$) directly reuses the confirmation code from its underlying DAWN.PKE$_\mathsf{C}$. Consequently, as long as the confirmation code of the underlying PKE satisfies the unpredictability stated in Theorem B.1, the confirmation code of DAWN.KEM$_\mathsf{C}$ inherits the same security bound.

We present the confirmation-code-augmented KEM encapsulation and decapsulation algorithms for DAWN. As shown in Algorithms 6 and 7, the confirmation code is generated by the underlying PKE.Enc algorithm during encapsulation and is recomputed for verification during decapsulation.

Algorithm 6 DAWN$_\mathsf{C}$.KEM$_\mathsf{C}$.Encap$_\mathsf{C}(pk)$

$m \leftarrow M$
$(\overline{K}, \rho) \leftarrow \text{Hash}_m(m, \text{Hash}_{pk}(pk))$
(ct,cd) $\leftarrow$ DAWN$_\mathsf{C}$.PKE$_\mathsf{C}$.Enc$_\mathsf{C}(pk, m, \rho)$
$K \leftarrow \text{Hash}_K(\overline{K}, ct)$
return (ct, K, cd)

Building upon the design paradigm, we adopt the Enc$_\mathsf{C}$ algorithms for ML-KEM and HQC proposed in [31] to construct two corresponding KEM schemes integrated with the confirmation code mechanism. This construction is well-suited for upper-layer protocol invocation requirements, and greatly enhances the compatibility and usability of the schemes in practical protocol deployments.

Algorithm 7 DAWN$_\mathsf{C}$.KEM$_\mathsf{C}$.Decap$_\mathsf{C}(sk, ct, pk)$

$(K', sk', pk, \text{H}_{pk}) \leftarrow sk$
$m \leftarrow$ DAWN.PKE.Dec(ct, sk')
$(\overline{K}, \rho) \leftarrow \text{Hash}_m(m, \text{H}_{pk})$
(ct',cd') $\leftarrow$ DAWN$_\mathsf{C}$.PKE$_\mathsf{C}$.Enc$_\mathsf{C}(pk, m, \rho)$
if $ct = ct'$ **then**
  $K \leftarrow \text{Hash}_K(\overline{K}, ct')$
else
  $K \leftarrow \perp, cd' \leftarrow \perp$
end if
return (K, cd')

C cUP Analysis

C.1 Preservation of EUF-CMA Security

Here we first need to show that the signature algorithm with the added confirmation code can preserve the EUF-CMA security property.

Remark C.1 (Security preservation of confirmation–code-augmented signatures)**.** The confirmation-code augmentation preserves the original EUF-CMA security of the underlying signature scheme. In our security experiments, the confirmation code is treated as a local auxiliary value computed by the honest execution; it is not returned to the adversary as part of the signing-oracle response. Thus, the adversarial interface of Sig$_\mathsf{C}$ exposes the same signature σ as the original signing oracle, while the confirmation code is used only internally for implementation-error detection.

Formally, the augmented signing algorithm produces the same signature σ as Sign, together with an auxiliary confirmation code cd, and the augmented verification algorithm preserves the same acceptance predicate as Vrfy. Therefore, any successful EUF-CMA forgery against Sig$_\mathsf{C}$ immediately yields a valid forgery against the underlying signature scheme Sig. Hence, for every EUF-CMA adversary $\mathcal{A}$ against the confirmation-code-augmented scheme, there exists an EUF-CMA adversary $\mathcal{B}$ against the original scheme with essentially the same running time such that $\mathsf{Adv}^{\text{EUF-CMA}}_{\text{Sig}_\mathsf{C},\mathcal{A}} = \mathsf{Adv}^{\text{EUF-CMA}}_{\text{Sig},\mathcal{B}}$. Consequently, adding confirmation codes does not weaken the standard unforgeability guarantee of the signature scheme.

C.2 ML-DSA cUP-Sig Analysis

Proof. We consider a non-adaptive adversary whose advantage is defined as the probability of winning the confirmation-code unpredictability experiment. The experiment is split into three sub-experiments: type A (programming errors), type B (input errors), and type C (parameter errors).

Part 1: Bounding $\mathsf{Adv}^{\text{cUP}(A)}_{\text{ML-DSA}_\mathsf{C},\mathcal{A}}$ (Programming Errors). The adversary directly guesses the confirmation code cd.In ML-DSA the coefficients of $\mathbf{w}_1$ are uniformly distributed

and independent.Specifically, in ML-DSA-44 each coefficient $w_1[i][j]$ is uniform over $\{0,\ldots,43\}$; after reduction modulo 16 the maximum probability of a single value is $3/44$ (attained for the values $0,1,2,3$). In ML-DSA-65/87 each coefficient $w_1[i][j]$ is uniform over $\{0,\ldots,15\}$; modulo 16 the distribution remains uniform, each value having probability $1/16$.Since cd and cd_0 have the same distribution, the adversary's success probability in sub-experiment A is bounded by $\mathsf{Adv}^{\mathsf{cUP(A)}}_{\mathsf{ML\text{-}DSA}_\mathsf{C},\mathcal{A}} \le (P_{\max})^\ell$, where $P_{\max}$ denotes the maximum probability of a single coefficient modulo 16.

Part 2: Bounding $\mathsf{Adv}^{\mathsf{cUP}(B)}_{\mathsf{ML\text{-}DSA}_\mathsf{C},\mathcal{A}}$ (Input Errors). The adversary modifies an input, which leads to a different recovered $\mathbf{w}_1'$ and thus a different cd_0'.In this context, we adopt a reconstructible commitment that requires the verifier to perform reconstruction. The commitment generation process relies on $\mathbf{y}$, which is randomly generated on each occasion and essentially serves as a kind of salt. Because the modification vector is chosen non-adaptively, cd_0' is independent of the original cd_0 and follows the same distribution.Consequently, the probability that cd_0' coincides with cd_0 does not exceed $P_{\max}^\ell$. Thus, we have $\mathsf{Adv}^{\mathsf{cUP(B)}}_{\mathsf{ML\text{-}DSA}_\mathsf{C},\mathcal{A}} \le (P_{\max})^\ell$.

Part 3: Bounding $\mathsf{Adv}^{\mathsf{cUP}(C)}_{\mathsf{ML\text{-}DSA}_\mathsf{C},\mathcal{A}}$ (Parameter Errors). The adversary alters the parameter $\gamma_1 - \beta$ to a value P^*, which changes the XOR constant. We conservatively assume that the non-adaptive adversary picks the modification δ_{par} uniformly at random. The winning condition requires that the modified norm bound be congruent to the original one modulo $2^{4\ell}$; this occurs with probability exactly $1/2^{4\ell}$.Thus, we have $\mathsf{Adv}^{\mathsf{cUP(C)}}_{\mathsf{ML\text{-}DSA}_\mathsf{C},\mathcal{A}} \le \frac{1}{2^{4\ell}}$.

Combining the three parts. Summing the bounds for the three sub-experiments yields: $\mathsf{Adv}^{\mathsf{cUP}}_{\mathsf{ML\text{-}DSA}_\mathsf{C},\mathcal{A}} \le 2\cdot(P_{\max})^k + 1/2^{4\ell}$, where $P_{\max} = 3/44$ for ML-DSA-44 and $P_{\max} = 1/16$ for ML-DSA-65/87. □

C.3 Falcon cUP-Sig Analysis

Proof. We bound the advantages of the three sub-experiments separately.

Part 1: Bounding $\mathsf{Adv}^{\mathsf{cUP}(A)}_{\mathsf{Falcon}_\mathsf{C},\mathcal{A}}$ (Programming Errors). In type A, the adversary directly replaces the confirmation code with a value $\delta_{\mathsf{cd}} \neq \perp$. The winning condition is $\delta_{\mathsf{cd}} = \mathsf{cd}$. Since the adversary chooses δ_{cd} without any information about cd, and due to the modulo operation, cd is distributed nearly uniformly over $\{0,1,\ldots,2^\ell-1\}$. The adversary's best strategy is to guess uniformly at random, with success probability at most $1/2^\ell$. Thus, we have $\mathsf{Adv}^{\mathsf{cUP\text{-}Sig}(A)}_{\mathsf{Falcon}_\mathsf{C},\mathcal{A}} \le 2^{-k}$.

Part 2: Bounding $\mathsf{Adv}^{\mathsf{cUP}(B)}_{\mathsf{Falcon}_\mathsf{C},\mathcal{A}}$ (Input Errors). In sub-experiment B, the adversary modifies the input (e.g., the message, signature, or public key) and may also modify the parameter β. Let cd be the original confirmation code and cd^* be the confirmation code computed from the modified input and/or modified parameter. The winning condition is $\mathsf{cd}^* = \mathsf{cd}$ and the verification decision $d = 1$.

Since the modification vector is chosen non-adaptively, cd is independent of the modified inputs and parameters. Moreover, due to the modulo operation, cd is distributed nearly uniformly over $\{0,1,\ldots,2^\ell-1\}$. Therefore, even conditioned on the verification passing ($d = 1$), the probability that cd^* equals the fixed value cd is at most $1/2^\ell$. Hence, we have $\mathsf{Adv}^{\mathsf{cUP\text{-}Sig}(B)}_{\mathsf{Falcon}_\mathsf{C},\mathcal{A}} \le 2^{-\ell}$.

Part 3: Bounding $\mathsf{Adv}^{\mathsf{cUP}(C)}_{\mathsf{Falcon}_\mathsf{C},\mathcal{A}}$ (Parameter Errors). In experiment C, only the parameter β is modified to a value $\beta^* \neq \beta$, while inputs and the confirmation code remain unchanged. The verification algorithm computes $\mathsf{cd}^* = (\beta^*)^2 - \|(s_1,s_2)\|^2 \bmod 2^\ell$. Analogous to our discussion in the ML-DSA context, here we consider honest implementation errors. Introducing a parameter error that does not alter the confirmation code can be viewed as randomly selecting an integer congruent to β. Consequently, we obtain the bound $\mathsf{Adv}^{\mathsf{cUP\text{-}Sig}(C)}_{\mathsf{Falcon}_\mathsf{C},\mathcal{A}} \le 2^{-\ell}$.

Combining the three parts. Summing the bounds for the three sub-experiments yields: $\mathsf{Adv}^{\mathsf{cUP\text{-}Sig}}_{\mathsf{Falcon}_\mathsf{C},\mathcal{A}} \le 3\cdot 2^{-\ell}$. □

C.4 PKE Confirmation Code in KEM

Here we explain that directly taking the confirmation code from PKE as a KEM-level output and allowing upper-layer protocols to invoke this code does not affect the IND-CCA security of the KEM.

Remark C.2 (Local confirmation codes and IND-CCA security)**.** The confirmation code cd is only stored locally and is not transmitted. In particular, it is not given to the IND-CCA adversary. Hence, the IND-CCA experiment for KEM_C is identical to the IND-CCA experiment for the classic KEM obtained by discarding the confirmation code.

Consider the classic KEM KEM obtained by simply ignoring the confirmation code. Its IND-CCA experiment proceeds as follows: the challenger runs $(K,c) \leftarrow \mathsf{Encaps}(pk)$ and gives (pk,c,K_b) to the adversary; for any decapsulation query $c \neq c^*$, the challenger returns $\mathsf{Decaps}(sk,c)$. Here, $\mathsf{Encaps}(pk)$ internally computes $(K,c,cd) \leftarrow \mathsf{Encaps}_\mathsf{C}(pk)$ and outputs only (K,c), while $\mathsf{Decaps}(sk,c)$ internally computes $(K',cd') \leftarrow \mathsf{Decaps}_\mathsf{C}(sk,c)$ and outputs only K'. In the IND-CCA experiment for KEM_C, the challenger computes $(K_0,c^*,cd^*) \leftarrow \mathsf{Encaps}_\mathsf{C}(pk)$, but the adversary only receives (pk,c^*,K_b) and never receives cd^*. Similarly, for any decapsulation query $c \neq c^*$, the challenger computes $(K',cd') \leftarrow \mathsf{Decaps}_\mathsf{C}(sk,c)$ but returns only K' to the adversary. Therefore, the adversary's view in the IND-CCA experiment for KEM_C is exactly the same as its view in the IND-CCA experiment for the classic KEM KEM. Consequently, for any adversary $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}_\mathsf{C},\mathcal{A}} = \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM},\mathcal{A}}$. Thus, extracting confirmation codes as private local auxiliary outputs does not affect the IND-CCA security of the KEM.

We directly adopt the confirmation code produced by PKE_C

inside KEM_C as the output for detecting implementation errors at the KEM level. We demonstrate that this method does not hinder the role of the confirmation codes (Corollary 3.1), with the proof detailed below.

Proof. We construct the PKE adversary $\mathcal{A}$ using the KEM adversary $\mathcal{B}$.

In the KEM cUP experiment, $\mathcal{B}$ receives a public key pk, a ciphertext ct^*, and a session key K^* from an honest encapsulation $(ct^*, K^*) \leftarrow \mathsf{Encap}_\mathsf{C}(pk)$. Its goal is to output cd^* while being unable to query $\mathsf{SampleE}$.

The adversary $\mathcal{A}$ simulates this environment for $\mathcal{B}$ in the PKE cUP experiment as follows:

1. $\mathcal{A}$ receives its PKE inputs (pk, sk, ct, m, ρ) from the PKE cUP experiment, where $ct \leftarrow \mathsf{KEM.PKE}_\mathsf{C}.\mathsf{Enc}_\mathsf{C}(pk, m; \rho)$ for m and ρ.
2. $\mathcal{A}$ computes the session key $K^* = \mathsf{Hash}_K(\overline{K^*}, ct^*)$, where $(\overline{K}, \rho) = \mathsf{Hash}_m(m, \mathsf{Hash}_{pk}(pk))$. Note that $\mathcal{A}$ can compute this because the derivation of $(\overline{K}, \rho)$ from m is deterministic and does not require $\mathsf{SampleE}$.
3. $\mathcal{A}$ provides $\mathcal{B}$ with (pk, sk, ct, K), perfectly simulating the KEM cUP challenge.
4. If $\mathcal{B}$ outputs a confirmation code cd', $\mathcal{A}$ forwards cd' as its own output in the PKE cUP experiment.

The reduction holds because, in the PKE unpredictability experiment, the adversary obtains (pk, sk, c, m, r) and can thus generate (pk, sk, ct, K) required for the KEM unpredictability experiment. Consequently, if the PKE experiment can produce the correct confirmation code under constrained oracle access, it follows that the adversary in the KEM experiment can also produce the same confirmation code. Hence, we obtain:

$$\mathsf{Adv}^{\text{cUP-KEM}}_{\mathsf{KEM}_\mathsf{C},\mathsf{SampleE},max}(\mathcal{B}) \leq \mathsf{Adv}^{\text{cUP-PKE}}_{\mathsf{PKE}_\mathsf{C},\mathsf{SampleE},max}(\mathcal{A}).$$

□

After establishing the unpredictability of the KEM confirmation code, we next clarify its intended role when integrated into upper-layer protocols such as TLS.

Remark C.3 (Role of KEM confirmation codes in upper-layer protocols.)**.** Recent analyses of KEM-based TLS [12, 37, 57] show that, when a KEM is used only for ephemeral key establishment in the TLS handshake, the protocol can be proven secure under assumptions weaker than full IND-CCA security of the KEM. In particular, the re-encryption check is not always essential for this specific use case. Therefore, our integration of KEM confirmation codes should not be interpreted as claiming that such checks are strictly required for ephemeral TLS key exchange.

Rather, our goal is to provide a protocol-level example of how KEM confirmation codes can be exposed and bound into an upper-layer protocol to detect implementation faults. This is especially relevant for settings such as KEMTLS [54], where KEMs are used for authentication and CCA-style decapsulation correctness becomes security-critical. By treating the confirmation codes of [31] as KEM-level outputs and incorporating them into the TLS key schedule, our construction offers a practical template for detecting faulty KEM implementations in real-world protocol deployments.

C.5 DAWN cUP-KEM Analysis

Following the cUP definition introduced in [31], we analyze the unpredictability of confirmation codes in the DAWN context, under the constraint that the adversary cannot simulate implementation errors via $\mathsf{SampleE}$. The cUP security proof is presented below.

Proof. In the confirmation code unpredictability experiment $\mathrm{Expt}^{\text{cUP-PKE}}_{\text{DAWN.PKE}_\mathsf{C},F,\mathsf{max}}$, $\mathcal{A}$ cannot call the function SampleE (i.e., $\mathsf{max} = 0$) and thus cannot directly obtain the noise polynomial $\mathbf{e}$. However, the adversary can compute $\mathbf{z} = \mathbf{hs} + (x^{n/4}+1)^{-1}\mathbf{m}$. For each coefficient $i \in S$, the adversary knows $z_i \in \mathbb{Z}_q$ and the compressed value $c_i = \lfloor z_i + e_i \rceil_{d_c}$, where e_i follows the ternary distribution:

$$\Pr[e_i = 0] = p_0 = \frac{n-2k}{n}, \Pr[e_i = 1] = \Pr[e_i = -1] = p_1 = \frac{k}{n}.$$

The adversary's goal is to guess $u_i = z_i + e_i$, which is equivalent to guessing e_i. Given (z_i, c_i), the adversary can determine the set of noise values consistent with the observation:

$$E_i(z_i, c_i) = \{e \in \{-1, 0, 1\} : \lfloor z_i + e \rceil_{d_c} = c_i\}.$$

The optimal strategy for the adversary is to guess the element in E_i with the highest prior probability. The success probability for coefficient i is:

$$p_i^*(z_i, c_i) = \max_{e \in E_i(z_i,c_i)} \Pr[e_i = e \mid z_i, c_i] = \frac{\max_{e \in E_i(z_i,c_i)} \Pr[e_i = e]}{\sum_{e' \in E_i(z_i,c_i)} \Pr[e_i = e']}.$$

Based on the composition of E_i, we analyze p_i^* in four cases:

1. If $|E_i| = 1$, let $E_i = \{e\}$, then $p_i^* = 1$.
2. If $E_i = \{0, 1\}$ or $E_i = \{0, -1\}$, then $p_i^* = \dfrac{p_0}{p_0 + p_1}$.
3. If $E_i = \{1, -1\}$, then $p_i^* = \dfrac{1}{2}$.
4. If $E_i = \{-1, 0, 1\}$, then $p_i^* = p_0$.

Let $\mathcal{Z}$ be the domain of z_i (i.e., $\mathbb{Z}_q$) and $\mathcal{C}$ the domain of c_i (i.e., $\mathbb{Z}_{2^{d_c}}$). Denote $P(z, c) = \Pr[z_i = z, c_i = c]$. In the experiment, z_i is uniformly distributed over $\mathbb{Z}_q$, and conditioned on z_i, the distribution of c_i is determined by the distribution of e_i and the compression function. By the law of total probability, the probability that the adversary correctly guesses a single coefficient i is: $P_{\text{succ}} = \sum_{(z,c) \in \mathcal{Z} \times \mathcal{C}} P(z, c) \cdot p_i^*(z, c)$. Consequently, the advantage of the adversary in the confirmation code unpredictability experiment is upper-bounded by: $\mathsf{Adv}^{\text{cUP-PKE}}_{\text{DAWN}_\mathsf{C},\mathsf{SampleE},0,\mathcal{A}} \leq (P_{\text{succ}})^k$, where the single-coefficient success probabilities are $P_{\text{succ}} \approx 0.730$ for DAWN-α-512 and $P_{\text{succ}} \approx 0.754$ for DAWN-α-1024. □

D Reductionist Security Analysis

D.1 Security Model

D.1.1 Model Synax.

We distinguish between the security properties of the protocol (such as authentication AUTH) and session-specific attributes (i.e., session state). This separation provides the benefit that protocol-specific properties are global and identical across all sessions, while session-specific attributes (e.g., session state, current stage, generated keys) are maintained independently per session.

We characterize the protocol specification properties by a vector $(\mathsf{M}, \mathsf{FS}, \mathsf{AUTH}, \mathsf{USE})$. Since we consider the protocol like full 1-RTT mode and exclude the pre-shared key mode, no additional replay protection property needs to be incorporated.

- $\mathsf{M} \in \mathbb{N}$: the number of stages.
- $\mathsf{FS} \in \{1, ..., \mathsf{M}, \infty\}$: the stage $j = \mathsf{FS}$ from which on keys are forward secret, and ∞ means no forward security.
- $\mathsf{AUTH} \in \{((u_1, m_1), ..., (u_\mathsf{M}, m_\mathsf{M}) | u_j, m_j \in \{1, ..., \mathsf{M}, \infty\})\}$: the authentication level evolution of stage keys throughout the protocol execution. For each stage i, u_i denotes the stage after which the stage-i key becomes unilaterally authenticated, while m_i indicates the stage after which it becomes mutually authenticated. If a particular authentication level is never achieved, the corresponding value is set to ∞.
- $\mathsf{USE} \in \{\mathsf{internal}, \mathsf{external}\}^\mathsf{M}$: the usage indicator use_i indicates the stage-i key is used internally in the key exchange protocol.

We denote by $\mathcal{U}$ the set of all users in the system and uniquely identify each protocol session by a $\mathsf{lab} \in \mathsf{LABELS} = \mathcal{U} \times \mathcal{U} \times \mathbb{N}$. Here, $\mathsf{lab} = (U, V, n)$ represents the n-th session between the session owner U and its intended communication partner V.

For each session, a tuple containing the following state information is maintained as an entry in the session list List_S. Certain variables maintain distinct values for each stage $i \in \{1, \ldots, M\}$.

- $\mathsf{lab} \in \mathsf{LABELS}$: the unique session label.
- $\mathsf{role} \in \{\mathsf{initiator}, \mathsf{responder}\}$: the role of the session owner. $\mathsf{auth} \in \mathsf{AUTH}$: the intended authentication type from AUTH.
- id: the identity of the session owner.
- $\mathsf{pid} \in \mathcal{U} \cup \{*\}$: the identity of the *intended* communication partner, where the wildcard symbol $'*'$ means that the partner may be unknown.
- $\mathsf{sid} \in (\{0,1\}^* \cup \{\perp\})^\mathsf{M}$: the *session* identifier for stage i, denoted sid_i, is initialized to $\perp$ and is only assigned a value from $\{0,1\}^*$ when the protocol reaches the acceptance state in that stage.
- $\mathsf{cid} \in (\{0,1\}^* \cup \{\perp\})^\mathsf{M}$: the *contributive* identifier for stage i, denoted sid_i, is initialized to $\perp$ and is only assigned a value from $\{0,1\}^*$ when the protocol reaches the acceptance state in that stage.
- $\mathsf{status} \in \{\mathsf{running}, \mathsf{accepted}, \mathsf{rejected}\}^\mathsf{M}$: the status of each stage key. When a session *accepts* the stage-i key, we set to acceptd. When a session *rejects* the stage-i key, we set to acceptd and the protocol stops. When a session *continues* after accepting the stage-i key, we set to running. Initially we set to $(\text{running}, \perp, \perp, \perp, \perp, \perp)$.
- $\mathsf{stage} \in \{0, ..., M\}$: the last accepted stage. Initially set to 0, it is incremented to i when status_i reaches $\mathsf{accepted}$.
- $\mathsf{st}_\mathsf{key} \in \{\mathsf{fresh}, \mathsf{revealed}\}^\mathsf{M}$: characterizes the state of the session key in stage i.
- $\mathsf{revealed} \in \{\mathsf{true}, \mathsf{false}\}^\mathsf{M}$: adversarial exposure indicator variable, where $\mathsf{revealed}_i$ denotes if the stage-i key has been leaked by the adversary. All entries are initialized to false.
- $\mathsf{tested} \in \{\mathsf{true}, \mathsf{false}\}^\mathsf{M}$: adversarial testing indicator variable, where tested_i denotes if the stage-i key has been tested by the adversary. All entries are initialized to false.
- $\mathsf{CD} \subset \{\mathsf{cd}_\mathsf{kem}, \mathsf{cd}_\mathsf{SCV}, \mathsf{cd}_\mathsf{hmac,S}, \mathsf{cd}_\mathsf{CCV}, \mathsf{cd}_\mathsf{hmac,C}, \infty\}$: The confirmation code set for a session. This set contains the confirmation codes that are computed and verified by the owner of this session during the protocol execution. Each confirmation code corresponds to a specific computational step in the protocol. Here, cd_kem denotes the key encapsulation mechanism confirmation code, cd_SCV represents the server certificate verification confirmation code, $\mathsf{cd}_\mathsf{hmac,S}$ indicates the server HMAC confirmation code, cd_CCV corresponds to the client certificate verification confirmation code, and $\mathsf{cd}_\mathsf{hmac,C}$ signifies the client HMAC confirmation code. The symbol ∞ serves as a special placeholder value. Confirmation codes are inherently local to each session and are not shared. The set is initialized as empty and is updated dynamically as the protocol progresses, recording only those confirmation codes that have been successfully computed and verified by the owner of this session.
- $\mathsf{ERROR} \subset \{\mathsf{KEM.error}, \mathsf{SIG.error}, \mathsf{HMAC.error}\}$: We define an *error set* comprising the **silent-yet-critical** steps that are the focus of this work, specifically, those implementation flaws that compromise security without affecting functional correctness. While we do not preclude the existence of errors in other protocol components, our analysis specifically concentrates on three critical steps: the re-encryption check in ephemeral key exchange, and the signature verification and HMAC verification during the authentication phase. We restrict these errors to verification-side implementation faults affecting only one party of an intended peer pair at a given verification location; the corresponding peer execution is assumed to remain correctly implemented. In this work, we provide precise definitions for each error type:
  - $\mathsf{KEM.error} \triangleq \mathsf{KEM}.\mathit{bypass}$: denotes the KEM re-encryption bypass error, i.e., a malicious implemen-

tation skips or tampers with the KEM decapsulation process (corresponding to the cUP proof in Definition B.1).
- $\mathsf{SIG.error} \triangleq \{\mathsf{SIG}.\mathit{program}, \mathsf{SIG}.\mathit{input}, \mathsf{SIG}.\mathit{param}\}$: denote programming, input, and parameter errors in signature operations, respectively (corresponding to the error classifications in Definition 3.1).
- $\mathsf{HMAC.error} \triangleq \{\mathsf{HMAC}.\mathit{program}, \mathsf{HMAC}.\mathit{input}\}$: denote programming and input errors in HMAC operations, respectively (corresponding to the error classifications in [24, Theorem 4.2]).

In the multi-stage model, we demonstrate that individual steps may contain distinct error types. For the confirmation code security analysis within isolated modules, we consider specific error classifications; however, in this context, we simply use boolean indicators to denote the presence or absence of errors in each step.

Furthermore, we establish two distinct analytical modes. For **matching security** and **multi-stage security** analysis, we assume an error-free implementation, formally represented as $\mathsf{ERROR} = \emptyset$. For assessing **implementation error detection capability**, we require $\mathsf{ERROR} \neq \emptyset$.

It is important to emphasize that our Confirmation Code Unpredictability (cUP) proof only guarantees that when an error in ERROR occurs, the confirmation codes generated by the two parties differ with noticeable probability, thereby causing session key mismatch through the key derivation chain. This guarantee does not extend to error types outside ERROR.

D.1.2 Adversary Interactions.

Following the DFGS model [15], we define two core security properties in the standard adversarial interaction model: *matching security* and *multi-stage security*. The adversary $\mathcal{A}$ is modeled as a probabilistic algorithm that controls all communication channels, with capabilities to intercept, inject, or drop messages at will. We explicitly restrict $\mathcal{A}$ from winning the security game by merely obtaining specific session keys, as such a win would lack cryptographic significance.

The adversary has access to a set of queries for interacting with protocol sessions. However, certain combinations of these queries are explicitly prohibited, as they would allow $\mathcal{A}$ to trivially win the test challenge by revealing and testing specific session keys. The adversary $\mathcal{A}$ interacts with the protocol through the following queries:

- $\mathsf{NewSession}(U, V, \mathsf{role}, \mathsf{auth})$: the adversary $\mathcal{A}$ creates a new session with identifier lab, setting the session owner $\mathsf{id} = U$, the intended communication partner $\mathsf{pid} = V$, and the authentication type auth.
- $\mathsf{Send}(\mathsf{lab}, m)$: the adversary $\mathcal{A}$ issues a Send query with message m to session lab. If no session with label lab exists, return $\perp$.
  The Send query can be used to initiate a session, where the protocol is executed on behalf of identity U in response to message m, returning the response and updating the execution state $\mathsf{lab.st_{exec}}$. The adversary $\mathcal{A}$ may also set an initialization message init. If the protocol is already executing and has reached state $\mathsf{accepted_i}$, the protocol immediately terminates and the adversary only receives $\mathsf{accepted_i}$. This restriction prevents trivial wins in the distinguishing game, as decrypting randomly encrypted messages would inevitably yield garbled content. The adversary may issue a $\mathsf{Send}(\mathsf{lab}, \mathsf{continue})$ query to resume protocol execution and receive the updated status.
  Furthermore, if a partnered session has been subjected to a Test query, the current session must set $\mathsf{lab.tested} \leftarrow \mathsf{true}$ to maintain consistency. If the session key for stage i has been accepted and the session is corrupted, the session state must be updated to $\mathsf{lab.st_{key,i}} \leftarrow \mathsf{revealed}$.
- $\mathsf{Reveal}(\mathsf{lab}, i)$: the adversary $\mathcal{A}$ issues a query to reveal the session key $\mathsf{lab.key_i}$ and updates the key state to $\mathsf{lab.st_{key,i}} \leftarrow \mathsf{revealed}$. If the specified session does not exist or the session key for stage i has not been accepted, the query returns $\perp$.
- $\mathsf{Corrupt}(U)$: the adversary $\mathcal{A}$ can obtain the long-term private key of user U, and we record the time at which user U is corrupted.
- $\mathsf{Test}(\mathsf{lab}, i)$: The adversary $\mathcal{A}$ issues a Test query for stage i. If session lab does not exist, or $\mathsf{lab.status} \neq \mathsf{accepted}$, or the session has already been marked with $\mathsf{lab.tested_i} \leftarrow \mathsf{true}$, return $\perp$.
  If the stage-i key is for internal use ($\mathsf{USE_i} = \mathsf{internal}$) and there exists a partnered session lab' (with $\mathsf{lab.sid_i} = \mathsf{lab}'.\mathsf{sid_i}$) but $\mathsf{lab}'.\mathsf{st_{exec}} \neq \mathsf{accepted_i}$, set $\mathsf{lost} \leftarrow \mathsf{true}$[3]. This prevents the adversary from testing a key while its partner is about to use a different (real) key, which would lead to protocol state inconsistency.
  If the tested session lacks verifiable authentication guarantees, i.e., according to the protocol's AUTH property, the session has not yet achieved the required minimum authentication level in stage i, or for responder-role sessions, no honest initiator session exists with a matching contributive identifier $\mathsf{cid_i}$, also set $\mathsf{lost} \leftarrow \mathsf{true}$.
  If none of the above checks trigger lost, set $\mathsf{lab.tested_i} \leftarrow \mathsf{true}$. The challenger then returns a key based on the secret test bit b_{test}: If $b_{\mathsf{test}} = 1$, return the real key $K \leftarrow \mathsf{lab.key_i}$. If $b_{\mathsf{test}} = 0$, sample a random key $K \leftarrow\!\!\$\ \mathcal{D}$ from the key distribution.
  For internally-used keys ($\mathsf{USE_i} = \mathsf{internal}$), overwrite $\mathsf{lab.key_i}$ with the returned key K to ensure consistency in subsequent protocol execution. For externally-used keys, only return the key without modifying the session state. Additionally, if there exists a partnered session lab' that

[3] The lost flag demarcates non-triviality in the security game, preventing adversaries from "cheating" to win. Once lost is set to true, the adversary loses the security game regardless of subsequent actions or final output.

has also accepted the stage-i key ($\mathsf{lab.sid_i} = \mathsf{lab'.sid_i}$ and both have status $\mathsf{accepted_i}$), synchronously set $\mathsf{lab'.tested_i} \leftarrow \mathsf{true}$ and, for $\mathsf{USE_i} = \mathsf{internal}$, synchronize the key value $\mathsf{lab'.key_i} \leftarrow \mathsf{lab.key_i}$ to maintain state consistency between partnered sessions after testing. Return the key key_i to the adversary $\mathcal{A}$.

- $\mathsf{Inject}(\mathsf{U}, \mathsf{i}, \mathsf{j}, \mathsf{error})$: Injects an implementation error into user U. This injection does not represent an active attack in the traditional network security sense, but rather simulates the presence of an implementation flaw in user U's code. We require that the error type error must be one of those defined in 3.1, B.1, and Definition 3.1 of [24]. The adversary can selectively inject the corresponding type of error between stages i and j. An injection is admissible only if it does not simultaneously affect the same verification point in the corresponding peer session. Thus, each fault is evaluated against a correctly implemented peer. The effect achieved by the adversary is the existence of a session with $\mathsf{lab.sid} = U$, $\mathsf{lab.cd} \neq \emptyset$, and $\mathsf{lab.ERROR} \neq \emptyset$. This indicates that the adversary has successfully simulated an implementation error in user U.

D.2 Security Analysis

D.2.1 Protocol Properties.

We analyze the PQCD-TLS protocol in the public-key multi-stage key exchange model with implementation errors. First, we present the properties of the protocol:

- $\mathsf{M} = 6$: the protocol comprises six stages. In the first and second stages, both parties generate the client handshake traffic key tk_{chs} and the server handshake traffic key tk_{shs}, along with the client application traffic secret CATS, server application traffic secret SATS, exporter master secret EMS, and resumption master secret RMS.
- $\mathsf{FS} = 1$: the protocol ensures forward secrecy for all keys derived.
- $\mathsf{AUTH} = \{(3,m),(3,m),(3,m),(4,m),(5,m),(6,m) | m \in \{6,\infty\}\}$: Stages 1 and 2 complete an ephemeral key exchange using temporary public-private key pairs and thus provide no authentication. Only after Stage 3 concludes do the parties begin identity authentication, achieving unilateral authentication upon completion of Stage 3. It is not until Stage 6 concludes that mutual authentication is established, elevating the authentication level to mutual.
- $\mathsf{USE} = (\mathsf{internal} : \{1,2\}, \mathsf{external} : \{3,4,5,6\})$: The session keys derived in Phases 1 and 2 will be used to encrypt subsequent handshake messages, while the keys from Phases 3 and 5 will be used to encrypt post-handshake application data (external to the handshake protocol itself). The keys from Phases 4 and 6 are intended for subsequent key derivation and ticket derivation. Therefore, only the keys from Phases 1 and 2 are considered internal to the handshake protocol.

D.2.2 Session Identifiers

Next, we define the session identifiers for each stage. When a session accepts a given stage, its session identifier is set based on the stage session key and the handshake messages.

We now define the session identifiers for each stage. When a session accepts a given stage, its session identifier is set based on the stage session key and the handshake messages. In Stage 1, the contributive identifier cid is set to ("CHTS", ClientHello) when the client and server process the ClientHello message. Only after processing the ServerHello message does cid become equal to the session identifier sid. In all subsequent stages, cid is always set equal to sid.

$$\mathsf{sid_1} = (\text{"CHTS"}, \mathsf{ClientHello} \ldots \mathsf{ServerHello})$$
$$\mathsf{sid_2} = (\text{"SHTS"}, \mathsf{ClientHello} \ldots \mathsf{ServerHello})$$
$$\mathsf{sid_3} = (\text{"SATS"}, \mathsf{ClientHello}, \ldots, \mathsf{ServerFinished})$$
$$\mathsf{sid_4} = (\text{"EMS"}, \mathsf{ClientHello}, \ldots, \mathsf{ServerFinished})$$
$$\mathsf{sid_5} = (\text{"CATS"}, \mathsf{ClientHello}, \ldots, \mathsf{ClientFinished})$$
$$\mathsf{sid_6} = (\text{"RMS"}, \mathsf{ClientHello}, \ldots, \mathsf{ClientFinished})$$

D.3 Matching Security

Definition D.1 (Matching security)**.** *Let* KE *be a* M*-stage key exchange protocol with properties* (M, AUTH, FS, USE) *and let* $\mathcal{A}$ *be a probabilistic polynomial-time (PPT) algorithm with* KE *via the queries defined in Section D.1 in the following experiment* $\mathrm{Expt}^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}}$*:*

Setup. *The challenger generates long-term public and private key pairs* (pk_u, sk_u) *for each participant* $u \in \mathcal{U}$*.*

Query. *The adversary* $\mathcal{A}$ *has access to the queries* NewSession, Send, Reveal, Corrupt, Test*.*

Stop. *At some point, the adversary* $\mathcal{A}$ *stops with no output.*

Let lab, lab′ *are two distinct labels such that* $\mathsf{lab.sid}_i = \mathsf{lab'.sid} \neq \perp$ *for some stage* $i \in \{1,\ldots,\mathsf{M}\}$ *and* $\mathsf{lab.ERROR} = \mathsf{lab'.ERROR} = \emptyset$*. If at least one of the following conditions holds, we say that* $\mathcal{A}$ *wins the game, denoted* $\mathrm{Expt}^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1$*.*

1. *Different* ***confirmation codes*** *in some stage of* lab *and* lab′*, such that* $\mathsf{lab.cd} \neq \mathsf{lab'.cd}$ *with* $i \in \{1,\ldots,\mathsf{M}\}$*.*
2. *Different* ***session key*** *in some stage of* lab *and* lab′*, such that* $\mathsf{lab.key}_i \neq \mathsf{lab'.key}_i$*.*
3. *Different* ***authentication types*** *in some stage of* lab *and* lab′*, such that* $\mathsf{lab.auth}_i \neq \mathsf{lab'auth}_i$*.*
4. *Different or unset* ***contributive identifiers*** *in some stage of* lab *and* lab′*, such that* $\mathsf{lab.cid}_i \neq \mathsf{lab'cid}_i$ *or* $\mathsf{lab.cid}_i = \mathsf{lab'cid}_i = \perp$*.*
5. *Non-opposite* ***roles*** *of the partnered sessions in some stage, such that* $\mathsf{lab.role} = \mathsf{lab'.role}$*.*
6. *Different* ***stages*** *share the same session identifier, such that* $\mathsf{lab.sid}_i = \mathsf{lab'.sid}_j \neq \perp$ *for some stage* $i, j \in \{1,\ldots,\mathsf{M}\}$ *with* $i \neq j$*.*

7. ***More than two sessions*** *share the same session identifier, such that* $\mathsf{lab.sid}_i = \mathsf{lab}'.\mathsf{sid}_i = \mathsf{lab}''.\mathsf{sid}_i \neq \perp$ *for some stage* $i, j \in \{1, ..., \mathsf{M}\}$.

We say KE *is* Match − secure *if all PPT adversary* $\mathcal{A}$ *the following advantage function is negligible in the security parameter:*

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} := \Pr\left[\mathrm{Expt}^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1\right]$$

The matching security theorem for the PQCD-TLS protocol has been stated in the main text Theo 5.1. We now provide the proof of this theorem.

Proof. We proceed to demonstrate that the protocol satisfies all requirements specified in Definition D.1:

1. *Sessions with the same session identifier for some stage hold the same confirmation code at that stage.*
   According to the definition of confirmation codes, we require that correctly executed cryptographic algorithms with confirmation codes must generate identical outputs. However, in this scenario, if the key encapsulation mechanism (KEM) during the ephemeral key exchange phase has a small probability of correctness error, only in this case would the input to the re-encryption algorithm during decapsulation differ from the input to the encryption algorithm during encapsulation, leading to different confirmation codes. This results in an error probability of $n_s\delta_e$.
2. The session identifier at each stage incorporates all handshake messages influencing key derivation, the shared secret generated by the KEM, and all data intended for confirmation code generation in the current stage. Since the session identifier uniquely determines all inputs to key derivation, any two sessions sharing the same sid must possess identical stage keys. Furthermore, as the error probability of the KEM has already been accounted for in our analysis of confirmation codes, we omit its consideration here.
3. In stage 1/2, the ephemeral key exchange via KEM is unauthenticated, so the authentication states of both sessions are inherently consistent. In stage 3/4, server certificate authentication is completed. Since the session identifiers sid_3 and sid_4 incorporate all messages up to ServerFinished, these sessions reside in the unilateral authentication stage and share the same authentication type. In stage 5/6, client certificate authentication is finalized. As the session identifier sid_5 and sid_6 encompasses all messages up to ClientFinished, these sessions operate under mutual authentication and exhibit identical authentication types. Consequently, the authentication type of the session is uniquely determined by the session identifier, ensuring that consistent session identifiers imply consistent authentication levels.
4. By definition, when a session accepts a stage key, cid is identical to the corresponding sid.
5. Any two sessions sharing the same session identifier must have complementary roles as initiator and responder. A session establishment will fail if role assignment is inconsistent.
6. By definition, each stage's session identifier is a unique label, precluding the occurrence of such a scenario.
7. If there exist three or more honest sessions sharing the same sid, this implies that one session must have identical nonces and KEM key exchanges with another session, constituting a collision between honest clients and servers. Following [54], we disregard KEM-specific considerations and analyze the collision bound under the birthday attack. The probability of such collisions occurring among n_s honest sessions is at most $n_s^2/2^{|\mathsf{nonce}|}$.

□

D.4 Muliti-Stage Security

Definition D.2 (Multi − Stage security of PQCD-TLS)**.** *Let* KE *be a multi-stage key exchange protocol with properties* (M, AUTH, FS, USE) *and key distribution* $\mathcal{D}$*, and* $\mathcal{A}$ *be a PPT adversary interacting with* KE *via the queries defined in secure model in the following game* $\mathrm{Expt}^{\mathsf{Multi-stage},\mathcal{D}}_{\mathsf{KE},\mathcal{A}}$*:*

Setup. *The challenger chooses a test bit* $b_{\mathsf{test}} \leftarrow\!\!\$\ \{0,1\}$*, sets* $\mathsf{lost} \leftarrow \mathsf{false}$*, enerates long-term public and private key pairs* (pk_u, sk_u) *for each participant* $u \in \mathcal{U}$ *and the* $\mathsf{ERROR} = \emptyset$*.*

Query. *The adversary* $\mathcal{A}$ *has access to the queries* NewSession, Send, Reveal, Corrupt, Test*. Note that certain queries may set* $\mathsf{lost} \leftarrow \mathsf{true}$*.*

Guess. *At some point,* $\mathcal{A}$ *stops and outputs a guess* b_{guess}*.*

Finalize. *If there exist* $\mathsf{lab}, \mathsf{lab}'$ *and some stage* $i \in \{1, ..., \mathsf{M}\}$ *such that* $\mathsf{lab.sid}_i = \mathsf{lab}'.\mathsf{sid}_i$*,* $\mathsf{lab.st}_{\mathsf{key},\mathsf{i}} = \mathsf{revealed}$*,* $\mathsf{lab}'.\mathsf{tested}_i = \mathsf{true}$ *and* $\mathsf{lab.ERROR}/\mathsf{lab}'.\mathsf{ERROR} \neq \emptyset$*, the challenger will set* $\mathsf{lost} \leftarrow \mathsf{true}$*.* [3]

We say that $\mathcal{A}$ *wins the game, denoted by* $\mathrm{Expt}^{\mathsf{Multi-stage},\mathcal{D}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1$*, if* $b_{\mathsf{guess}} = b_{\mathsf{test}}$ *and* $\mathsf{lost} = \mathsf{false}$*. We say* KE *is* Multi − Stage *security with properties* (M, AUTH, FS, USE) *if* KE *is* Match *security and for all PPT adversaries the following advantage function is negligible in the security parameter:*

$$\mathsf{Adv}^{\mathsf{Multi-Stage},\mathcal{D}}_{\mathsf{KE},\mathcal{A}} := \Pr\left[\mathrm{Expt}^{\mathsf{Multi-Stage},\mathcal{D}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1\right] - \frac{1}{2}$$

Theorem D.1 (Multi − Stage security of PQCD-TLS)**.** *The PQCD-TLS is* Multi − Stage *secure with* (M, AUTH, FS, USE)*. For any PPT adversary* $\mathcal{A}$ *against the* Multi − Stage *security there exist efficient algorithms*

[3] We require that the test session and its matching session remain uncorrupted; otherwise, the adversary could trivially win the security game.

$\mathcal{B}_1, ..., \mathcal{B}_7$ *such that*

$$\mathsf{Adv}^{\text{Multi-Stage}}_{\text{PQCD-TLS},\mathcal{A}} \leq \frac{n_s^2}{2^{|\text{nonce}|}} + \mathsf{Adv}^{\text{COLL}}_{\mathsf{H},\mathcal{B}_1} + n_s\delta_e + 6n_s \cdot \begin{pmatrix} n_s \cdot \begin{pmatrix} 2\mathsf{Adv}^{\text{IND-1CCA}}_{\mathsf{KEM_C},\mathcal{B}_2} + \frac{q_G}{|\mathcal{K}|} + 2\mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_3} \\ + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_4} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_5} \end{pmatrix} \\ + n_u \cdot \mathsf{Adv}^{\text{EUF-CMA}}_{\mathsf{SIG_C},\mathcal{B}_6} \end{pmatrix}.$$

where n_s is the number of sessions, n_u is the number of users, $\mathcal{K}$ denotes the KEM shared secret space, the ephemeral KEM is assumed to be δ_e-correct, q_G is the maximum number of queries to the random oracle G.

Proof. We prove that the adversary's advantage in winning the distinguishing game is negligible through a sequence of game transitions.

Game 0. This is the original $\mathsf{Multi-Stage}$ game that,

$$\mathsf{Adv}^{\mathsf{Multi-Stage},\mathcal{D}}_{\text{PQCD-TLS},\mathcal{A}} = \mathsf{Adv}^{G_0}_{\text{PQCD-TLS},\mathcal{A}}$$

Game 1. (Eliminate nonces collision) We exclude the event that any two honest sessions generate colliding nonces r_C or r_S. Based on the birthday attack, we bound the collision probability among n_s sessions by:

$$\mathsf{Adv}^{G_0}_{\mathcal{A}} \leq \mathsf{Adv}^{G_1}_{\mathcal{A}} + \frac{n_s^2}{2^{|nonce|}}$$

Game 2. (Eliminate hash collision) In this game, the challenger will abort if any two honest sessions compute the same hash for different inputs of hash funciton H. If this happens, it induces a reduction $\mathcal{B}_1$ that can break the collision-resistance of H.

$$\mathsf{Adv}^{G_1}_{\mathcal{A}} \leq \mathsf{Adv}^{\text{COLL}}_{\mathsf{H},\mathcal{B}_1} + \mathsf{Adv}^{G_2}_{\mathcal{A}}$$

Game 3 (Eliminate KEM correctness failures). In this game, the challenger aborts if any honest ephemeral KEM execution encounters a correctness failure. Since the ephemeral KEM is δ_e-correct and there are at most n_s honest KEM executions, we have,

$$\mathsf{Adv}^{G_2}_{\mathcal{A}} \leq \mathsf{Adv}^{G_3}_{\mathcal{A}} + n_s\delta_e.$$

Game 4. We now restrict $\mathcal{A}$ to making only a single Test query. Following the hybrid argument from Günther [33], the advantage for the six stages can be reduced to at most $\frac{1}{6n_s}$. Thus, for any adversary $\mathcal{A}_1$ that randomly selects one session and makes only one Test query, all other Test queries in the original adversary $\mathcal{A}$ can be replaced with Reveal queries.

$\mathcal{A}_1$ needs to identify how sessions are partnered. However, in the protocol, while sid_1 and sid_2 are transmitted in cleartext, sid_3, sid_4, sid_5, and sid_6 are encrypted using session keys. The adversary can only obtain these later identifiers by revealing the corresponding session keys.

Through the reduction from $\mathcal{A}$ to $\mathcal{A}_1$, we obtain the following advantage relation:

$$\mathsf{Adv}^{G_3}_{\mathcal{A}} \leq 6n_s \cdot \mathsf{Adv}^{G_4}_{\mathcal{A}_1}$$

We now focus on stage i of session lab, and assume that the session number is known.

Case Analysis: We now distinguish between two cases (which essentially correspond to the separation between authentication and confidentiality).

- **Case A:** The tested session lab has an honest contributive partnered session in stage 1, i.e., there exists $\mathsf{lab}' \neq \mathsf{lab}$ such that $\mathsf{lab.cid}_1 = \mathsf{lab}'.\mathsf{cid}_1$.
- **Case B:** The tested session lab does not have an honest contributive partnered session in stage 1, i.e., no such lab' exists with $\mathsf{lab.cid}_1 = \mathsf{lab}'.\mathsf{cid}_1$.

We correspond **Case A** to **Game A.0** and correspond **Case B** to **Game B.0**. The adversary's advantage can be decomposed according to the two cases:

$$\mathsf{Adv}^{G_3}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A0}}_{\mathcal{A}_1} + \mathsf{Adv}^{G_{B0}}_{\mathcal{A}_1}$$

Case A. Test session has a contributive partner.

Game A.1 (Guess contributive partner session). In this game, the challenger first guesses a session $\mathsf{lab}' \neq \mathsf{lab}$ such that $\mathsf{lab}'.\mathsf{cid}_i \neq \mathsf{lab.cid}_i$; otherwise, the game aborts. Consequently, the adversary's advantage is confined to a single session:

$$\mathsf{Adv}^{G_{A0}}_{\mathcal{A}_1} \leq n_s \cdot \mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1}$$

Game A.2 (Patching G). Following [57], we model $G(\mathsf{dES}, x) := \mathsf{HKDF.Extract}(\mathsf{dES}, x)$ as a programmable random oracle. Hence, the handshake secret is written as $\mathsf{HS} = G(\mathsf{dES}, ss_e \parallel \mathsf{cd_{kem}})$. This game only modifies the simulation of the KEM execution associated with the fixed test session lab and its contributive partner lab'; all other sessions are simulated exactly as in Game A.1.

Let $m^\star$ be the internal message of this target KEM execution. The simulator deterministically computes $(ct^\star, cd^\star) \leftarrow \mathsf{Enc}_C(pk_e, m^\star)$, $ss^\star \leftarrow \mathsf{KDF}(m^\star, pk_e)$ and associates the target handshake secret with $G(\mathsf{dES}^\star, ss^\star \parallel cd^\star)$. We assume deterministic confirmation-code generation, where $cd_m = \mathsf{Code}(pk_e, m)$ denotes the code produced by $\mathsf{Enc}_C(pk_e, m)$. Thus, for fixed (pk_e, m), the corresponding confirmation code is uniquely determined.

The simulator maintains the standard lazy-sampling table $\mathcal{L}_G$ for G and an auxiliary bookkeeping list $\mathcal{L}_D$, recording the target association as $(ct^\star, ss^\star, cd^\star, Y^\star)$, where $Y^\star$ is the corresponding random-oracle output.

Effectively, the simulation switches the target association from the decapsulation-side computation $G(\mathsf{dES}^\star, \mathsf{KDF}(\mathsf{Dec}(sk_e, ct^\star), pk_e) \parallel \mathsf{Code}(pk_e, \mathsf{Dec}(sk_e, ct^\star)))$ to the encapsulation-side computation $G(\mathsf{dES}^\star, \mathsf{KDF}(m^\star, pk_e) \parallel \mathsf{Code}(pk_e, m^\star))$, $ct^\star = \mathsf{Enc}(pk_e, m^\star)$, following the patching strategy of Game 2 in [31].

Since KEM correctness failures have already been excluded in Game 3, the decapsulation- and encapsulation-side values of this honest target execution coincide. Thus, the above

change only rewrites the association used to represent the same random-oracle value and does not alter the adversary's view. Therefore,

$$\mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1} = \mathsf{Adv}^{G_{A2}}_{\mathcal{A}_1}.$$

Game A.3 (Random HS). In this game, we replace the handshake secret of the fixed test session lab and its contributive partner lab',$\mathsf{HS}^\star = G(\mathsf{dES}^\star, ss_e^\star \parallel \mathsf{cd}^\star_{\mathsf{kem}})$, with an independent uniformly random value $\widetilde{\mathsf{HS}} \xleftarrow{\$} \{0,1\}^\lambda$. All subsequent values in these two partnered sessions are computed using $\widetilde{\mathsf{HS}}$. To bound this transition, we construct an IND-1CCA adversary $\mathcal{B}_2$. It receives a challenge $(pk_e, ct^\star, ss_b)$, where ss_b is either the real shared secret associated with $ct^\star$ or an independent uniform value from $\mathcal{K}$. The challenge reveals neither the internal message $m^\star$ nor the corresponding confirmation code $\mathsf{cd}^\star_{\mathsf{kem}} = \mathsf{Code}(pk_e, m^\star)$.

The simulator samples $\widetilde{\mathsf{HS}} \xleftarrow{\$} \{0,1\}^\lambda$ and creates an unresolved target entry $(ct^\star, ss_b, \bot, \widetilde{\mathsf{HS}})$ in $\mathcal{L}_D$. The test session and its partnered session use $\widetilde{\mathsf{HS}}$ throughout the remainder of the protocol simulation. At this point, the simulator neither knows nor guesses $m^\star$ or $\mathsf{cd}^\star_{\mathsf{kem}}$.

Whenever a candidate message m appears in the underlying KEM simulation, the simulator computes $(ct_m, cd_m) \leftarrow \mathsf{Enc}_C(pk_e, m)$. If $ct_m = ct^\star$, then m identifies the challenge ciphertext and the simulator obtains $\mathsf{cd}^\star_{\mathsf{kem}} = cd_m$. If the corresponding random-oracle point has not previously been queried, the simulator patches $G(\mathsf{dES}^\star, ss_b \parallel \mathsf{cd}^\star_{\mathsf{kem}}) = \widetilde{\mathsf{HS}}$, and updates the target entry in $\mathcal{L}_D$ accordingly.

Define QUERY as the event that, before $\mathsf{cd}^\star_{\mathsf{kem}}$ is resolved, the adversary queries G at a point $(\mathsf{dES}^\star, ss_b \parallel cd)$ for some cd. If QUERY does not occur, the challenge random-oracle point is still undefined when the patching takes place, and the programming is perfectly consistent with lazy sampling.

When $b = 1$, the value $ss_b \xleftarrow{\$} \mathcal{K}$ is independent of the adversary's view. Hence, for at most q_G queries to G, $\Pr[\mathsf{QUERY} \mid b = 1] \le \frac{q_G}{|\mathcal{K}|}$. Let $p_0 = \Pr[\mathsf{QUERY} \mid b = 0]$, $p_1 = \Pr[\mathsf{QUERY} \mid b = 1]$. The reduction $\mathcal{B}_2$ distinguishes the two IND-1CCA branches whenever the corresponding shared-secret component is queried, and thus $\frac{|p_0 - p_1|}{2} \le \mathsf{Adv}^{\mathsf{IND\text{-}1CCA}}_{\mathsf{KEM}_C,\mathcal{B}_2}$. Therefore, $\Pr[\mathsf{QUERY} \mid b = 0] \le 2\mathsf{Adv}^{\mathsf{IND\text{-}1CCA}}_{\mathsf{KEM}_C,\mathcal{B}_2} + \frac{q_G}{|\mathcal{K}|}$.

Conditioned on QUERY not occurring, the adversary never observes the challenge random-oracle point $(\mathsf{dES}^\star, ss_e^\star \parallel \mathsf{cd}^\star_{\mathsf{kem}})$. By lazy sampling, its output is therefore an independent uniformly random λ-bit value from the adversary's perspective, and replacing it with $\widetilde{\mathsf{HS}}$ does not change the adversary's view. Consequently,

$$\mathsf{Adv}^{G_{A2}}_{\mathcal{A}_1} \le \mathsf{Adv}^{G_{A3}}_{\mathcal{A}_1} + 2\mathsf{Adv}^{\mathsf{IND\text{-}1CCA}}_{\mathsf{KEM}_C,\mathcal{B}_2} + \frac{q_G}{|\mathcal{K}|}.$$

Game A.4 (Random CHTS, SHTS). In this game, we replace both CHTS and SHTS in the test session lab and its partnered session lab' with random strings $\widetilde{\mathsf{CHTS}}$ and $\widetilde{\mathsf{SHTS}}$. This substitution affects the subsequent generation of tk_{chs} and tk_{shs}, so we replace tk_{chs}, tk_{shs} with $\widetilde{tk_{chs}}, \widetilde{tk_{shs}} \leftarrow\$ \{0,1\}^{len}$, where len denotes the sum of the key length and IV length required by the subsequent AEAD scheme.

The $\mathsf{DeriveTK}$ function invokes $\mathsf{HKDF.Expand}$, while the generation of MS invokes $\mathsf{HKDF.Extract}$. If $\mathcal{A}_1$ can distinguish between **GameA.3** and **GameA.4**, then $\mathcal{B}_3$ can win two independent PRF-sec games. Specifically, if the outputs are real then they correspond to **GameA.3**, and if the outputs are random then they correspond to **GameA.4**. Thus:

$$\mathsf{Adv}^{G_{A3}}_{\mathcal{A}_1} \le \mathsf{Adv}^{G_{A4}}_{\mathcal{A}_1} + 2\mathsf{Adv}^{\mathrm{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_3}$$

Game A.5 (Random dHS) In this game, we replace dHS with $\widetilde{\mathsf{dHS}}$, which in turn affects the derivation of MS. Consequently, we replace MS with $\widetilde{\mathsf{MS}} \leftarrow\$ \{0,1\}^\lambda$.

If $\mathcal{A}_1$ can distinguish between **GameA.4** and **GameA.5**, then $\mathcal{B}_4$ can win the PRF-sec game corresponding to the MS computation. Specifically, if the output is real then it corresponds to **GameA.4**, and if the output is random then it corresponds to **GameA.5**. Thus:

$$\mathsf{Adv}^{G_{A4}}_{\mathcal{A}_1} \le \mathsf{Adv}^{G_{A5}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathrm{PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_4}$$

Game A.6 (Random MS). In this game, we replace the master secret MS with a uniformly random value $\widetilde{\mathsf{MS}}$. This affects all secrets derived from MS, namely EMS, SATS, CATS, and RMS.

The derivation of these secrets additionally incorporates the authentication confirmation codes, including cd_{SCV}, cd_{CCV}, $cd_{hmac,S}$, and $cd_{hmac,C}$. Importantly, this transition does not rely on the secrecy of these confirmation codes. For signature confirmation codes, we rely on the security-preservation argument established above: given the public verification tuple (pk, m, σ), the corresponding confirmation code can be deterministically recomputed by executing Vrfy_C. Hence, its use in subsequent key derivation provides no additional information beyond the signature transcript. For HMAC confirmation codes, we rely on the security result of [24], which shows that HMAC remains pseudorandom even when the inner-hash value used as the confirmation code is available to the adversary. Therefore, these confirmation codes may be conservatively treated as auxiliary inputs known to the adversary without weakening the respective underlying security guarantees.

Consequently, the confirmation-code components are kept fixed and consistently simulated in both games, while the secret input inherited from MS is replaced by a random value. By the dual-PRF security of $\mathsf{HKDF.Expand}$, the resulting values $\mathsf{SATS}, \mathsf{EMS}, \mathsf{CATS}, \mathsf{RMS}$ are indistinguishable from independently uniform values $\widetilde{\mathsf{SATS}}, \widetilde{\mathsf{EMS}}, \widetilde{\mathsf{CATS}}, \widetilde{\mathsf{RMS}}$.

Thus, if $\mathcal{A}_1$ distinguishes **Game A.5** from **Game A.6**, we construct a reduction $\mathcal{B}_5$ against the dual-PRF security of

HKDF.Expand, Thus:

$$\mathsf{Adv}^{G_{A5}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A6}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathrm{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_5}.$$

At this point, we have replaced all stage keys in the test session lab with random strings. Consequently, the adversary gains no advantage in guessing the test bit b_{test}:

$$\mathsf{Adv}^{G_{A6}}_{\mathcal{A}_1} = 0$$

Case B. Test session does not have a contributive partner in Stage 1. In this case, the test session lacks a partnered session in stage 1, indicating that the adversary can impersonate the other party to interact with the test session owner. We now impose restrictions on the timing of the adversary's Test query. If the test session is a client session, i.e., lab.role = initiator, the Test query cannot be issued before stage 3, and the server must not be corrupted prior to the test; otherwise, the adversary is considered to have failed. (This prevents the adversary from trivially winning by obtaining the server's long-term private key and forging certificates for authentication.). If the test session is a server session, i.e., lab.role = responder, the Test query can only be issued at stage 6, and the client must not be corrupted.

GameB.1 (Guess Contributive Partner). In this game, the challenger first guesses the target user identity $U \in \mathcal{U}$ of the test session lab and aborts if the guess is incorrect. This restricts the adversary to impersonating a single specific user. Therefore:

$$\mathsf{Adv}^{G_{B0}}_{\mathcal{A}_1} \leq n_u \cdot \mathsf{Adv}^{G_{B1}}_{\mathcal{A}_1}.$$

GameB.2 (Signature Forgery). In this game, we define the abort condition as follows: the challenger aborts if the test session lab accepts a signature (either *CCV* or *SCV*) for the current message from user U. We can reduce the abort probability in **GameB.2** to the EUF-CMA security of the signature scheme SIG via adversary $\mathcal{B}_7$.

Adversary $\mathcal{B}_7$ is given the certificate public key pk_U of user U. All message-signature pairs obtained by $\mathcal{A}_1$ during other protocol executions can be viewed as multiple queries made by $\mathcal{B}_7$ to its signing oracle. If lab accepts a signature that was not previously generated by the signing oracle, $\mathcal{B}_7$ outputs it as a successful forgery. Crucially, we require that user U has not previously signed this specific message. Thus:

$$\mathsf{Adv}^{G_{B1}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B2}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathrm{EUF-CMA}}_{\mathsf{SIG_C},\mathcal{B}_6}.$$

In **Case B**, the adversary cannot issue a Test query, as subsequent stage sessions would be rejected if the adversary cannot break the signature scheme $\mathsf{SIG_C}$. Consequently, the adversary gains no advantage in guessing the test bit b_{test}.

$$\mathsf{Adv}^{G_{B2}}_{\mathcal{A}_1} = 0.$$

In summary, we obtain the final bound on the adversary's advantage:

$$\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{PQCD\text{-}TLS},\mathcal{A}} \leq \frac{n_s^2}{2^{|\mathsf{nonce}|}} + \mathsf{Adv}^{\mathrm{COLL}}_{\mathsf{H},\mathcal{B}_1} + n_s\delta_e + 6n_s \cdot \begin{pmatrix} n_s \cdot \begin{pmatrix} 2\mathsf{Adv}^{\text{IND-1CCA}}_{\mathsf{KEM_C},\mathcal{B}_2} + \frac{q_G}{|\mathcal{K}|} + 2\mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_3} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_4} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_5} \end{pmatrix} \\ +n_u \cdot \mathsf{Adv}^{\text{EUF-CMA}}_{\mathsf{SIG_C},\mathcal{B}_6} \end{pmatrix}.$$

□

D.5 MCIE Security

Definition D.3 (Matching correctness with implementation error)**.** *Let* KE *be a* M*-stage key exchange protocol with properties* (M, AUTH, FS, USE) *and let* $\mathcal{A}$ *be a probabilistic polynomial-time (PPT) algorithm with* KE *via the queries defined in Section D.1 in the following game* $\mathrm{Expt}^{\mathsf{MCIE}}_{\mathsf{KE},\mathcal{A}}$*:*

Setup. *The challenger generates long-term public and private key pairs* (pk_u, sk_u) *for each participant* $u \in \mathcal{U}$*.*
Query. *The adversary* $\mathcal{A}$ *has access to the queries* NewSession, Send, Reveal, Corrupt, Test, Inject. Inject *only permits one-sided verification faults, while the corresponding peer execution remains correct.*
Stop. *At some point, the adversary* $\mathcal{A}$ *stops with no output.*

We say that the adversary $\mathcal{A}$ *wins the experiment, denoted by* $\mathrm{Expt}^{\mathsf{MCIE}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1$*, if there exist two (distinct) session labels* lab, lab′ *and a stage* $i \in \{1,\ldots,\mathsf{M}\}$ *such that* lab.ERROR $\neq \emptyset \vee$ lab′.ERROR $\neq \emptyset$, lab.sid$_i$ = lab′.sid$_i \neq \perp$*, the confirmation code associated with the injected implementation error is an input to the derivation of the stage-i key, both sessions have completed stage-i and* lab.key$_i$ = lab′.key$_i$*. The advantage of* $\mathcal{A}$ *is defined as :*

$$\mathsf{Adv}^{\mathsf{MCIE}}_{\mathsf{KE},\mathcal{A}} := \Pr[\mathrm{Expt}^{\mathsf{MCIE}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1].$$

Theorem D.2 (Matching correctness with implementation error of PQCD-TLS)**.** *The PQCD-TLS is* MCIE *secure with* (M, AUTH, FS, USE)*. For any PPT adversary* $\mathcal{A}$ *against the* MCIE *security there exist efficient algorithms* $\mathcal{B}_1,\ldots,\mathcal{B}_6$ *such that*

$$\mathsf{Adv}^{\mathsf{MCIE}}_{\mathsf{PQCD-TLS},\mathcal{A}} \leq \mathsf{Adv}^{\mathrm{COLL}}_{\mathsf{H},\mathcal{B}_1} + n_s \cdot \delta_e + n_e \cdot \begin{pmatrix} \mathsf{Adv}^{\text{cUP-KEM}}_{\mathsf{KEM_C},\mathcal{B}_2} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_3} \\ +\mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_4} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_5} \\ +\mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_6} \end{pmatrix}.$$

where n_s *is the number of sessions, the* n_e *is the number of users whose implementations contain errors, and the ephemeral KEM is assumed to be* δ_e*-correct.*

Proof. We use the Game-based approach to conduct the security proof. **Game 0.** This is the original MCIE game that,

$$\mathsf{Adv}^{\mathsf{MCIE}}_{\mathsf{PQCD-TLS},\mathcal{A}} = \mathsf{Adv}^{G_0}_{\mathsf{PQCD-TLS},\mathcal{A}}.$$

Game 1. (Eliminate hash collision) In this game, the challenger aborts if any two honest sessions compute the same hash value for different inputs in any evaluation of H. If this happens, it induces a reduction $\mathcal{B}_1$ that can break the collision-resistance of H.

$$\mathsf{Adv}^{G_0}_{\mathcal{A}} \leq \mathsf{Adv}^{\mathrm{COLL}}_{\mathsf{H},\mathcal{B}_1} + \mathsf{Adv}^{G_1}_{\mathcal{A}}.$$

Game 2. (Eliminate inherent KEM correctness errors) In this game, the challenger aborts if any honestly generated key encapsulation results in a correctness failure (i.e., the decapsulated key does not match the encapsulated one). Such a failure would prevent two honest parties from deriving the same shared secret regardless of any adversarial interference, thereby affecting the security experiment. We exclude this scenario by bounding the probability of encountering such an inherent KEM fault across all sessions.

$$\mathsf{Adv}^{G_1}_{\mathcal{A}} \leq n_s \cdot \delta_e + \mathsf{Adv}^{G_2}_{\mathcal{A}}.$$

Game 3. (Guess an error user) Let $\mathcal{U}_{\mathsf{error}} \subseteq \mathcal{U}$ denote the set of users whose implementations contain flaws. In this game, the challenger guesses a user $U \leftarrow\!\!\$\ \mathcal{U}_{\mathsf{error}}$ uniformly at random. The game proceeds only if the eventual matching-failure event (as defined in MCIE) involves a session owned by U; otherwise the challenger aborts. Since the adversary's view is independent of this guess until the event occurs, we have

$$\mathsf{Adv}^{G_2}_{\mathcal{A}} \leq n_e \cdot \mathsf{Adv}^{G_3}_{\mathcal{A}}.$$

Game 4. (Exclude KEM-stage errors) In this game, the challenger handles the case where the guessed user U possesses a KEM-stage error. The game proceeds as in Game 3, but with an additional abort event: if any two sessions $\mathsf{lab}, \mathsf{lab}'$ reach the stage $1/2$ where $\mathsf{lab.sid}_i = \mathsf{lab}'.\mathsf{sid}_i \neq \bot$, $\mathsf{KEM.error} \in \mathsf{lab.ERROR}$, and $\mathsf{lab.cd}_{\mathsf{kem}} = \mathsf{lab}'.\mathsf{cd}_{\mathsf{kem}}$, then the challenger immediately terminates the game and declares the adversary to have won. Under this rule, a win via a KEM-stage error can occur only if the flawed implementation produces the same confirmation code as a correct one.

If such a termination event happens, the challenger can directly output the pair of confirmation codes as a solution to the confirmation-code unpredictability (cUP) game for the KEM scheme. This allows us to construct an adversary $\mathcal{B}_2$ against the KEM's cUP security that perfectly simulates Game 3; the occurrence of the abort event in the simulation implies that $\mathcal{B}_2$ wins its cUP game. Therefore, the difference in the adversary's advantage between the two games is bounded by the cUP advantage of the KEM:

$$\mathsf{Adv}^{G_3}_{\mathcal{A}} \leq \mathsf{Adv}^{G_4}_{\mathcal{A}} + \mathsf{Adv}^{\text{cUP-KEM}}_{\mathsf{KEM_C},\mathcal{B}_2}.$$

Game 5. (Exclude server signature verification errors) In this game, the challenger handles the case where the guessed user U has a server signature verification error. The game proceeds as in Game 4, but with an additional abort rule: if there exist two sessions $\mathsf{lab}, \mathsf{lab}'$ that reach the stage $3/4$ with $\mathsf{lab.sid}_i = \mathsf{lab}'.\mathsf{sid}_i \neq \bot$, $\mathsf{SIG.error} \in \mathsf{lab.ERROR}$, and $\mathsf{lab.cd}_{\mathsf{SCV}} = \mathsf{lab}'.\mathsf{cd}_{\mathsf{SCV}}$, then the challenger immediately terminates the game and declares the adversary to have won. Under this rule, a win via a server signature verification flaw can occur only if the flawed implementation produces the same confirmation code as a correct one.

If such a termination event happens, the challenger can directly output the pair of confirmation codes $(\mathsf{cd}_{\mathsf{SCV}}, \mathsf{cd}'_{\mathsf{SCV}})$ as a solution to the confirmation-code unpredictability (cUP) game for the signature scheme. This allows us to construct an adversary $\mathcal{B}_3$ against the signature scheme's cUP security that perfectly simulates Game 4; the occurrence of the abort event in the simulation implies that $\mathcal{B}_3$ wins its cUP game. Thus:

$$\mathsf{Adv}^{G_4}_{\mathcal{A}} \leq \mathsf{Adv}^{G_5}_{\mathcal{A}} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_3}.$$

Game 6. (Exclude server HMAC verification errors) In this game, the challenger handles the case where the guessed user U has a server HMAC verification error. The game proceeds as in Game 5, but with an additional abort rule: if there exist two sessions $\mathsf{lab}, \mathsf{lab}'$ that reach the stage $3/4$ with $\mathsf{lab.sid}_i = \mathsf{lab}'.\mathsf{sid}_i \neq \bot$, $\mathsf{HMAC.error} \in \mathsf{lab.ERROR}$, and $\mathsf{lab.cd}_{\mathsf{hmac,S}} = \mathsf{lab}'.\mathsf{cd}_{\mathsf{hmac,S}}$, then the challenger immediately terminates the game and declares the adversary to have won. Under this rule, a win via a server HMAC verification flaw can occur only if the flawed implementation produces the same confirmation code as a correct one.

If such a termination event happens, the challenger can directly output the pair of confirmation codes $(\mathsf{cd}_{\mathsf{hmac,S}}, \mathsf{cd}'_{\mathsf{hmac,S}})$ as a solution to the confirmation-code unpredictability (cUP) game for the HMAC function. This allows us to construct an adversary $\mathcal{B}_4$ against HMAC cUP security that perfectly simulates Game 5; the occurrence of the abort event in the simulation implies that $\mathcal{B}_4$ wins its cUP game. Thus:

$$\mathsf{Adv}^{G_5}_{\mathcal{A}} \leq \mathsf{Adv}^{G_6}_{\mathcal{A}} + \mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_4}.$$

Game 7. (Exclude client signature verification errors) In this game, the challenger handles the case where the guessed user U has a client signature verification error. The game proceeds as in Game 6, but with an additional abort rule: if there exist two sessions $\mathsf{lab}, \mathsf{lab}'$ that reach the stage $5/6$ with $\mathsf{lab.sid}_i = \mathsf{lab}'.\mathsf{sid}_i \neq \bot$, $\mathsf{SIG.error} \in \mathsf{lab.ERROR}$, $\mathsf{lab.cd}_{\mathsf{CCV}} = \mathsf{lab}'.\mathsf{cd}_{\mathsf{CCV}}$, and $\mathsf{lab.role} = \mathsf{responder}$, then the challenger immediately terminates the game and declares the adversary to have won. Under this rule, a win via a client signature verification flaw can occur only if the flawed implementation produces the same confirmation code.

If such a termination event happens, the challenger can directly output the pair of confirmation codes $(\mathsf{cd}_{\mathsf{CCV}}, \mathsf{cd}'_{\mathsf{CCV}})$

as a solution to the confirmation-code unpredictability (cUP) game for the signature scheme. This allows us to construct an adversary $\mathcal{B}_5$ against the signature scheme's cUP security that perfectly simulates Game 6; the occurrence of the abort event in the simulation implies that $\mathcal{B}_5$ wins its cUP game. Thus:

$$\mathsf{Adv}^{G_6}_{\mathcal{A}} \leq \mathsf{Adv}^{G_7}_{\mathcal{A}} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_5}.$$

Game 8. (Exclude client HMAC verification errors) In this game, the challenger handles the case where the guessed user U has a client HMAC verification error. The game proceeds as in Game 7, but with an additional abort rule: if there exist two sessions $\mathsf{lab}, \mathsf{lab}'$ that reach the stage 5/6 with $\mathsf{lab.sid}_i = \mathsf{lab}'.\mathsf{sid}_i \neq \perp$, $\mathsf{HMAC.error} \in \mathsf{lab.ERROR}$, $\mathsf{lab.cd_{hmac,C}} = \mathsf{lab}'.\mathsf{cd_{hmac,C}}$, and $\mathsf{lab.role} = \mathsf{responder}$, then the challenger immediately terminates the game and declares the adversary to have won. Under this rule, a win via a client HMAC verification flaw can occur only if the flawed implementation produces the same confirmation code.

If such a termination event happens, the challenger can directly output the pair of confirmation codes $(\mathsf{cd_{hmac,C}}, \mathsf{cd'_{hmac,C}})$ as a solution to the confirmation-code unpredictability (cUP) game for the HMAC function. This allows us to construct an adversary $\mathcal{B}_6$ against HMAC's cUP security that perfectly simulates Game 7; the occurrence of the abort event in the simulation implies that $\mathcal{B}_6$ wins its cUP game. Thus:

$$\mathsf{Adv}^{G_7}_{\mathcal{A}} \leq \mathsf{Adv}^{G_8}_{\mathcal{A}} + \mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_6}.$$

At this point, we have eliminated all scenarios in which an implementation flaw coexists with identical confirmation codes. Consequently, in the current game, whenever a flaw is present, the two parties cannot generate the same confirmation code. Moreover, because hash collisions have been removed, the inputs to the key-derivation function—now differing in at least the confirmation codes—are distinct, which guarantees that the derived session keys also differ.

$$\mathsf{Adv}^{G_8}_{\mathcal{A}} = 0.$$

In summary, we obtain the final bound on the adversary's advantage:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{MCIE}}_{\mathsf{PQCD-TLS},\mathcal{A}} \leq\ & \mathsf{Adv}^{\mathrm{COLL}}_{\mathsf{H},\mathcal{B}_1} + n_s \cdot \delta_e \\ & + n_e \cdot \begin{pmatrix} \mathsf{Adv}^{\text{cUP-KEM}}_{\mathsf{KEM_C},\mathcal{B}_2} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_3} \\ + \mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_4} + \mathsf{Adv}^{\text{cUP-Sig}}_{\mathsf{SIG_C},\mathcal{B}_5} \\ + \mathsf{Adv}^{\text{cUP-MAC}}_{\mathsf{HMAC_C},\mathcal{B}_6} \end{pmatrix}.\end{aligned}$$

□

We now explain the practical meaning of Theorem 5.3 for implementers. The purpose of the protocol is to ensure that, when implementation errors are present, they are reliably exposed as key mismatches and handshake failures during testing, so that faulty implementations are not deployed in production. Concretely, suppose an implementer has completed a candidate implementation that may contain implementation errors, and the confirmation codes are extracted according to Table 1. Then, in a single interoperability test, the maximum probability that the two parties still derive the same shared key is bounded by 0.08, corresponding to the worst-case configuration DAWN_C-α-512+ML-DSA_C-44+HMAC_C-SHA3-256. Thus, an implementation error leads to inconsistent shared keys with high probability. Moreover, if the implementer repeats the interoperability test independently, e.g., for 50 runs, then the probability that all 50 executions still derive matching shared keys is at most $0.08^{50} \approx 2^{-182}$. Therefore, Theorem 5.3 provides not only a theoretical guarantee for exposing implementation errors, but also a practical guideline for how confirmation codes should be extracted and used in real-world protocol implementations.

E Experimental Description and Results.

E.1 Implementation Platform

Our experimental platform consists of a desktop computer equipped with an Intel Core i5-14600KF processor (14 cores, 20 threads), 32 GB of DDR4 memory operating at 2666 MHz, and a 1 TB NVMe SSD (ZhiTai TiPlus7100). The system runs Microsoft Windows 11 Professional (64-bit, version 10.0.26100).

The post-quantum cryptographic algorithms are based on two open-source implementations: the Open Quantum Safe (`liboqs`) library and the `PQClean` project, which supplies verified, side channel protected code derived from NIST's PQC standardization submissions. Classical algorithms are taken directly from the Go standard library's `crypto` package. All C-based post-quantum implementations are integrated into a unified Go testing framework using cgo (Go 1.25.5, `GOARCH=amd64`, `GOOS=windows`). This approach combines the performance of mature, optimized C libraries with the convenience and tooling of the Go ecosystem for experiment orchestration.

E.2 Implementation Error Injection

To further evaluate the error-detection capability of confirmation-code binding, we conduct five representative fault-injection experiments covering the main verification steps considered in our model. Specifically, we inject: (1) signature verification bypass, (2) signature parameter mismatch, (3) bypass or incorrect execution of the FO re-encryption check during KEM decapsulation, (4) omission of the Finished-message HMAC verification, and (5) malformed CertificateVerify inputs caused by incorrect transcript construction or parsing. These fault classes correspond to

the verification errors considered in our formal analysis. For each injected fault, we evaluate three execution modes: a correct implementation, a faulty implementation without confirmation-code binding, and the same faulty implementation with confirmation-code binding enabled. The first mode provides the expected reference behavior. The second mode illustrates whether the injected fault can remain silent in the baseline handshake, while the third evaluates whether confirmation-code binding converts the same fault into an observable protocol failure.

The experiments show that several injected faults can remain functionally silent when confirmation-code binding is disabled. In particular, if a faulty verification routine incorrectly accepts or omits a security-critical check, the handshake may still proceed because the original protocol flow does not necessarily depend on the internal verification computation itself. This reproduces the type of implementation failure targeted by our design: security-critical verification logic may be faulty while basic interoperability remains unaffected. When confirmation-code binding is enabled, the faulty execution generally produces a confirmation code different from that generated by the correctly implemented peer. Because the corresponding confirmation code is incorporated into the stage-specific key derivation, this mismatch propagates to subsequent stage secrets or traffic secrets. As a result, the two peers eventually observe a protocol-level failure, including Finished verification failure, AEAD decryption failure, or unsuccessful application-data exchange, depending on the location of the injected fault.

These experiments are intended to demonstrate the practical role of confirmation codes during implementation testing rather than to claim coverage of arbitrary software faults. The mechanism detects faults that affect the verification computations captured by our confirmation-code constructions and formal error model. Within this scope, the results show that confirmation-code binding can turn otherwise silent verification faults into observable interoperability failures, making it suitable for development, continuous-integration, and cross-implementation testing during post-quantum TLS deployment.