

Kopis: A KEM for Obfuscation

Andrea Basso 

IBM Research Europe – Zurich

Andrea.Basso@ibm.com

Michael Rosenberg 

research@mrosenberg.pub

Version 1.0, Sep. 29, 2026

Abstract

Password-authenticated key exchange (PAKE) and obfuscated key exchange (OKEX) are widely used protocols, appearing in passport access control, Tor’s censorship evasion, and more. As quantum threats grow nearer, there have been an increasing number of proposals for post-quantum PAKE and OKEX. All such protocols are similar in that they build on a KEM, obfuscating public keys and/or ciphertexts sent over the wire, e.g., by adding a random mask or applying an ideal cipher. Many propose instantiation with ML-KEM, a NIST-standardized lattice-based KEM.

Unfortunately, ML-KEM is a poor fit for these obfuscating protocols. The algebraic structure of ML-KEM public keys and ciphertexts means that obfuscating these values requires custom procedures for hash-to-vector, matrix expansion, serialization/deserialization, and randomized ciphertext decompression. In order to be broadly useful, these procedures also must be constant-time to prevent side channels and must be standardized to permit interoperability. No such set of procedures exists today.

We observe that the barriers to instantiating PAKE and OKEX disappear if some of the KEM’s algebraic structure is removed, i.e., if ML-KEM is replaced with a KEM whose public keys and ciphertexts appear to be *uniform as bytestrings*.

In this work, we specify **Kopis**, a Module Learning-with-Rounding (MLWR) KEM with public keys and ciphertexts that are uniform as bytestrings. **Kopis** is intended to be a drop-in replacement for ML-KEM, in size, speed, and security. **Kopis** is nearly identical to **Saber**, a NIST PQC finalist, and thus inherits its years of cryptanalysis.

We demonstrate that **Kopis** bears the security properties needed for use in various PAKE and OKEX schemes and provide a formally verified Rust implementation complete with portable, AVX2, and NEON backends, and a non-formally-verified C implementation for Cortex-M4. We perform benchmarks and find that **Kopis** is comparable to and often outperforms the fastest known ML-KEM implementations.

Keywords: key agreement, password-based cryptography, post-quantum cryptography

1 Introduction

The key encapsulation mechanism (KEM) is the most prominent building block of post-quantum encryption. It has been deployed in file encryption protocols [Val25], encrypted messaging protocols [CS25, App24], TLS [KKWS26], DNSSEC [NW26], SSH [Ope25], and many more widely used protocols. There is one class of protocols, however, where KEM-based implementations are still not standardized and face implementation difficulties. These are protocols that require *obfuscation* of the key exchange material, i.e., which require that some algebraic operations be performed on a public key or ciphertext. We highlight two classes of such protocols, OKEX and PAKE, which demonstrate the difficulties in instantiation, and propose Kopsis as a solution.

OKEX. Obfuscated key exchange (OKEX) [GSV24] is a key exchange protocol which is intended to “look like nothing” to a passive observer, or, more properly stated, to look like a uniform sequence of bytes of random length. Use cases for OKEX protocols include protocol metadata hiding and censorship circumvention, such as in Tor’s *obfs4* protocol [The19].

The post-quantum OKEX protocols *pq-obfs* and *Drivel* [GSV24, GRSV25] use KEMs as building blocks and require that transmitted KEM public keys and ciphertexts be pseudorandom bytestrings. To handle this, the authors of [GSV24] define an abstraction, *obfuscated KEM* (OKEM), which is a KEM plus encoding and decoding algorithms for public keys and ciphertexts, mapping them to/from pseudorandom bytestrings. They further propose a concrete encoding mechanism for ML-KEM, called *Kameleon*, and define the OKEM ML-Kameleon by combining the two.

PAKE. Password-authenticated key exchange (PAKE) protocols permit parties with mutual knowledge of a low-entropy secret (the password) to interact over an insecure channel to derive a high-entropy secret (the session key). A secure PAKE guarantees that (1) any passive adversary is unable to derive the session key, and (2) any active adversary has at most one attempt per session at guessing the password. PAKE protocols are widespread, appearing in IoT authentication, passport access controls, and secure backup systems.

The post-quantum PAKE protocols *NolC*, *Tempo*, *EKE-PRF*, *CHIC*, *CAKE*, and *OCAKE* [BCP⁺23, ABJS24, JRX25, ABJ25a, ABJ25b] all use KEMs as building blocks and all require some algebraic manipulation of the public key, and sometimes also the ciphertext. The specific manipulations vary from PAKE to PAKE, but the two broad classes involve either (1) defining a keyed permutation family on the space of public keys/ciphertexts, or (2) defining an addition operation over the space of public keys/ciphertexts, as well as a hash function to those spaces.

Difficulties instantiating OKEX and PAKE. There are implementation and specification difficulties with existing post-quantum OKEX and PAKE protocols. OKEX protocols rely on ML-Kameleon. This is a functional choice, but the ML-Kameleon proposed standard [GSV26] is still in its early stages, has no constant-time implementation (aside from the one built for this work), and adds

meaningful overhead to both computation time and ciphertext size.¹ PAKE protocols either rely on ML-Kemleon or else must perform parsing and arithmetic that are specialized to the underlying KEM and define tricky constant-time hash functions which output to the public key space.² Further, due to the nature of the public key obfuscation, some PAKEs require that encapsulation be constant-time with respect to the public key, a property that ML-KEM does not have.³

There is thus a gap between the theory and practice of OKEX and PAKE protocols. As quantum timelines shorten, it is important that this gap be filled.

Insight. We observe that all the above difficulties are resolved by simply using a KEM whose public keys and ciphertexts are pseudorandom bytestrings, and whose encapsulation algorithm is constant-time with respect to the public key. Indeed, this is sufficient for OKEX, since such a KEM would be a *natural OKEM*, meaning all encoding algorithms are just the identity function. And it is sufficient for PAKE because addition in public key/ciphertext space is simply XOR, hashing to public key/ciphertext space is just ordinary hashing, and keyed permutation families can be trivially instantiated using a wide-block cipher, such as hctr2 [CHB21].

Our Contribution. We present and propose to standardize *Kopis*, a KEM whose public keys and ciphertexts are pseudorandom bytestrings, and whose encapsulation algorithm (and in fact all algorithms) is constant-time with respect to the public key (and all other inputs). We give an overview of *Kopis*, present a specification, and argue for the security of *Kopis*, relying on the fact that it is intentionally nearly identical to *Saber*, a NIST PQC finalist [DKRV18]. We further argue that it has all the necessary security properties to instantiate PAKE and OKEX protocols. We believe *Kopis* also is relevant for obfuscation use cases outside of PAKE and OKEX, such as oblivious transfer [MR19], private set intersection [RT21], steganography [KJGR21], and anamorphic encryption [PPY22].

We implement *Kopis* in Rust with three backends (portable, AVX2, NEON), and formally verify the correctness of all of them using the Lean theorem prover [dMKA⁺15]. We also implement *Kopis* in C for the ARM Cortex-M4 architecture, but do not formally verify it. We conduct benchmarks and find that *Kopis* is competitive with, and even outperforms, the fastest known ML-KEM libraries across all measured CPU architectures and feature sets. We conclude that *Kopis* is a viable drop-in replacement for ML-KEM for protocols which need obfuscation properties.

We include the full specification in [Appendix A](#). We also include a description in [Appendix B](#) of our methodology for using code generated by a large language

¹Concretely, encoding and decoding a ML-KEM-768 public key takes up to $4.6\times$ as long as encapsulation alone ([Section 4](#)), and ML-Kemleon-768 ciphertexts are 41% larger than ML-KEM's.

²In *Tempo*, the runtime of this blinding procedure alone is up to $5.6\times$ that of encapsulation ([Section 4](#)).

³Making this constant-time incurs a slowdown of $7.9\times$ on matrix expansion, and $3.3\times$ when doing key generation, encapsulation, and decapsulation in sequence [[ABJ25b](#), Table 1].

model (LLM) while still maintaining high-assurance guarantees.

2 Preliminaries

We use $:=$ to denote deterministic assignment, $\leftarrow$ to denote assignment where the right-hand side is a non-deterministic algorithm, and $\leftarrow \$$ to denote uniformly sampling from a set. We write $P(x, y; r)$ to denote running a procedure P with inputs x, y and random coins r . When “; r ” is omitted, it is assumed that r is uniformly sampled. We let λ denote our security parameter. We let $\mathbb{B}^n$ denote the space of bytestrings of length n . We denote vectors and matrices with bold font.

We define $\lceil x \rceil$ as the rounding of $x \in \mathbb{Q}$ to the nearest integer. When used with ring elements or vectors thereof, we apply rounding coefficient-wise, using the integer representative in $[0, q)$ if it is an integer modulo q . We write $\lceil \mathbf{x} \rceil_{q \rightarrow p}$ as shorthand for $\lceil (2^p/2^q)\mathbf{x} \rceil$.

We make use of the Module-Learning-with-Rounding (MLWR) problem, specifically over $R_n = R/(2^n)R$ where R is the quotient ring $\mathbb{Z}[X]/(X^{256} + 1)$. An $\text{MLWR}_{m, \ell, \mu, 2^q, 2^p}$ sample is of the form

$$(\mathbf{A}, \mathbf{b} = \lceil \mathbf{A}\mathbf{s} \rceil_{q \rightarrow p}) \in R_q^{m \times \ell} \times R_p^m$$

where the public part $\mathbf{A} \leftarrow \$ R_q^{m \times \ell}$, the secret $\mathbf{s} \leftarrow \beta_\mu(R^\ell)$, and β_μ is a centered binomial distribution with parameter μ (always even), taking on values in $[-\mu/2, \mu/2]$, and with probability mass function $p(x) = \frac{\mu!}{(\mu/2+x)!(\mu/2-x)!} 2^{-\mu}$. The $\text{MLWR}_{m, \ell, \mu, 2^q, 2^p}$ problem refers to deciding whether an input is an MLWR sample or a uniform value in $R_q^{m \times \ell} \times R_p^m$.

3 Our Construction

We construct **Kopis**, a Module-LWR-based KEM. As stated above, it is nearly identical to **Saber**, specifically its round 3 (and final) version [BMD⁺20]. It is constructed in the same style as ML-KEM, namely by building an IND-CPA-secure public key encryption (PKE) scheme and then applying the Fujisaki-Okamoto (FO) transform [FO99]. **Kopis** uses the same FO variant as ML-KEM, namely FO_m^χ [HHK17, §3.3] with additional hashing of the public key.

Since **Saber** strongly resembles ML-KEM, **Kopis** also strongly resembles ML-KEM. **Kopis**’s deviations from the last **Saber** specification are a mix of lessons learned during the standardization of Kyber into ML-KEM and performance optimizations. **Kopis** also has fewer implementation and usage gotchas than ML-KEM: its algorithms are all constant-time (whereas ML-KEM’s encapsulation and key generation running time depend on the seed in the public key), and its algorithms do not require public or secret key validation (whereas ML-KEM’s algorithms fail if pk has an unreduced coefficient or sk has a miscomputed hash).

We instantiate **Kopis** at the NIST security levels I, III, and V, yielding variants **Kopis-512**, **Kopis-768**, and **Kopis-1024**, respectively. Benchmarks can be

Table 1: Sizes (in bytes) of KEM public keys, ciphertexts, and secret keys.

KEM	Level I			Level III			Level V		
	pk	ct	sk	pk	ct	sk	pk	ct	sk
Kopis	672	736	32	992	1088	32	1312	1472	32
ML-KEM	800	768	1632	1184	1088	2400	1568	1568	3168
ML-Kameleon	800	1152	1632	1184	1536	2400	1568	1920	3168

found in [Section 4](#). Public key and ciphertext sizes are the same as Saber’s and slightly less than ML-KEM’s across the board. Ciphertexts are 23–36% smaller than those of ML-Kameleon ([Table 1](#)). As a result, unlike ML-KEM-512, Kopis-512 is able to fit a public key and ciphertext into a single UDP packet (1472 B, assuming an Ethernet packet size of 1500 B), a property that may be useful in key exchanges with such payloads [[HNS⁺21](#), [ADH⁺22](#), [GRSV25](#)].

Parameters. All variants use polynomials modulo $X^{256} + 1$, with public matrix entries in R_{13} and rounded public-key and ciphertext vectors in R_{10} . The module rank ℓ is the dimension of the $\ell \times \ell$ public matrix, and t is the number of bits per coefficient in the compressed ciphertext polynomial in R_t . All secrets are sampled from β_μ . We set $(\ell, t, \mu) = (2, 3, 10)$, $(3, 4, 8)$, and $(4, 6, 6)$ for Kopis-512, Kopis-768, and Kopis-1024, respectively. These match the parameters for LightSaber, Saber, and FireSaber [[BMD⁺20](#), §3].

A full specification of Kopis can be found in [Appendix A](#), and will appear in the C2SP standards body [[Val](#)].

3.1 New changes

We first highlight the most substantial differences between Saber and Kopis. These changes improve ergonomics and speed.

Secret key is 32 bytes. ML-KEM and Saber define secret keys as a 5-tuple, containing the secret vector, public key, public key hash, and implicit rejection secret. These values depend on each other. The specification does not define how to, for instance, treat a secret key whose public key hash does not match its public key. This brittleness has since been formalized in terms of the MAL-BIND-K-PK and MAL-BIND-K-CT security properties, which ML-KEM has been shown not to satisfy [[CDM24](#), [Sch24](#)]. As was pointed out in the same work, and recommended by others [[Val24](#)], a simple solution is to make secret keys 32 bytes, from which we can regenerate the rest of the values. We do this in Kopis. We do not, however, prevent implementers from internally caching the expanded secret key for optimization purposes.

All hashes are TurboSHAKE. We replace SHA-3 and SHAKE with TurboSHAKE [[BDH⁺23](#), [VWA⁺25](#)] everywhere, most notably for hashing public keys, hashing ciphertexts, and expanding matrix seeds. Specifically, TurboSHAKE128 replaces SHAKE128, and TurboSHAKE256 replaces SHA-3 and SHAKE256. The primary

difference in the primitives is that TurboSHAKE uses 12-round Keccak, whereas SHAKE and SHA-3 use 24-round Keccak. The change provides a meaningful speed boost, particularly in embedded systems, and improves masking efficiency due to the significant cost of masked implementations of Keccak [KDvB⁺22]. At the same time, the security margin remains comfortably high. From the TurboSHAKE authors [BDH⁺23]:

Changing the number of rounds in the underlying permutation from 24 in the SHA-3 standard functions to 12 in TurboSHAKE implies a drastic reduction in safety margin. Still, TurboSHAKE is a reduced-round version of Keccak and thereby directly benefits from all the cryptanalysis on the latter. There is ample evidence from third-party cryptanalysis that 12 rounds provides a comfortable security margin [62 citations], as well as from our own investigations [5 citations]...

At the time of this writing, the best collision attack applicable to TurboSHAKE or to any SHA-3 instance works only when the permutation is reduced to 6 rounds, and preimage attacks reach only 4 rounds. Hence our proposal has a safety margin of 6 out of 12 rounds for collision and (second) preimage resistance.

3.2 Fixes copied from ML-KEM

The remaining differences between Kopsis and Saber come from changes that were made to Kyber [ABD⁺21] before it was standardized. We list them below.

Matrix generation is pointwise. Rather than generate the public matrix as $\mathbf{A} = \text{XOF}(\rho)$, we now generate it as $\mathbf{A}[i][j] = \text{XOF}(\rho \| i \| j)$. This permits machines with SIMD support to parallelize matrix generation and permits memory-constrained devices to perform matrix multiplication element-by-element without having to keep the entire matrix in memory [KS25, BKS19]. We do the same for the generation of the secret vector $\mathbf{s}$.

Module dimension is hashed into seeds. The public matrix seed, secret seed, and implicit rejection secret are now derived from the hash of the secret key concatenated with the module dimension ℓ . This ensures that the same secret key does not produce related matrices when (possibly accidentally) used at multiple security levels.

Remove defensive hashing. The encapsulation procedures of Saber and Kyber sample 32 random bytes and immediately hash them. This was seen as a hedge against a poorly configured or backdoored random number generator. It was removed in ML-KEM because it was not believed to be reliable: “NIST believes that adding an extra hash to Kyber will not do much to mitigate the potential harms faced by the jeopardized system” [Dan23]. Thus, we exclude this hedge in Kopsis.

Remove ciphertext from final hash. Saber and Kyber include the ciphertext in the final shared secret hash. This inclusion was removed in ML-KEM, since its removal speeds up encapsulation and “simplifies proofs and leads to better bounds without requiring a new failure-bound analysis” [Sch23]. For the same

reasons, and because it does not affect any of the security properties we require, **Kopis** does the same.

3.3 Why not X?

We consider whether it is possible to use for our purposes (OKEX and PAKE) existing natural OKEMs, i.e., KEMs whose public keys and ciphertexts are pseudorandom bytestrings. Recall that our goal is to provide a drop-in replacement for ML-KEM in protocols that need their underlying KEM to have obfuscation properties.

Saber. Since **Kopis** is so similar to **Saber** [DKRV18], it is reasonable to ask why one wouldn't simply use **Saber** itself for the highlighted use cases.

Firstly, there is the issue of standardization. The final submission is not in agreement with its own reference implementation.⁴ Thus, there is no single source of truth on **Saber**'s behavior. Secondly, to the authors' knowledge, there are no actively maintained **Saber** implementations in any language. This is a meaningful barrier to usability. Thus, we propose **Kopis** as a modern version of **Saber**, with a clean specification incorporating minor but significant improvements and a maintained and optimized implementation.

Sable. **Sable** [MKKV21] is a proposed variant of **Saber**. While it maintains its overall structure, it uses a more aggressive choice of parameters, which we preferred to avoid in favor of conservativeness.

FrodoKEM. **FrodoKEM** [BCD⁺16] is a NIST Round 3 alternate candidate, based on unstructured Learning-with-Errors (LWE) with a power-of-two modulus. Since the end of the NIST competition, it has been adopted as a standard by the ISO and BSI [Int26, Ger26], and is now widely seen as a conservative choice of KEM.

FrodoKEM does not fit many use cases of ML-KEM, though, precisely because of its conservatism. Its public keys and ciphertexts are an order of magnitude larger than those of ML-KEM, and its runtimes are an order of magnitude longer. It was for these reasons that NIST chose not to select it as a finalist [MAA⁺20].

SMAUG-T. **SMAUG-T** [CCC⁺26] is one of the two winners of the Korean PQ KEM competition. It is an MLWE-MLWR hybrid scheme with a power-of-two modulus. Unlike **FrodoKEM**, its public keys and ciphertexts are comparable in size to those of ML-KEM.

SMAUG-T is an attractive candidate for our use cases, but there is little public literature on the scheme. Further, the authors of this work were only able to find a single public implementation of the final **SMAUG-T** standard, located on its official website [Cry]. The code package contains a C implementation of **SMAUG-T** in the directory, which we benchmarked and found to be 2–4× slower than the **libcrux** ML-KEM Rust crate using the portable backend.⁵

⁴The disagreements are entirely syntactic, amounting to differences in the ordering of hash inputs and return values. Nonetheless, these differences yield an entirely incompatible KEM.

⁵There is recent work [HPKS26] showing speedups of 50–64%. The code does not appear to be public, so we could not run experiments ourselves. Even taking the speedups at face

Given its lack of public scrutiny, lack of public implementations, and poor performance compared to ML-KEM, we reject SMAUG-T as a solution.

Round5. Round5 [BBF⁺19] is a lattice-based KEM that uses error-correcting codes to correct decryption failures. Although it offered enticing performance numbers, Round5 was not selected to move on to NIST PQC Round 3 due to the unclear security picture (a few minor attacks appeared in the literature, and some parameter sets were not covered by the security proof) and the fact that it did not offer a royalty-free license [MAA⁺20].

NTRU-HPS/NTRU-HRSS. The NTRU-HPS and NTRU-HRSS KEMs [CDH⁺20] are based on the NTRU assumption and appear to be natural OKEMs. Their public key and ciphertext sizes are comparable to those of ML-KEM, as are their encapsulation and decapsulation times, and they earned a place in NIST PQC Round 3 before being eliminated.

These KEMs are not drop-in replacements for ML-KEM when it comes to key generation performance. Key generation time is 10–20× ML-KEM’s. This makes these schemes poor choices for OKEX and PAKE, where every protocol execution requires the generation of an ephemeral keypair. Further, public key uniformity requires the additional assumption of *decisional NTRU* [LTV12], which does not follow from the security of the underlying scheme. This is because there is no known search-to-decision reduction for NTRU; moreover, the decisional NTRU assumption has no known worst-case-to-average-case reduction.

Classic McEliece. The Classic McEliece KEM is a code-based ISO standard [Int26] and NIST PQC finalist. Recently, it has been shown to be a natural OKEM [Him25].

However, for reasons similar to FrodoKEM’s, Classic McEliece is not a viable drop-in replacement for ML-KEM. While its ciphertexts are roughly 13% of the size of ML-KEM ciphertexts, the public keys are massive, ranging from 261 kB to 1.4 MB. This is problematic when used with any of the enumerated PAKE and OKEX protocols, since all require sending a fresh KEM public key at the beginning of each session.

3.4 Security of Kopsis

Kopsis inherits the security of Saber directly, as there are no deviations in Kopsis from Saber that meaningfully affect security.

Saber itself has received years of scrutiny from research teams around the world. In the NIST final round report which selected Kyber over Saber, it was written that the two “were very comparable to each other. NIST is confident in the security that each provides,” and “NIST finds the MLWE problem, which Kyber depends upon, marginally more convincing than the other assumptions like MLWR” [MAA⁺20]. This should still inspire confidence in the MLWR assumption. Outside of the NIST competition, Saber has also been:

value, encapsulation would still be roughly 2× slower than libcrux’s at level III.

Table 2: Classical security estimates for Kopsis and ML-KEM. Core-SVP denotes the lattice-estimator’s rough (SVP-only) model; full denotes its default MAT-ZOV model.

Parameter set	Core-SVP		Full	
	Kopsis	ML-KEM	Kopsis	ML-KEM
512	118.26	115.51	139.94	139.66
768	182.28	174.32	204.87	196.37
1024	247.62	241.76	267.30	262.34

- implemented in hardware and embedded systems [HLKH24, FBA⁺24, IARp23, CHX22, FVR⁺22, IAB⁺23, ZHLR22, DMG23, AMJ⁺23, IAR⁺21, BHK⁺21, BRvV22, PC20, ZZY⁺21, BDK⁺21, FSS20, SRB20, KBMSRV18, MBMTK⁺20, WGY20, SPOG19],
- attacked in hardware [BPL25, LCS⁺25, MKB⁺24, KCS⁺24, NDJ23, BNGD23, FKK⁺22, TUX⁺23, WND22, KDvB⁺22, BC22, CGMZ23, DHP⁺22, UXT⁺22, XIU⁺21, DB22, PP21, NDGJ21, RBRC22, SKL⁺20, RBRC20, RRCB20],
- formally verified [HMS22],
- directly cryptanalyzed [GM23, MJZ22, DHP⁺22, OW21, HV20, DRV20],
- and cryptanalyzed as a consequence of (M)LWR cryptanalysis [BBS26, WJLY25, NR23, WZW21, DVV19, DVV18, CZZ18, Mon18, BGM⁺16, BLR⁺18, AKPW13].

The security of Saber rests on the hardness of the Module Learning-with-Rounding (MLWR) problem [BPR12]. Asymptotically, LWR and its module variant are known to be at least as hard as (M)LWE [BPR12, AKPW13]. However, these reductions are not tight for the parameters used in practice, either requiring a superpolynomial modulus-to-rounding ratio or imposing bounds on the number of samples. Our security estimates therefore do not rely on the reductions but follow the standard heuristic methodology: we model each MLWR sample as an MLWE sample in which the rounding error is treated as a noise term drawn uniformly from an interval whose width depends on the rounding value, and estimate the cost of the best known attacks against the resulting MLWE instance. This approach follows both the standard approach for (M)LWR-based constructions and the security analysis of Saber [DKR⁺20]. Recently, a new analysis of LWR with power-of-two moduli [BBS26] shows an LWR-specific improved attack. However, the attack applies to regimes with a high number of samples (for a plain LWR instance with dimension $N = 512$ and Kopsis parameters, it requires more than 2^{56} samples), while Kopsis and Saber only reveal a few samples per instance (on the order of hundreds).

In Table 2, we report security estimates for Kopsis, obtained using the Core-SVP estimate and the full model of the lattice estimator by Albrecht et al. [APS15].

When compared to ML-KEM, **Kopis** has better security in both models and at all security levels.

Decryption failures. The decryption procedure in **Kopis** may fail when the rounding errors are above the decoding threshold, causing the underlying PKE to recover at least one message bit incorrectly [BMD⁺20, Theorem 6.2]. Since **Kopis** uses the same parameters as **Saber**, it also has the same decryption-failure probabilities: approximately 2^{-120} , 2^{-136} , and 2^{-165} for **Kopis-512**, **Kopis-768**, and **Kopis-1024**, respectively [BMD⁺20, §3, Theorem 6.2].

These decryption failures can be exploited in secret-key recovery attacks [DGJ⁺19, DRV20]. However, **Saber**’s analysis finds that decryption-failure attacks, both classical and quantum, are more costly than direct attacks on the MLWR problem (considering a limit of 2^{64} decryption queries per public key) [BMD⁺20, §6.3].

3.5 Security Properties

We show that **Kopis** achieves key security properties needed for general use, and for use in PAKE and OKEX. We include relevant snippets from the specification in Figure 1 and describe the relevant experiments in Figures 2 and 3.

We highlight that no theorem below is novel. All are restatements of theorems proved in prior published work (sometimes with slight adaptations to highlight the specific property).

3.5.1 Properties Necessary for PAKE and OKEX

IND-CCA. IND-CCA is the baseline requirement of every protocol we consider. **Kopis** is IND-CCA in the ROM and the QROM.

Proof. **Kopis**’s PKE is IND-CPA under $\text{MLWR}_{\ell+1, \ell, \mu, 2^{13}, 2^{10}}$. This is [DKRV18, Theorem 4], whose proof applies to **Kopis** unchanged. Our PKE differs from **Saber**’s only in how the public matrix and private vector are expanded from their seeds (pointwise versus reading sequentially), but both are modeled as random oracle outputs either way.

Kopis’s KEM is the Fujisaki-Okamoto variant FO_m^χ applied to that PKE. This carries an IND-CPA PKE to an IND-CCA KEM in the ROM [HHK17, Theorems 3.1 and 3.6] and in the QROM [GLX24, Theorems 5–7] [BHH⁺19, Theorem 5]. $\square$

UNI-CTXT⁺. Recall that pq-obfs and Drivel require the underlying OKEM to have ciphertext uniformity UNI-CTXT⁺ with $\mathcal{CT} = \mathbb{B}^n$ for some n , and CAKE requires only UNI-CTXT. **Kopis** is UNI-CTXT⁺ with $\mathcal{CT} = \mathbb{B}^{32(t+10\ell)} = \mathbb{B}^{\text{CT_SIZE}}$.

Proof. This is shown along the way in the proof of [DKRV18, Theorem 3]. We reproduce the relevant hops of the proof below.

Write $\mathbf{b}, \mathbf{b}', \mathbf{s}, \mathbf{s}', v', c_m$ for $\text{vec_b}, \text{vec_bprime}, \text{vec_s}, \text{vec_sprime}, \text{vprime}, c_m$, and let h_1 denote the all-4 constant added by **CompressToR10** and **CompressToRt**.

```

fn SkToPk(sk: [u8; 32]) → [u8; PK_SIZE]:
  let (_, _, pk, _) = ExpandSecretKey(sk)
  return pk

fn PkeEncrypt(
  randomness: [u8; 32],
  pk: [u8; PK_SIZE],
  msg: [u8; 32]
) → [u8; CT_SIZE]:
  let vec_b = deserialize_vec(10, pk[..256*ℓ*10/8])
  let mat_seed = pk[256*ℓ*10/8..]
  let m = deserialize_elem(1, msg)

  let mat_A = GenMat(mat_seed)
  let vec_sprime = GenSecret(randomness)
  let vec_bprime = CompressToR10(mat_A * vec_sprime)
  let vprime = transpose(vec_b)
  * (vec_sprime as VecR10)
  let cm = CompressToRt(vprime - ((m as R10) << 9))

  let ct = serialize_vec(10, vec_bprime)
  || serialize_elem(t, cm)
  return ct

fn PkeDecrypt(
  sk: [u8; 32],
  ct: [u8; CT_SIZE]
) → [u8; 32]:
  let (vec_s, ...) = ExpandSecretKey(sk)
  let vec_bprime = deserialize_vec(
    10,
    ct[..256*ℓ*10/8]
  )
  let cm = deserialize_elem(t, ct[256*ℓ*10/8..])
  let v = transpose(vec_bprime) * (vec_s as VecR10)
  let cm10 = (cm as R10) << (10 - t)
  let mprime = DecodeMsg(v - cm10)
  return serialize_elem(1, mprime)

fn KemEncap(
  randomness: [u8; 32],
  pk: [u8; PK_SIZE]
) → ([u8; 32], [u8; CT_SIZE]):
  # Derive the output key and encryption randomness
  let pkh = TurboSHAKE256(pk, 32, DOMSEP_PKHASH)
  let b = TurboSHAKE256(randomness || pkh, 64, DOMSEP_F0)
  let (k, r) = (b[..32], b[32..])

  # Encrypt to pk. `randomness` is itself the message
  let ct = PkeEncrypt(r, pk, randomness)

  return (k, ct)

fn KemDecap(sk: [u8; 32], ct: [u8; CT_SIZE]) → [u8; 32]:
  let (_, z, pk, pkh) = ExpandSecretKey(sk)

  let randomness = PkeDecrypt(sk, ct)
  let b = TurboSHAKE256(randomness || pkh, 64, DOMSEP_F0)
  let (k, rprime) = (b[..32], b[32..])
  let cprime = PkeEncrypt(rprime, pk, randomness)

  if ct == cprime:
    return k
  else:
    return TurboSHAKE256(z || ct, 32, DOMSEP_NOREJECT)

fn CompressToR10(v: VecR13) → VecR10:
  let h1 = make_r13([4u13; 256])
  let h = make_vec13([h1; ℓ])
  let s = (v + h) >> 3
  return (s as VecR10)

fn CompressToRt(r: R10) → Rt:
  let h1 = make_r10([4u10; 256])
  let s = (r + h1) >> (10 - t)
  return (s as Rt)

```

Figure 1: Excerpts from the specification (Appendix A) relevant for security properties. The security parameters ℓ and t denote the public matrix dimensions and the base-2 logarithm of the modulus of the space of compressed ring elements, respectively.

We model the public matrix $\mathbf{A}$ as a uniformly random value (it is the output of a random oracle). For $u \in R_n$ and $0 \leq a < b \leq n$ let $u[a..b] \in R_{b-a}$ denote the coefficient-wise bit slice ($u \gg a$) as R_{b-a} . Thus, $[x]_{13 \rightarrow 10} = (x + h_1)[3..13]$ and $\text{CompressToRt}(r) = (r + h_1)[10 - t..10]$.

Note that serialization is a bijection, since every modulus is a power of two and every ring element has 256 coefficients, so an element of R_n occupies exactly $32n$ bytes. Thus, showing pseudorandomness of a serialization is equivalent to showing pseudorandomness of its inputs. We now show UNI-CTXT^+ with the game-hopping argument from [DKRV18].

Game 0. The UNI-CTXT^+ game with $b = 0$: the ciphertext is (the serialization

of $(\mathbf{b}', c_m)$ with $\mathbf{b}' = \lceil \mathbf{A}\mathbf{s}' \rceil_{13 \rightarrow 10}$ and

$$\begin{aligned} c_m &= \text{CompressToRt}(v' - ((m \text{ as } R_{10}) \ll 9)) \\ &= (\mathbf{b}^T(\mathbf{s}' \text{ as } R_{10}^\ell) - ((m \text{ as } R_{10}) \ll 9) + h_1)[10 - t..10] \end{aligned}$$

Game 1. Replace $\mathbf{b} = \lceil \mathbf{A}^T \mathbf{s} \rceil_{13 \rightarrow 10}$ with a uniform element of R_{10}^ℓ . The distinguishing advantage of this hop is bounded by $\text{MLWR}_{\ell, \ell, \mu, 2^{13}, 2^{10}}$. Equivalently, since $2^{10} \mid 2^{13}$: sample $\mathbf{b}_q \leftarrow \$ R_{13}^\ell$ and set $\mathbf{b} := (\mathbf{b}_q \text{ as } R_{10}^\ell)$. Reducing mod 2^{10} maps uniform to uniform, so this is equivalent.

Game 2. Rewrite c_m :

$$\begin{aligned} \text{Game 1: } c_m &= (\mathbf{b}^T(\mathbf{s}' \text{ as } R_{10}^\ell) - ((m \text{ as } R_{10}) \ll 9) + h_1)[10 - t..10] \\ \text{Game 2: } c_m &= w[7 - t..7] - ((m \text{ as } R_t) \ll (t - 1)), \quad w := (\mathbf{b}_q^T \mathbf{s}' + h_1)[3..13], \end{aligned}$$

where we treat h_1 as a member of R_{13} or R_{10} as necessary.

These values are identical. Write $u := \mathbf{b}_q^T \mathbf{s}' + h_1 \in R_{13}$. Observe $w = u[3..13]$ and $y := \mathbf{b}^T(\mathbf{s}' \text{ as } R_{10}^\ell) + h_1 = u[0..10]$. Then $w[7 - t..7] = u[10 - t..10] = y[10 - t..10]$. The subtraction of the appropriate shifts of m preserves equality on both sides.

Note that this rewrite only works when $1 \leq t \leq 7$, which is true for every parameter set of **Saber** (and thus **Kopis**).

Game 3. Replace $\mathbf{b}'$ and w with uniform elements of R_{10}^ℓ and R_{10} . Since $(\mathbf{A}; \mathbf{b}_q^T)$ is uniform on $R_{13}^{(\ell+1) \times \ell}$ and $(\mathbf{b}', w) = \lceil (\mathbf{A}; \mathbf{b}_q^T) \mathbf{s}' \rceil_{13 \rightarrow 10}$, the distinguishing advantage of this hop is bounded by $\text{MLWR}_{\ell+1, \ell, \mu, 2^{13}, 2^{10}}$.

Game 4. Replace c_m with a uniform element of R_t . This hop is perfectly indistinguishable: a bit slice of a uniform value is uniform, as is a fixed subtraction of $2^{t-1}m$ from a uniform value.

Game 5. Undo Game 1, so that pk is honestly generated. The ciphertext no longer depends on $\mathbf{b}_q$, so this changes only the distribution of pk . The advantage of the hop is again bounded by $\text{MLWR}_{\ell, \ell, \mu, 2^{13}, 2^{10}}$.

This is the UNI-CTXT^+ game with $b = 1$. □

UNI-PK. Public key uniformity UNI-PK is required by **pq-obfs** (with $\mathcal{PK} = \mathbb{B}^n$ for some n), **NoIC**, **Tempo**, **CHIC**, **CAKE**, and **OAKE**. **Kopis** is UNI-PK with $\mathcal{PK} = \mathbb{B}^{320\ell+32} = \mathbb{B}^{\text{PK_SIZE}}$.

Proof. As with UNI-CTXT^+ , this comes from the proof of [DKRV18, Theorem 3]. Specifically, it is shown by the first hop in the proof above. □

SPR\\$-CCA. Recall that **CAKE**, **OAKE**, **pq-obfs**, and **Drivel** require SPR\\$-CCA. **Kopis** is SPR\\$-CCA with $\mathcal{CT} = \mathbb{B}^{32(t+10\ell)} = \mathbb{B}^{\text{CT_SIZE}}$.

Proof. [MX23] discusses SPR-CCA, essentially SPR\$-CCA but parameterized by a sampler $\mathcal{S}$ that chooses ct , rather than ct being sampled uniformly. The authors show SPR-CCA for Kyber⁶ and state that their results extend to Saber, and hence to Kopsis⁷, since it uses the same variant of the Fujisaki-Okamoto transform. This implies SPR\$-CCA only if the sampler used is the uniform one. We check that using the uniform sampler in their proof preserves the proof's validity.

The authors of [MX23] combine two properties to prove SPR-CCA: SDS-IND—i.e., UNI-CTXT⁺ but using a sampler $\mathcal{S}$ instead of uniformly sampling c_1 —and *disjointness*—i.e., that

$$\Pr[c \in \{\text{Encap}(\text{pk}; r)\}_{r \in \mathcal{R}} : (\cdot, \text{pk}) \leftarrow \text{Keygen}(), c \leftarrow \mathcal{S}]$$

is negligible, where $\mathcal{R}$ is the space of randomness (in our case, $\mathcal{R} = \mathbb{B}^{32}$). Kopsis is statistically disjoint with respect to the uniform sampler, since $|\{\text{Encap}(\text{pk}; r)\}_{r \in \mathbb{B}^{32}}| \leq 2^{256} \ll 2^{5888}$, which is the size of the ciphertext space at the lowest security level. Further, we know Kopsis is UNI-CTXT⁺ from above. Thus, Kopsis achieves SPR-CCA with respect to the uniform sampler, i.e., SPR\$-CCA. $\square$

PNM. CAKE and OCAKE require pseudorandom non-malleability PNM [JRX25]. Kopsis is PNM with $\mathcal{CT} = \mathbb{B}^{\text{CT_SIZE}}$.

Proof. SPR\$-CCA implies PNM [JRX25, Remark 3.7]. $\square$

sPR2. CAKE and OCAKE require strong second pseudorandomness sPR2 [JRX25]. Kopsis is sPR2 with $\mathcal{CT} = \mathbb{B}^{\text{CT_SIZE}}$.

Proof. SPR\$-CCA and SSMT\$-CCA imply sPR2 [JRX25, Definition 2.5]. In turn, SSMT\$-CCA is proven in [Xag22] for KEMs with SDS-IND and an even weaker Fujisaki-Okamoto transform than the one used by Kopsis (not hashing pk). Since we have SDS-IND from the argument above, we conclude Kopsis is sPR2. $\square$

CR_q. CAKE and OCAKE require collision resistance CR_q for polynomial q [JRX25]. Kopsis is CR_q.⁸

⁶The authors claim this with $\mathcal{S}$ set to the uniform sampler in Kyber's ciphertext space, but this is not true. After performing matrix multiplication, Kyber compresses ciphertexts, which produces an easily observable bias [GSV24]. Thus, the SPR-CCA claim is true under the MLWE assumption, but only with respect to the sampler that samples uniform values and compresses them in the same way.

⁷More precisely, it is shown for a simpler Fujisaki-Okamoto transform, FO_m^L , which hashes neither the public key nor the ciphertext into the final key. It is then shown that this result can be lifted to the FO transform that Kyber v3 uses (hashes public key and ciphertext). The same argument shows that it can be lifted to the FO transform that ML-KEM uses (hashes public key but not ciphertext).

⁸As the authors note, it is necessary for CR that the KEM shared secret have length 3λ . This can be achieved by simply applying a KDF to the output of Kopsis.

<u>Exp UNI-PK_b</u> $(\cdot, \text{pk}_0) \leftarrow \text{Keygen}()$ $\text{pk}_1 \leftarrow \$ \mathcal{PK}$ $b' \leftarrow \mathcal{A}(\text{pk}_b)$ return $b = b'$	<u>Exp OW-PCA</u> $(\text{sk}, \text{pk}) \leftarrow \text{Keygen}()$ $(\cdot, c^*) \leftarrow \text{Encap}(\text{pk})$ $K \leftarrow \mathcal{A}^{\text{PCO}_\perp(\text{sk}, \cdot, \cdot)}(\text{pk}, c^*)$ return $K = \text{Decap}(\text{sk}, c^*)$	<u>Exp ANO-PCA_b</u> $(\text{sk}_0, \text{pk}_0) \leftarrow \text{Keygen}()$ $(\text{sk}_1, \text{pk}_1) \leftarrow \text{Keygen}()$ $(\cdot, c^*) \leftarrow \text{Encap}(\text{pk}_b)$ $b' \leftarrow \mathcal{A}^{\text{PCO}_{c^*}(\text{sk}_0, \cdot, \cdot)}(\text{pk}_0, \text{pk}_1, c^*)$ return $b = b'$
<u>Exp UNI-CTXT_b</u> $(\cdot, \text{pk}) \leftarrow \text{Keygen}()$ $(\cdot, c_0) \leftarrow \text{Encap}(\text{pk})$ $c_1 \leftarrow \$ \mathcal{CT}$ $b' \leftarrow \mathcal{A}(\text{pk}, c_b)$ return $b = b'$	<u>Exp SPR\$-CCA_b</u> $(\text{sk}, \text{pk}) \leftarrow \text{Keygen}()$ $(K_0^*, c_0^*) \leftarrow \text{Encap}(\text{pk})$ $(c_1^*, K_1^*) \leftarrow \$ \mathcal{CT} \times \mathcal{K}$ $b' \leftarrow \mathcal{A}^{\text{Dec}_{c_b^*}(\text{sk}, \cdot)}(\text{pk}, c_b^*, K_b^*)$ return $b = b'$	<u>Exp SSMT\$-CCA_b</u> $(\text{sk}, \text{pk}) \leftarrow \text{Keygen}()$ $(c^*, K_0^*) \leftarrow \$ \mathcal{CT} \times \mathcal{K}$ $K_1^* := \text{Decap}(\text{sk}, c^*)$ $b' \leftarrow \mathcal{A}^{\text{Dec}_{c^*}}(\text{pk}, c^*, K_b^*)$ return $b = b'$
<u>Exp C2PRI</u> $(\text{sk}, \text{pk}) \leftarrow \text{Keygen}()$ $(K^*, c^*) \leftarrow \text{Encap}(\text{pk})$ $c \leftarrow \mathcal{A}(\text{sk}, \text{pk}, K^*, c^*)$ return $\text{Decap}(\text{sk}, c) = K^*$ $\wedge c \neq c^*$	<u>Oracle Dec_{c*}(sk, c)</u> if $c = c^*$: return $\perp$ return $\text{Decap}(\text{sk}, c)$ <u>Oracle PCO_{c*}(sk, c, K)</u> if $c = c^*$: return $\perp$ return $K = \text{Decap}(\text{sk}, c)$	<u>Exp IND-CCA_b</u> $(\text{sk}, \text{pk}) \leftarrow \text{Keygen}()$ $(K_0^*, c^*) \leftarrow \text{Encap}(\text{pk})$ $K_1^* \leftarrow \$ \mathcal{K}$ $b' \leftarrow \mathcal{A}^{\text{Dec}_{c^*}(\text{sk}, \cdot)}(\text{pk}, c^*, K_b^*)$ return $b = b'$

Figure 2: Experiments for some standard KEM properties. We call the version of UNI-CTXT without the gray box just UNI-CTXT, and the version with the gray box UNI-CTXT⁺. $\mathcal{PK}$ and $\mathcal{CT}$ refer to the spaces of KEM public keys and ciphertxts, respectively.

Proof. CR_q for polynomial q is shown for Kyber v3 [JR25]. We reproduce the argument below, as it applies identically to Saber and Kopsis.

Every **Decap** call hashes in some value that binds to the specific keypair, thus making collisions as hard as finding hash collisions. More precisely, in the successful decapsulation case, the output of **Decap** is (a subslice of) TurboSHAKE256($\text{randomness} \parallel \text{pkh}, \dots$), thus it includes (the hash of) pk . In the unsuccessful decapsulation case, the output is TurboSHAKE256($z \parallel \text{ct}, \dots$), thus it includes z , which itself is a hash of sk . Modeling the two invocations of TurboSHAKE256 as two random oracles (since they have distinct domain separators and input lengths), we can upper-bound the adversary's CR_q success probability by the probability of finding a single random oracle collision for an output length of 2λ , i.e., $q_{\text{RO}}/2^\lambda$ where q_{RO} is the number of random oracle queries the adversary makes. $\square$

OW-PCA and ANO-PCA. NoIC and CHIC require that the KEM be OW-PCA and ANO-PCA. Kopsis is OW-PCA and ANO-PCA.

Proof. OW-PCA is implied by IND-CCA, and ANO-PCA is implied by SPR\$-CCA [Xag22].

Exp PNM _b	Exp sPR2 _b	Exp CR _q
$(\text{sk}, \text{pk}) \leftarrow \text{Keygen}()$	$(\text{sk}, \text{pk}) \leftarrow \text{Keygen}()$	for $i \in 1..q$:
$(K'_0, c_0) \leftarrow \text{Encap}(\text{pk})$	$(\cdot, c_0) \leftarrow \text{Encap}(\text{pk})$	$(\text{pk}_i, \text{sk}_i) \leftarrow \text{Keygen}()$
$(c_1, K'_1) \leftarrow \mathcal{CT} \times \mathcal{K}$	$c_1 \leftarrow \mathcal{CT}$	$c^* \leftarrow \mathcal{A}(\text{pk}_1, \dots, \text{pk}_q)$
$(\text{st}, c^*) \leftarrow \mathcal{A}(\text{pk}, c_b, K'_b)$	$K := \text{Decap}(\text{sk}, c_b)$	for $i \in 0..q$:
if $c^* = c_b$: abort	$b' \leftarrow \mathcal{A}(\text{pk}, c_b, K)$	$K_i \parallel \tau_i := \text{Decap}(\text{sk}_i, c^*)$
$K := \text{Decap}(\text{sk}, c^*)$	return $b = b'$	return $\exists_{i \neq j} \tau_i = \tau_j$
$b' \leftarrow \mathcal{A}(\text{st}, K)$		
return $b = b'$		

Figure 3: Experiments for KEM properties needed for CAKE, OCAKE, and EKE-PRF. CR requires that K have length 3λ , and lets K_i be the first λ bits of the output of Decap .

□

Constant-time encapsulation. Because of the power-of-two moduli used in *Kopis*, public matrix expansion is constant-time, and so encapsulation is constant-time in the public key. In fact, every algorithm is constant-time in all inputs.

Not proving Tempo. We opt not to prove the properties that would allow one to instantiate *Tempo* with *Kopis*. This is mainly due to the fact that *Tempo*'s proofs reference MLWE and ML-KEM directly. We believe that the arguments hold identically for *Kopis*, but it would take up much space to work through the details.

Instead, we argue that there is no need to instantiate *Tempo* with *Kopis*, because its predecessor, *NoIC*, can be instantiated with *Kopis*. Indeed, the original purpose of *Tempo* was to fix the fact that *NoIC* could not be instantiated with a KEM whose matrix expansion step was variable-time. This fix is not free: the reduction of *Tempo* has worse tightness than that of *NoIC*, due to its use of the *Oracle MLWE* assumption, which resembles MLWE but with adversarial choice of public matrix seed. The authors show that *Oracle MLWE* is equivalent to MLWE, but the reduction includes a factor equal to the number of oracle queries made by the adversary. Since *Kopis* does not have variable-time matrix expansion, *Tempo* and its tightness overhead are entirely unnecessary.

Finally, we note that the only practical difference in implementing *NoIC* over *Tempo* is producing 32 additional bytes in the hash-to-public key space function and XORing them with the public matrix seed, a minuscule overhead by any measure.

3.5.2 Additional Properties

We identify some additional properties of *Kopis* that improve ergonomics and allow deployment in more scenarios.

MAL-BIND. Recall that MAL-BIND-K-CT indicates that the shared secret is bound to the ciphertext, i.e., that it is not possible to arrive at the same shared secret given two distinct ciphertexts [CDM24]. MAL-BIND-K-PK is defined similarly, but with public keys. Because of its expanded secret key format, ML-KEM was shown to trivially not attain either of these properties. Kopsis achieves both of these properties.

Proof. As was stated in [CDM24, §7.2], a KEM whose secret keys are 32-byte seeds sidesteps the trivial binding attacks: the public-key hash is recomputed consistently and correctly on every secret key load, as is the implicit rejection secret. Since Kopsis uses a 32-byte seed as a secret key, it achieves MAL-BIND-K-CT and MAL-BIND-K-PK. $\square$

C2PRI. It was shown in [BCD⁺24] that ML-KEM can be combined with a Diffie-Hellman noninteractive key exchange to achieve a KEM with *hybrid* IND-CCA security, i.e., security under both classical and post-quantum assumptions. The combiner—called QSF in the paper, CG in the IETF draft standard [CBG26], and X-Wing when instantiated with ML-KEM—relies on the *ciphertext second preimage resistance* (C2PRI) property of the underlying KEM for its classical security. Kopsis is C2PRI, and is thus compatible with efficient combiners.

Proof. This property is shown in [BCD⁺24, Theorem 3] for ML-KEM, and in [CHH⁺25, Theorems 23–27] in the QROM and for KEMs with various FO transforms. In particular, [CHH⁺25, Theorem 24] shows it for any KEM with $\text{FO}_m^{\mathcal{X}}$ ⁹ and the domain-separated random oracle instantiation that Kopsis and ML-KEM use. $\square$

4 Implementation and Benchmarks

We implement Kopsis in Rust, with three computational backends: portable, AVX2, and NEON. For performance, we rely heavily on the number-theoretic transforms (NTTs) described in [CHK⁺21]. We transcribe our specification to Lean and use Aeneas [HP22] to extract the Rust code to Lean [dMKA⁺15]. We formally verify that all three backends match a Lean transcription of our specification.

Scope of verification. We briefly describe what our Lean proofs establish. For each of the three backends, we prove that the Rust code (extracted by Aeneas) is equivalent to a Lean transcription of the Kopsis specification (Appendix A). The audit surface is as follows: the Kopsis Lean specification (209 lines of code), a TurboSHAKE Lean specification (45 lines, using a pre-existing Keccak Lean formalization), and 18 theorems covering API correctness (most duplicates, to cover all three security levels). Our trust base covers dependencies (a

⁹Really, it shows it for $\text{U}_m^{\mathcal{X}}$ and a rigid PKE scheme, but notes that the T re-encryption transform makes a PKE scheme into a rigid PKE. Since our $\text{FO}_m^{\mathcal{X}} = \text{U}_m^{\mathcal{X}} \circ \text{T}$, the theorem follows.

TurboSHAKE crate and a constant-time operations crate) and intrinsics (SIMD instructions and some mathematical intrinsics). To ensure constant-timeness, we rely on taint testing and binary inspection.

Much of the development and verification was aided by a large language model (LLM). See [Appendix B](#) for more detail on our methodology for producing high-assurance guarantees while delegating much of the code generation to an LLM.

4.1 Rust Microbenchmarks

We benchmark our formally verified Rust implementation of `Kopis`¹⁰ and compare it against the current fastest known Rust implementations of ML-KEM: `aws-lc-rs` [Ser23], `libcrux` [Lab24] (formally verified as well), `selkie` [Con26], and `graviola` [BP24]. In all tables, we report the median measured runtime, after a 3-second warmup. The maximum observed standard error of the median was 0.84%. Portable and AVX2 benchmarks were performed on an Intel i9-12900H CPU, with hyperthreading and Turbo Boost disabled. NEON benchmarks were performed on an Apple M3 chip.

All KEM experiments were done with derandomized implementations.¹¹ All implementations but `aws-lc-rs` use *unpacked* public and secret keys, meaning they store the expanded public key, deserialized public key vector, and public key hash in the data structure. We include `aws-lc-rs` in benchmarks for completeness, but note that the comparison in encapsulation and decapsulation is unfair, since it is parsing and expanding serialized values on every call.

Portable. Benchmarks of portable backends can be found in [Table 3](#). We compare against `libcrux` and `aws-lc-rs`. We omit `graviola`, since it has no portable implementation, only AVX2 and NEON.

We ensured that none of the portable implementations used SIMD-accelerated hashing. In this regime, we see that `Kopis` outperforms the other implementations in all operations except key generation. This is a fundamental tradeoff in our implementation: in order to achieve fast encapsulation and decapsulation, we must convert the public matrix to its NTT representation. This step is not needed in ML-KEM, since the matrix is directly sampled in the NTT domain.

AVX2 and NEON. Benchmarks of AVX2 backends can be found in [Table 4](#), and those of NEON backends in [Table 5](#). We benchmark the libraries from the portable case, plus `graviola`, and minus `libcrux` in the NEON case. Times for levels I and V are omitted for `graviola`, since it only implements ML-KEM-768. In both tables, we see the same dynamic as in the portable case: `Kopis` is largely faster than the competition in encapsulation and decapsulation, but falls short in key generation.

¹⁰Code at <https://github.com/rozbb/kopis-rs>.

¹¹This required modifications to the public API of `aws-lc-rs`. Code at <https://github.com/rozbb/aws-lc-rs/tree/mlkem-derand>.

Table 3: Portable runtime (in μs) of **Kopis** and Rust ML-KEM implementations.

Library	Level I			Level III			Level V		
	K	E	D	K	E	D	K	E	D
kopis	12.4	6.47	11.6	22.0	8.37	15.2	35.1	11.4	19.6
aws-lc-rs	12.6	15.4	19.7	20.0	23.7	29.4	30.8	34.6	41.3
libcrux	14.2	8.32	13.7	24.4	11.9	18.8	38.8	17.1	25.4
selkie	15.5	11.1	17.9	28.1	14.6	23.1	40.6	17.3	28.9

Table 4: AVX2 runtime (in μs) of **Kopis** and Rust ML-KEM implementations. **graviola** only implements ML-KEM-768.

Library	Level I			Level III			Level V		
	K	E	D	K	E	D	K	E	D
kopis	5.13	2.79	5.66	9.30	3.53	7.45	13.6	4.58	9.44
aws-lc-rs	5.35	5.90	7.67	8.45	9.11	11.6	12.3	12.9	16.1
libcrux	10.5	4.30	6.67	13.9	4.58	8.08	18.6	5.83	11.2
selkie	6.77	5.22	7.90	12.2	5.59	9.95	17.8	6.83	12.7
graviola	—	—	—	10.7	7.25	9.54	—	—	—

4.2 Cortex-M4 Microbenchmarks

We build our **Kopis** ARM Cortex-M4 implementation in C as a modification of the last **Saber** implementation in the **pqm4** library [SK18], a popular KEM and signature benchmarking library for the Cortex-M4.¹² While the implementation is not formally verified, we do benefit from having a small diff from the **Saber** implementation (+344/−102). We also benefit from all the optimizations done to the **Saber** implementation. We compare **Kopis** against the most recent ML-KEM implementation and the last available **Saber** implementation before it was removed from the **pqm4** repository. All tests are done with respect to

¹²Code at <https://github.com/rozbb/pqm4/tree/kopis>.

Table 5: NEON runtime (in μs) of **Kopis** and Rust ML-KEM implementations. **graviola** only implements ML-KEM-768.

Library	Level I			Level III			Level V		
	K	E	D	K	E	D	K	E	D
kopis	5.15	2.62	4.82	9.40	3.48	6.62	16.1	4.80	8.98
aws-lc-rs	3.87	4.55	5.74	6.80	7.25	8.93	9.91	10.6	13.1
selkie	5.76	3.88	6.19	10.6	4.98	8.44	15.3	6.59	10.9
graviola	—	—	—	7.37	4.15	5.18	—	—	—

Table 6: Cortex-M4 runtime (in kyc) of KEM implementations in pqm4.

Library	Level I			Level III			Level V		
	K	E	D	K	E	D	K	E	D
kopis	234	263	316	417	456	529	667	714	811
saber	306	417	403	560	711	685	866	1046	1018
ml-kem	374	372	406	610	624	670	969	981	1041

Table 7: Cortex-M4 memory use (in kB) of KEM implementations in pqm4.

Library	Level I			Level III			Level V		
	K	E	D	K	E	D	K	E	D
kopis	5.65	6.19	6.41	6.68	7.22	7.44	7.70	8.24	8.46
saber	5.60	6.29	6.30	6.63	7.31	7.32	7.66	8.34	8.34
ml-kem	4.37	5.44	5.41	5.40	6.47	6.45	6.44	7.50	7.48

the speed-optimized variant of the KEM (as opposed to the memory-optimized variant). In order to compare unpacked representations against unpacked representations, we deserialize the unpacked secret key for decapsulation (against the specification’s “SHOULD NOT” directive against this).

All benchmarks were performed on an STM32F4DISCOVERY board at 24 MHz to avoid flash memory wait states. The maximum observed standard error of the median was 0.6%. Since the algorithms of **Saber** and **Kopis** are constant-time, their runtimes did not vary from run to run.

As we can see from [Tables 6](#) and [7](#), **Kopis** outperforms **ML-KEM** and **Saber** on the Cortex-M4 across the board, notably including key generation. The speedup is entirely attributable to hashing. With NTTs, **Kopis**’s key generation would be fundamentally slower than **ML-KEM**’s if both used the same hash functions. But **ML-KEM** uses **SHA-3** and **SHAKE**, and so 75%, 74%, and 67% of the time spent in key generation, encapsulation, and decapsulation is just hashing, respectively. By using **TurboSHAKE**, **Kopis** brings this down to 59%, 52%, and 45%, respectively.

Stack overhead shows the opposite scenario, with **Kopis** losing across the board to **ML-KEM**, though never by more than 30%.

4.3 PAKE and OKEX Benchmarks

We now extend our analysis to the protocols which motivated **Kopis** in the first place: password-authenticated key exchange (PAKE) and obfuscated key exchange (OKEX). We find that **Kopis** provides meaningful speedups across the board, as well as simplified implementation requirements.

4.3.1 PAKE

We first compare the runtime of the **Tempo** PAKE [ABJS24], instantiated with ML-KEM, to **NolC** [BCP⁺23], instantiated with **Kopis** (recall that we do not show that **Tempo** can be instantiated with **Kopis**). Since the two are similar up to a 32-byte hash and XOR, this comparison is apt. Recall that the structure of both PAKEs is to hash the password (and a salt) to the public key space (for ML-KEM, vectors modulo q), and then add that value to the real public key (for ML-KEM, using vector addition). Importantly, since this hash function takes the password as input, it must be constant-time.

To avoid mixing and matching different ML-KEM libraries and hashing backends, we will limit ourselves to estimating the primary costs of the **Tempo** and **NolC** protocols, namely initial key generation, constant-time hash-to-vector ($\times 2$ since both parties must do it), encapsulation, and decapsulation. Of these, the only numbers not covered above are those from the constant-time hash-to-vector algorithm.

We implement a constant-time hash-to-vector algorithm described in **Tempo**, specifically the single-modular-division (SMD) algorithm (also called *Algorithm C1*), both for portable and AVX2 backends.¹³ We measure a minimum cost of 25.5 μ s for ML-KEM-768.¹⁴ Picking a good-at-everything ML-KEM implementation (*graviola*) and adding up the enumerated costs above, we get a total **Tempo** AVX2 runtime estimate of 78.5 μ s.

We do the same computation, instantiating **NolC** with **Kopis**. Our constant-time hash-to-vector for **Kopis-768** is just an XOF invocation producing 992 B, with no modular reduction necessary. We measure this to be 1.62 μ s. This makes the total **Kopis** AVX2 runtime estimate 23.5 μ s, or a 70% speedup compared to **Tempo**.

4.3.2 OKEX

We compare the runtime of **Drivel** [GRSV25], the hybrid-post-quantum obfuscated key exchange protocol, instantiated with ML-Kameleon versus **Kopis**. We estimate the primary costs of the online phase of **Drivel**: X25519 key generation, X25519 Diffie-Hellman computation, Elligator encoding, OKEM key generation, encapsulation, and decapsulation, and ordinary KEM key generation, encapsulation, and decapsulation.

In ML-Kameleon, key generation, encapsulation, and decapsulation all invoke Kameleon operations. We implement and benchmark Kameleon with AVX2, and report the numbers in Table 8.¹⁵ We also take benchmarks of X25519 key generation and Diffie-Hellman operations on AVX2 using the *curve25519-dalek* crate.

¹³Code at <https://github.com/rozbb/ct-kameleon/tree/94acc098f928719683fe3299367b20f96bdaa8b1>.

¹⁴In fact, nearly all of this time is spent just inside the **SHAKE-128** invocation. Each 12-bit coefficient requires 24 bytes of randomness to avoid bias when reducing. Thus, the total number of bytes squeezed from **SHAKE-128** for ML-KEM-768 is $24 \text{ B} \cdot 256 \cdot 3 = 18.4 \text{ kB}$.

¹⁵Code also at <https://github.com/rozbb/ct-kameleon/tree/94acc098f928719683fe3299367b20f96bdaa8b1>.

Table 8: AVX2 runtimes (in microseconds) for Kameleon encoding and decoding of public keys and ciphertexts.

Value	Level I		Level III		Level V	
	enc	dec	enc	dec	enc	dec
pk	4.50	9.26	6.76	14.2	9.25	18.3
ct	12.7	13.6	17.6	18.5	21.1	23.0

Key generation is 11.82 μ s, DH is 36.0 μ s, Ristretto map-to-curve is 2.97 μ s, and its inverse is 27.5 μ s.¹⁶ We estimate the KEM operations by summing *graviola* key generation, encapsulation, and decapsulation, and multiplying by two (since all operations are done by each side), yielding 55.0 μ s. Adding the elliptic curve overhead and *Kameleon* ciphertext encoding and decoding overheads, we find Drivel’s AVX2 runtime estimate to be 169.4 μ s. We also have an additional overhead of 448 B (41%) due to ciphertext encoding.

Doing the same with *Kopis* lets us skip *Kameleon* entirely, yielding an AVX2 runtime estimate of 118.9 μ s, a 30% speedup.

The runtime difference is starker with *pq-obfs*, since it must encode public keys as well. Instantiated with *ML-Kameleon-768*, we estimate the runtime (using AVX2) to be 112.0 μ s. Instantiated with *Kopis*, it is 40.6 μ s, a 64% speedup.

5 Future Work

We present some ideas not addressed in this work, which we believe would be interesting and useful as follow-up.

Kopis-512mini. While it is possible to fit a *Kopis-512* public key and ciphertext (1408 B total) into a single UDP packet, doing so leaves little room for protocol headers. Optimistically assuming an Ethernet MTU of 1500 B, the maximum UDP payload is 1472 B, leaving just 64 B for protocol headers. This is insufficient for, e.g., WireGuard [Don17], which has 116 B of headers in the initial message. Considering other plausible MTUs makes matters worse. IPv6 only requires an MTU of at least 1280 B, meaning there are at most $1232 - 116 = 1116$ bytes remaining for both a public key and ciphertext.

One path forward, as the authors of [HNS⁺21] note, is to define a variant of *Kopis-512* with a far higher decryption failure rate, ideally one which roughly matches the likelihood of UDP transmission failure, which is already the dominant source of key exchange failure. The noticeable decryption failure rate breaks IND-CCA-security, but this is fine, since only IND-CPA-security is needed for the ephemeral part of the key exchange. We estimate that increasing the

¹⁶A Curve25519 map-to-curve inverse function was not readily available in *curve25519-dalek*, so we instead use Ristretto, a prime-order coset representation of Curve25519, with little overhead compared to the base curve.

decryption failure rate to 2^{-30} allows us to free 64 B from the public key. This is enough for WireGuard at an MTU of 1500 B, but does not resolve the problem at lower MTUs. For these, we suspect an entirely different KEM is necessary.

Side-channel-secure and HW implementations. Saber is well-suited for side-channel resistant implementations, since the power-of-two moduli simplify masking operations and the conversion between different types of masking [BBD⁺25, Sec. 5]. Hardware implementations of Saber are also simplified, since no dedicated reduction is needed [RB20].

Kopis inherits these advantages and offers further opportunities for efficient implementations. Replacing SHAKE with TurboSHAKE halves the number of Keccak rounds, thus lowering the cost of masked hashing. Moreover, pointwise matrix generation enables parallel hashing evaluations in hardware. These features motivate exploring masked implementations of Kopis, as well as high-performance hardware implementations.

Standardizing PAKE on Kopis. The existing IETF draft standard for hybrid-post-quantum PAKE [VJW26] (built on NoI^C) currently only supports ML-Kemelon for the underlying OKEM. It may be helpful for, e.g., embedded device support, to extend the specification to offer Kopis as an alternate OKEM.

6 Rights Release

To the extent possible under law, the authors have waived all copyright and related or neighboring rights to this specification, as described in [CC0 1.0 Universal](#).

To the best of the authors’ knowledge, no patents or patent applications cover Kopis as described in this specification. Each author further grants to every person a perpetual, worldwide, no-charge, royalty-free, irrevocable license under any patent rights that author holds or later acquires, and that are necessarily infringed by implementing this specification, to make, have made, use, offer to sell, sell, import, and otherwise practice the subject matter described in this specification.

Acknowledgements

Many thanks to Mark Schultz-Wu, Shannon Veitch, and David Anderson for their helpful feedback on early drafts.

AI use designation. We designate this work as having *substantial AI contribution*. We use an LLM extensively for the implementation and formal verification of our primary result. Methodology details are documented extensively in [Appendix B](#).

References

- [ABD⁺21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber. algorithm Specifications And Supporting Documentation, version 3.02. August 2021. URL: <https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf>.
- [ABJ25a] Afonso Arriaga, Manuel Barbosa, and Stanislaw Jarecki. NoIC: PAKE from KEM without ideal ciphers. Cryptology ePrint Archive, Report 2025/231, 2025. URL: <https://eprint.iacr.org/2025/231>.
- [ABJ25b] Afonso Arriaga, Manuel Barbosa, and Stanislaw Jarecki. Tempo: ML-KEM to PAKE compiler resilient to timing attacks. Cryptology ePrint Archive, Report 2025/1399, 2025. URL: <https://eprint.iacr.org/2025/1399>.
- [ABJS24] Afonso Arriaga, Manuel Barbosa, Stanislaw Jarecki, and Marjan Skrobot. C’est Très CHIC: A compact password-authenticated key exchange from lattice-based KEM. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part V*, volume 15488 of *LNCS*, pages 3–33. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0935-2_1.
- [ADH⁺22] Yawning Angel, Benjamin Dowling, Andreas Hülsing, Peter Schwabe, and Fiona Johanna Weber. Post quantum noise. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 97–109. ACM Press, November 2022. doi:10.1145/3548606.3560577.
- [AKPW13] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited - new reduction, properties and applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 57–74. Springer, Berlin, Heidelberg, August 2013. doi:10.1007/978-3-642-40041-4_4.
- [AMJ⁺23] Aikata Aikata, Ahmet Can Mert, David Jacquemin, Amitabh Das, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy. A unified cryptoprocessor for lattice-based signature and key-exchange. *IEEE Transactions on Computers*, 72(6):1568–1580, 2023. doi:10.1109/TC.2022.3215064.
- [App24] Apple Security Engineering and Architecture. iMessage with PQ3: The new state of the art in quantum-secure messaging at scale - Apple Security Research, February 2024. URL: <https://security.apple.com/blog/imessage-pq3/>.

- [APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015. URL: <http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml>.
- [BBD⁺25] Andrea Basso, Joppe W. Bos, Jan-Pieter D’Anvers, Angshuman Karmakar, Jose Maria Bermudo Mera, Joost Renes, Sujoy Sinha Roy, Frederik Vercauteren, Peng Wang, Yuesu Wang, Shicong Zhang, and Chenxin Zhong. Using learning with rounding to instantiate post-quantum cryptographic algorithms. Cryptology ePrint Archive, Report 2025/1382, 2025. URL: <https://eprint.iacr.org/2025/1382>.
- [BBF⁺19] Hayo Baan, Sauvik Bhattacharya, Scott R. Fluhrer, Óscar García-Morchón, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*, pages 83–102. Springer, Cham, 2019. doi:10.1007/978-3-030-25510-7_5.
- [BBS26] Jules Baudrin, Rachelle Heim Boissier, and François-Xavier Standaert. On the concrete hardness of LWR with a power of two modulus. Cryptology ePrint Archive, Paper 2026/250, 2026. URL: <https://eprint.iacr.org/2026/250>.
- [BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *IACR TCHES*, 2022(4):553–588, 2022. doi:10.46586/tches.v2022.i4.553-588.
- [BCD⁺16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1006–1018. ACM Press, October 2016. doi:10.1145/2976749.2978425.
- [BCD⁺24] Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karoline Varner, and Bas Westerbaan. X-Wing. *CiC*, 1(1):21, 2024. doi:10.62056/a3qj89n4e.
- [BCP⁺23] Hugo Beguinet, Céline Chevalier, David Pointcheval, Thomas Ricosset, and Mélissa Rossi. GeT a CAKE: Generic transformations from key encapsulation mechanisms to password authenticated key exchanges. In Mehdi Tibouchi and Xiaofeng Wang,

- editors, *ACNS 2023, Part II*, volume 13906 of *LNCS*, pages 516–538. Springer, Cham, June 2023. doi:[10.1007/978-3-031-33491-7_19](https://doi.org/10.1007/978-3-031-33491-7_19).
- [BDH⁺23] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. TurboSHAKE. Cryptology ePrint Archive, Report 2023/342, 2023. URL: <https://eprint.iacr.org/2023/342>.
- [BDK⁺21] Michiel Van Beirendonck, Jan-Pieter D’anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of saber. *J. Emerg. Technol. Comput. Syst.*, 17(2), April 2021. doi:[10.1145/3429983](https://doi.org/10.1145/3429983).
- [BGM⁺16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 209–224. Springer, Berlin, Heidelberg, January 2016. doi:[10.1007/978-3-662-49096-9_9](https://doi.org/10.1007/978-3-662-49096-9_9).
- [BHH⁺19] Nina Bindel, Mike Hamburg, Kathrin Hövelmanns, Andreas Hülsing, and Edoardo Persichetti. Tighter proofs of CCA security in the quantum random oracle model. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 61–90. Springer, Cham, December 2019. doi:[10.1007/978-3-030-36033-7_3](https://doi.org/10.1007/978-3-030-36033-7_3).
- [BHK⁺21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, Nov. 2021. URL: <https://tches.iacr.org/index.php/TCHES/article/view/9295>, doi:[10.46586/tches.v2022.i1.221-244](https://doi.org/10.46586/tches.v2022.i1.221-244).
- [BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2019*, volume 11627, pages 209–228. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science. URL: https://link.springer.com/10.1007/978-3-030-23696-0_11, doi:[10.1007/978-3-030-23696-0_11](https://doi.org/10.1007/978-3-030-23696-0_11).
- [BLR⁺18] Shi Bai, Tancrede Lepoint, Adeline Roux-Langlois, Amin Sakzad, Damien Stehlé, and Ron Steinfeld. Improved security

- proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance. *Journal of Cryptology*, 31(2):610–640, April 2018. doi:[10.1007/s00145-017-9265-9](https://doi.org/10.1007/s00145-017-9265-9).
- [BMD⁺20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 3 Submission), 2020. URL: <https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf>.
- [BNGD23] Linus Backlund, Kalle Ngo, Joel Gärtner, and Elena Dubrova. Secret key recovery attack on masked and shuffled implementations of crystals-kyber and saber. In Jianying Zhou, Lejla Batina, Zengpeng Li, Jingqiang Lin, Eleonora Losiouk, Suryadipta Majumdar, Daisuke Mashima, Weizhi Meng, Stjepan Picek, Mohammad Ashiqur Rahman, Jun Shao, Masaki Shimaoka, Ezekiel Soremekun, Chunhua Su, Je Sen Teh, Aleksei Udovenko, Cong Wang, Leo Zhang, and Yury Zhauniarovich, editors, *Applied Cryptography and Network Security Workshops*, pages 159–177, Cham, 2023. Springer Nature Switzerland.
- [BP24] Joe Birr-Pixton. Graviola: High quality, fast and easy to build cryptography for Rust, September 2024. URL: <https://github.com/ctz/graviola>.
- [BPL25] Francesco Berti, Sasha Petri, and Itamar Levi. Too easy fault injection attacks on learning with rounding (LWR). Cryptology ePrint Archive, Report 2025/2246, 2025. URL: <https://eprint.iacr.org/2025/2246>.
- [BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737. Springer, Berlin, Heidelberg, April 2012. doi:[10.1007/978-3-642-29011-4_42](https://doi.org/10.1007/978-3-642-29011-4_42).
- [BRvV22] Joppe W. Bos, Joost Renes, and Christine van Vredendaal. Post-Quantum cryptography with contemporary Co-Processors: Beyond kronecker, Schönhage-Strassen & nussbaumer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3683–3697, Boston, MA, August 2022. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/bos>.
- [CBG26] Deire Connolly, Richard Barnes, and Paul Grubbs. Hybrid PQ/T Key Encapsulation Mechanisms. Internet Draft draft-irtf-cfrg-hybrid-kems-11, Internet Engineering Task Force, July

2026. Num Pages: 37. URL: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-hybrid-kems-11>.
- [CCC⁺26] Jung Hee Cheon, Hyeongmin Choe, Joongeun Choi, Dongyeon Hong, Jeongdae Hong, Chi-Gon Jung, Honggoo Kang, Minhyeok Kang, Taekyung Kim, Jeongbeen Ko, Janghyun Lee, Seonghyuck Lim, Aesun Park, Seunghwan Park, Jungjoo Seo, Hyoeun Seong, Junbum Shin, and MinJune Yi. SMAUG-T: the Key Exchange Algorithm based on Module-LWE and Module-LWR. February 2026.
 - [CDH⁺20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Technical report, National Institute of Standards and Technology, 2020. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>.
 - [CDM24] Cas Cremers, Alexander Dax, and Niklas Medinger. Keeping up with the KEMs: Stronger security notions for KEMs and automated analysis of KEM-based protocols. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 1046–1060. ACM Press, October 2024. doi:10.1145/3658644.3670283.
 - [CGMZ23] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. *IACR TCHES*, 2023(1):153–192, 2023. doi:10.46586/tches.v2023.i1.153-192.
 - [CHB21] Paul Crowley, Nathan Huckleberry, and Eric Biggers. Length-preserving encryption with HCTR2. Cryptology ePrint Archive, Report 2021/1441, 2021. URL: <https://eprint.iacr.org/2021/1441>.
 - [CHH⁺25] Deirdre Connolly, Kathrin Hövelmanns, Andreas Hülsing, Stavros Kousidis, and Matthias Meijers. Starfighters—on the general applicability of X-wing. Cryptology ePrint Archive, Report 2025/1397, 2025. URL: <https://eprint.iacr.org/2025/1397>.
 - [CHK⁺21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kanwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings. *IACR TCHES*, 2021(2):159–188, 2021. URL: <https://tches.iacr.org/index.php/TCHES/article/view/8791>, doi:10.46586/tches.v2021.i2.159-188.

- [CHX22] Elizabeth Carter, Pengzhou He, and Jiafeng Xie. High-performance polynomial multiplication hardware accelerators for KEM Saber and NTRU. Cryptology ePrint Archive, Report 2022/628, 2022. URL: <https://eprint.iacr.org/2022/628>.
- [Con26] Deirdre Connolly. mlkem-selkie, August 2026. URL: <https://github.com/selkie-cryptography/mlkem-selkie>.
- [Cry] Cryptolab. SMAUG-T & HAETAE - SMAUG-T. URL: <https://kqc.cryptolab.co.kr/smaug-t>.
- [CS25] Graeme Connell and Rolfe Schmidt. Signal Protocol and Post-Quantum Ratchets, October 2025. URL: <https://signal.org/blog/spqr/>.
- [CZZ18] Long Chen, Zhenfeng Zhang, and Zhenfei Zhang. On the hardness of the computational ring-LWR problem and its applications. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 435–464. Springer, Cham, December 2018. doi:10.1007/978-3-030-03326-2_15.
- [Dan23] Quynh H. Dang. Email in pqc-forum: Discussion about Kyber’s tweaked FO transform, may 2023. URL: <https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/WFRDl8DqYQ4/m/MRa500CvAAAJ>.
- [DB22] Jan-Pieter D’Anvers and Senne Batsleer. Multitarget decryption failure attacks and their application to Saber and Kyber. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *PKC 2022, Part I*, volume 13177 of *LNCS*, pages 3–33. Springer, Cham, March 2022. doi:10.1007/978-3-030-97121-2_1.
- [DGJ⁺19] Jan-Pieter D’Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 565–598. Springer, Cham, April 2019. doi:10.1007/978-3-030-17259-6_19.
- [DHP⁺22] Jan-Pieter D’Anvers, Daniel Heinz, Peter Pessl, Michiel Van Beirendonck, and Ingrid Verbauwhede. Higher-order masked ciphertext comparison for lattice-based cryptography. *IACR TCHES*, 2022(2):115–139, 2022. doi:10.46586/tches.v2022.i2.115-139.
- [DKR⁺20] Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report,

National Institute of Standards and Technology, 2020. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>.

- [DKRV18] 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, *AFRICACRYPT 18*, volume 10831 of *LNCS*, pages 282–305. Springer, Cham, May 2018. doi:10.1007/978-3-319-89339-6_16.
- [DMG23] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. High-speed hardware architectures and fpga benchmarking of crystals-kyber, ntru, and saber. *IEEE Transactions on Computers*, 72(2):306–320, 2023. doi:10.1109/TC.2022.3222954.
- [dMKA⁺15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.
- [Don17] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *NDSS 2017*. The Internet Society, February / March 2017. doi:10.14722/ndss.2017.23160.
- [DRV20] Jan-Pieter D’Anvers, Mélissa Rossi, and Fernando Virdia. (One) failure is not an option: Bootstrapping the search for failures in lattice-based encryption schemes. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 3–33. Springer, Cham, May 2020. doi:10.1007/978-3-030-45727-3_1.
- [DVV18] Jan-Pieter D’Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. On the impact of decryption failures on the security of LWE/LWR based schemes. Cryptology ePrint Archive, Report 2018/1089, 2018. URL: <https://eprint.iacr.org/2018/1089>.
- [DVV19] Jan-Pieter D’Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on ring/mod-LWE/LWR based schemes. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*, pages 103–115. Springer, Cham, 2019. doi:10.1007/978-3-030-25510-7_6.

- [FBA⁺24] Décio Luiz Gazzoni Filho, Guilherme Brandão, Gora Adj, Arwa Alblooshi, Isaac A. Canales-Martínez, Jorge Chávez-Saab, and Julio López. Pqc-amx: Accelerating saber and frodokem on the apple m1 and m3 socs. In *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*, pages 9–16, 2024. doi:[10.1109/ARITH61463.2024.00012](https://doi.org/10.1109/ARITH61463.2024.00012).
- [FKK⁺22] Michael Fahr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray A. Perlner, Arkady Yerukhimovich, and Daniel Apon. When frodo flips: End-to-end key recovery on FrodoKEM via rowhammer. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 979–993. ACM Press, November 2022. doi:[10.1145/3548606.3560673](https://doi.org/10.1145/3548606.3560673).
- [FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO’99*, volume 1666 of *LNCS*, pages 537–554. Springer, Berlin, Heidelberg, August 1999. doi:[10.1007/3-540-48405-1_34](https://doi.org/10.1007/3-540-48405-1_34).
- [FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280, Aug. 2020. URL: <https://tches.iacr.org/index.php/TCHES/article/view/8683>, doi:[10.13154/tches.v2020.i4.239-280](https://doi.org/10.13154/tches.v2020.i4.239-280).
- [FVR⁺22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR TCHES*, 2022(1):414–460, 2022. doi:[10.46586/tches.v2022.i1.414-460](https://doi.org/10.46586/tches.v2022.i1.414-460).
- [Ger26] German Federal Office for Information Security. Bsi tr-02102-1 — cryptographic mechanisms: Recommendations and key lengths, January 2026.
- [GLX24] Jiangxia Ge, Heming Liao, and Rui Xue. Measure-rewind-extract: Tighter proofs of one-way to hiding and CCA security in the quantum random oracle model. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part IV*, volume 15487 of *LNCS*, pages 3–34. Springer, Singapore, December 2024. doi:[10.1007/978-981-96-0894-2_1](https://doi.org/10.1007/978-981-96-0894-2_1).
- [GM23] Qian Guo and Erik Mårtensson. Do not bound to a single position: Near-optimal multi-positional mismatch attacks against Kyber and Saber. In Thomas Johansson and Daniel Smith-Tone,

- editors, *Post-Quantum Cryptography - 14th International Workshop, PQCrypto 2023*, pages 291–320. Springer, Cham, August 2023. doi:[10.1007/978-3-031-40003-2_11](https://doi.org/10.1007/978-3-031-40003-2_11).
- [GRSV25] Felix Günther, Michael Rosenberg, Douglas Stebila, and Shannon Veitch. Hybrid obfuscated key exchange and KEMs. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part III*, volume 16002 of *LNCS*, pages 575–609. Springer, Cham, August 2025. doi:[10.1007/978-3-032-01881-6_18](https://doi.org/10.1007/978-3-032-01881-6_18).
- [GSV24] Felix Günther, Douglas Stebila, and Shannon Veitch. Obfuscated key exchange. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 2385–2399. ACM Press, October 2024. doi:[10.1145/3658644.3690220](https://doi.org/10.1145/3658644.3690220).
- [GSV26] Felix Günther, Douglas Stebila, and Shannon Veitch. Kameleon Encodings. Internet Draft draft-irtf-cfrg-kameleon-02, Internet Engineering Task Force, July 2026. Num Pages: 15. URL: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-kameleon-02>.
- [HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Cham, November 2017. doi:[10.1007/978-3-319-70500-2_12](https://doi.org/10.1007/978-3-319-70500-2_12).
- [Him25] Marc Himmelberger. Implementing and Evaluating Quantum-Safe Fully Encrypted Protocols. June 2025. URL: <http://hdl.handle.net/20.500.11850/746588>, doi:[10.3929/ETHZ-B-000746588](https://doi.org/10.3929/ETHZ-B-000746588).
- [HLKH24] Muhammad Asfand Hafeez, Wai-Kong Lee, Angshuman Kar-makar, and Seong Oun Hwang. Efficient tmvp-based polynomial convolution on gpu for post-quantum cryptography targeting iot applications. *IEEE Internet of Things Journal*, 11(13):23428–23443, 2024. doi:[10.1109/JIOT.2024.3384507](https://doi.org/10.1109/JIOT.2024.3384507).
- [HMS22] Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Formal verification of Saber’s public-key encryption scheme in Easy-Crypt. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 622–653. Springer, Cham, August 2022. doi:[10.1007/978-3-031-15802-5_22](https://doi.org/10.1007/978-3-031-15802-5_22).
- [HNS⁺21] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Fiona Johanna Weber, and Philip R. Zimmermann. Post-quantum

- WireGuard. In *2021 IEEE Symposium on Security and Privacy*, pages 304–321. IEEE Computer Society Press, May 2021. doi:[10.1109/SP40001.2021.00030](https://doi.org/10.1109/SP40001.2021.00030).
- [HP22] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022. doi:[10.1145/3547647](https://doi.org/10.1145/3547647).
- [HPKS26] Yu-Lim Hyoung, Do-Yun Park, Hyun-Ji Kim, and Hwa-Jeong Seo. An optimized native rust implementation of the KpqC algorithms. Cryptology ePrint Archive, Paper 2026/1420, 2026. URL: <https://eprint.iacr.org/2026/1420>.
- [HV20] Loïs Huguenin-Dumittan and Serge Vaudenay. Classical misuse attacks on NIST round 2 PQC - the power of rank-based schemes. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 2020, Part I*, volume 12146 of *LNCS*, pages 208–227. Springer, Cham, October 2020. doi:[10.1007/978-3-030-57808-4_11](https://doi.org/10.1007/978-3-030-57808-4_11).
- [IAB+23] Malik Imran, Felipe Almeida, Andrea Basso, Sujoy Sinha Roy, and Samuel Pagliarini. High-speed SABER key encapsulation mechanism in 65nm CMOS. *Journal of Cryptographic Engineering*, 13(4):461–471, November 2023. doi:[10.1007/s13389-023-00316-2](https://doi.org/10.1007/s13389-023-00316-2).
- [IAR+21] Malik Imran, Felipe Almeida, Jaan Raik, Andrea Basso, Sujoy Sinha Roy, and Samuel Pagliarini. Design space exploration of saber in 65nm asic. In *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, ASHES '21*, page 85–90, New York, NY, USA, 2021. Association for Computing Machinery. doi:[10.1145/3474376.3487278](https://doi.org/10.1145/3474376.3487278).
- [IARp23] Malik Imran, Aikata Aikata, Sujoy Sinha Roy, and Samuel pagliarini. Towards high-speed ASIC implementations of post-quantum cryptography. Cryptology ePrint Archive, Report 2023/716, 2023. URL: <https://eprint.iacr.org/2023/716>.
- [Int26] International Organization for Standardization. Iso/iec 18033-2:2006/amd 2:2026, information security — encryption algorithms — part 2: Asymmetric ciphers — amendment 2, June 2026.
- [JRX25] Jake Januzelli, Lawrence Roy, and Jiayu Xu. Under what conditions is encrypted key exchange actually secure? In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part II*, volume 15602 of *LNCS*, pages 451–481. Springer, Cham, May 2025. doi:[10.1007/978-3-031-91124-8_16](https://doi.org/10.1007/978-3-031-91124-8_16).

- [KBMSRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on arm: Cca-secure module lattice-based key encapsulation on arm. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, Aug. 2018. URL: <https://tches.iacr.org/index.php/TCHES/article/view/7275>, doi:10.13154/tches.v2018.i3.243–266.
- [KCS⁺24] Suparna Kundu, Siddhartha Chowdhury, Sayandeep Saha, Angshuman Karmakar, Debdeep Mukhopadhyay, and Ingrid Verbauwhede. Carry your fault: A fault propagation attack on side-channel protected LWE-based KEM. *IACR TCHES*, 2024(2):844–869, 2024. doi:10.46586/tches.v2024.i2.844–869.
- [KDvB⁺22] Suparna Kundu, Jan-Pieter D’Anvers, Michiel van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked Saber. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 93–116. Springer, Cham, September 2022. doi:10.1007/978-3-031-14791-3_5.
- [KJGR21] Gabriel Kaptchuk, Tushar M. Jois, Matthew Green, and Aviel D. Rubin. Meteor: Cryptographically secure steganography for realistic distributions. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1529–1548. ACM Press, November 2021. doi:10.1145/3460120.3484550.
- [KKWS26] K. Kwiatkowski, P. Kampanakis, B. E. Westerbaan, and D. Stebila. Post-Quantum Traditional (PQ/T) Hybrid Key Agreement Mechanisms for TLS 1.3. Request for Comments RFC 10024, Internet Engineering Task Force, August 2026. Num Pages: 9. URL: <https://datatracker.ietf.org/doc/rfc10024>, doi:10.17487/RFC10024.
- [KS25] YoungBeom Kim and Seog Chung Seo. An optimized instantiation of post-quantum mqtt protocol on 8-bit avr sensor nodes. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ASIA CCS ’25, page 248–266, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3708821.3733873.
- [Lab24] CE Labs. libcrux: The formally verified crypto library for Rust, 2024. URL: <https://github.com/celabshq/libcrux>.
- [LCS⁺25] Jinnuo Li, Chi Cheng, Muyan Shen, Peng Chen, Qian Guo, Dongsheng Liu, Liji Wu, and Jian Weng. Grafted trees bear better fruit: An improved multiple-valued plaintext-checking

- side-channel attack against kyber. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025. doi:[10.23919/DATE64628.2025.10992764](https://doi.org/10.23919/DATE64628.2025.10992764).
- [LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Howard J. Karloff and Toniann Pitassi, editors, *44th ACM STOC*, pages 1219–1234. ACM Press, May 2012. doi:[10.1145/2213977.2214086](https://doi.org/10.1145/2213977.2214086).
- [MAA⁺20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith-Tone, and Jacob Alperin-Sheriff. Status report on the second round of the NIST post-quantum cryptography standardization process. Technical Report NIST IR 8309, National Institute of Standards and Technology, Gaithersburg, MD, July 2020. URL: <https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf>, doi:[10.6028/NIST.IR.8309](https://doi.org/10.6028/NIST.IR.8309).
- [MBMTK⁺20] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based kem. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi:[10.1109/DAC18072.2020.9218727](https://doi.org/10.1109/DAC18072.2020.9218727).
- [MJZ22] Ruiqi Mi, Haodong Jiang, and Zhenfeng Zhang. Lattice reduction meets key-mismatch: New misuse attack on lattice-based NIST candidate KEMs. Cryptology ePrint Archive, Report 2022/1064, 2022. URL: <https://eprint.iacr.org/2022/1064>.
- [MKB⁺24] Puja Mondal, Suparna Kundu, Sarani Bhattacharya, Angshuman Karmakar, and Ingrid Verbauwhede. A practical key-recovery attack on LWE-based key-encapsulation mechanism schemes using rowhammer. In Christina Pöpper and Lejla Batina, editors, *ACNS 2024, Part III*, volume 14585 of *LNCS*, pages 271–300. Springer, Cham, March 2024. doi:[10.1007/978-3-031-54776-8_11](https://doi.org/10.1007/978-3-031-54776-8_11).
- [MKKV21] Jose Maria Bermudo Mera, Angshuman Karmakar, Suparna Kundu, and Ingrid Verbauwhede. Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms. *IACR TCHES*, 2021(4):474–509, 2021. URL: <https://tches.iacr.org/index.php/TCHES/article/view/9073>, doi:[10.46586/tches.v2021.i4.474-509](https://doi.org/10.46586/tches.v2021.i4.474-509).
- [Mon18] Hart Montgomery. A nonstandard variant of learning with rounding with polynomial modulus and unbounded samples. In

- Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 312–330. Springer, Cham, 2018. doi:[10.1007/978-3-319-79063-3_15](https://doi.org/10.1007/978-3-319-79063-3_15).
- [MR19] Daniel Masny and Peter Rindal. Endemic oblivious transfer. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 309–326. ACM Press, November 2019. doi:[10.1145/3319535.3354210](https://doi.org/10.1145/3319535.3354210).
- [MX23] Varun Maram and Keita Xagawa. Post-quantum anonymity of Kyber. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCs*, pages 3–35. Springer, Cham, May 2023. doi:[10.1007/978-3-031-31368-4_1](https://doi.org/10.1007/978-3-031-31368-4_1).
- [NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked ind-cca secure saber kem implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):676–707, Aug. 2021. URL: <https://tches.iacr.org/index.php/TCHES/article/view/9079>, doi:[10.46586/tches.v2021.i4.676-707](https://doi.org/10.46586/tches.v2021.i4.676-707).
- [NDJ23] Kalle Ngo, Elena Dubrova, and Thomas Johansson. A side-channel attack on a masked and shuffled software implementation of Saber. *Journal of Cryptographic Engineering*, 13(4):443–460, November 2023. doi:[10.1007/s13389-023-00315-3](https://doi.org/10.1007/s13389-023-00315-3).
- [NR23] Parker Newton and Silas Richelson. A lower bound for proving hardness of learning with rounding with polynomial modulus. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCs*, pages 805–835. Springer, Cham, August 2023. doi:[10.1007/978-3-031-38554-4_26](https://doi.org/10.1007/978-3-031-38554-4_26).
- [NW26] Sebastiaan Neuteboom and Bas Westerbaan. 1.1.1.1 now supports post-quantum DNSSEC, all 2,420 bytes of it, September 2026. URL: <https://blog.cloudflare.com/post-quantum-dnssec-1111/>.
- [Ope25] OpenSSH. OpenSSH: Post-Quantum Cryptography, October 2025. URL: <https://www.openssh.org/pq.html>.
- [OW21] Satoshi Okada and Yuntao Wang. Recovery attack on bob’s reused randomness in CRYSTALS-KYBER and SABER. In Qiong Huang and Yu Yu, editors, *ProvSec 2021*, volume 13059 of *LNCs*, pages 155–173. Springer, Cham, November 2021. doi:[10.1007/978-3-030-90402-9_9](https://doi.org/10.1007/978-3-030-90402-9_9).

- [PC20] İrem Keskinçurt Paksoy and Murat Cenk. TMVP-based multiplication for polynomial quotient rings and application to Saber on ARM Cortex-M4. Cryptology ePrint Archive, Report 2020/1302, 2020. URL: <https://eprint.iacr.org/2020/1302>.
- [PP21] Peter Pessl and Lukas Prokop. Fault attacks on CCA-secure lattice KEMs. *IACR TCHES*, 2021(2):37–60, 2021. URL: <https://tches.iacr.org/index.php/TCHES/article/view/8787>, doi:10.46586/tches.v2021.i2.37-60.
- [PPY22] Giuseppe Persiano, Duong Hieu Phan, and Moti Yung. Anamorphic encryption: Private communication against a dictator. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 34–63. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-07085-3_2.
- [RB20] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR TCHES*, 2020(4):443–466, 2020. URL: <https://tches.iacr.org/index.php/TCHES/article/view/8690>, doi:10.13154/tches.v2020.i4.443-466.
- [RBRC20] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. Drop by drop you break the rock - exploiting generic vulnerabilities in lattice-based PKE/KEMs using EM-based physical attacks. Cryptology ePrint Archive, Report 2020/549, 2020. URL: <https://eprint.iacr.org/2020/549>.
- [RBRC22] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) nist pqc candidates for practical message recovery attacks. *IEEE Transactions on Information Forensics and Security*, 17:684–699, 2022. doi:10.1109/TIFS.2021.3139268.
- [Res26] Microsoft Research. SymCRust, 2026. URL: <https://github.com/microsoft/SymCrypt/tree/c2e575ace0ea4b6b7a4184c1f19b81d1d5b2b5be/SymCRust>.
- [RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR TCHES*, 2020(3):307–335, 2020. URL: <https://tches.iacr.org/index.php/TCHES/article/view/8592>, doi:10.13154/tches.v2020.i3.307-335.
- [RT21] Mike Rosulek and Ni Trieu. Compact and malicious private set intersection for small sets. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1166–1181. ACM Press, November 2021. doi:10.1145/3460120.3484778.

- [Sch23] Peter Schwabe. Email in pqc-forum: Discussion about Kyber’s tweaked FO transform, apr 2023. URL: <https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/WFRDl8DqYQ4/m/54var7dfAQAJ>.
- [Sch24] Sophie Schmieg. Unbindable kemmy schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK. Cryptology ePrint Archive, Report 2024/523, 2024. URL: <https://eprint.iacr.org/2024/523>.
- [Ser23] Amazon Web Services. aws-lc-rs, April 2023. URL: <https://github.com/aws/aws-lc-rs>.
- [SK18] Peter Schwabe and Matthias Kannwischer. Post-quantum crypto library for the arm cortex-m4, March 2018. original-date: 2018-03-16T09:58:41Z. URL: <https://github.com/mupq/pqm4>.
- [SKL⁺20] Bo-Yeon Sim, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Tae-Ho Lee, Jaeseung Han, Hyojin Yoon, Jihoon Cho, and Dong-Guk Han. Single-trace attacks on message encoding in lattice-based kems. *IEEE Access*, 8:183175–183191, 2020. doi:10.1109/ACCESS.2020.3029521.
- [SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 534–564. Springer, Cham, April 2019. doi:10.1007/978-3-030-17259-6_18.
- [SRB20] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):443–466, Aug. 2020. URL: <https://tches.iacr.org/index.php/TCHES/article/view/8690>, doi:10.13154/tches.v2020.i4.443-466.
- [The19] The Tor Project. obfs4 (the obfuscator), protocol specification, 2019. URL: <https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transport/lyrebird/-/blob/main/doc/obfs4-spec.txt>.
- [TUX⁺23] Yutaro Tanaka, Rei Ueno, Keita Xagawa, Akira Ito, Junko Takahashi, and Naofumi Homma. Multiple-valued plaintext-checking side-channel attacks on post-quantum KEMs. *IACR TCHES*, 2023(3):473–503, 2023. doi:10.46586/tches.v2023.i3.473-503.

- [UXT⁺22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. *IACR TCHES*, 2022(1):296–322, 2022. doi:[10.46586/tches.v2022.i1.296-322](https://doi.org/10.46586/tches.v2022.i1.296-322).
- [Val] Filippo Valsorda. C2SP — The Community Cryptography Specification Project. URL: <https://c2sp.org/>.
- [Val24] Filippo Valsorda. Let’s All Agree to Use Seeds as ML-KEM Keys, August 2024. URL: <https://words.filippo.io/ml-kem-seeds/>.
- [Val25] Filippo Valsorda. age | C2SP, December 2025. URL: <https://c2sp.org/age@v1.1.0>.
- [VJW26] Jelle Vos, Stanislaw Jarecki, and Christopher A. Wood. Hybrid Post-Quantum Password Authenticated Key Exchange. Internet Draft draft-vos-cfrg-pqpake-02, Internet Engineering Task Force, July 2026. Num Pages: 55. URL: <https://datatracker.ietf.org/doc/draft-vos-cfrg-pqpake-02>.
- [VWA⁺25] Benoît Viguier, David Wong, Gilles Van Assche, Quynh Dang, and Joan Daemen. KangarooTwelve and TurboSHAKE. Request for Comments RFC 9861, Internet Engineering Task Force, October 2025. Num Pages: 28. URL: <https://datatracker.ietf.org/doc/rfc9861>, doi:[10.17487/RFC9861](https://doi.org/10.17487/RFC9861).
- [WGY20] Bin Wang, Xiaozhuo Gu, and Yingshan Yang. Saber on ESP32. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 2020, Part I*, volume 12146 of *LNCS*, pages 421–440. Springer, Cham, October 2020. doi:[10.1007/978-3-030-57808-4_21](https://doi.org/10.1007/978-3-030-57808-4_21).
- [WJLY25] Zhedong Wang, Haoxiang Jin, Feng-Hao Liu, and Yang Yu. Revisiting the robustness of (R/M)LWR under polynomial moduli with its applications. In Goichiro Hanaoka and Bo-Yin Yang, editors, *ASIACRYPT 2025, Part III*, volume 16247 of *LNCS*, pages 167–199. Springer, Singapore, December 2025. doi:[10.1007/978-981-95-5099-9_6](https://doi.org/10.1007/978-981-95-5099-9_6).
- [WND22] Ruize Wang, Kalle Ngo, and Elena Dubrova. Making biased DL models work: Message and key recovery attacks on Saber using amplitude-modulated EM emanations. Cryptology ePrint Archive, Report 2022/852, 2022. URL: <https://eprint.iacr.org/2022/852>.

- [WZW21] Yang Wang, Yanmin Zhao, and Mingqiang Wang. On the hardness of ring/module/polynomial LWR problems. Cryptology ePrint Archive, Report 2021/1026, 2021. URL: <https://eprint.iacr.org/2021/1026>.
- [Xag22] Keita Xagawa. Anonymity of NIST PQC round 3 KEMs. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 551–581. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-07082-2_20.
- [XIU⁺21] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. Fault-injection attacks against NIST’s post-quantum cryptography round 3 KEM candidates. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 33–61. Springer, Cham, December 2021. doi:10.1007/978-3-030-92075-3_2.
- [ZHLR22] Jipeng Zhang, Junhao Huang, Zhe Liu, and Sujoy Sinha Roy. Time-memory trade-offs for saber+ on memory-constrained risc-v platform. *IEEE Transactions on Computers*, 71(11):2996–3007, 2022. doi:10.1109/TC.2022.3143441.
- [ZZY⁺21] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. Lwrpro: An energy-efficient configurable crypto-processor for module-lwr. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(3):1146–1159, 2021. doi:10.1109/TCSI.2020.3048395.

A Specification

Below we include a $\text{\LaTeX}$ translation of Kopis’s Markdown specification.

A.1 Kopis KEM Specification

In this doc we define the Kopis key encapsulation mechanism (KEM). We do this in two parts, first defining an IND-CPA-secure public key encryption (PKE) scheme, then defining the IND-CCA-secure KEM via the Fujisaki-Okamoto transform.

A.2 Preliminaries

We first specify all the algorithms, syntax, and mathematics we will need for the specification.

A.2.1 Dependencies

We use the TurboSHAKE XOF family defined in [RFC 9861](#). We invoke it as TurboSHAKE128/TurboSHAKE256(M , L , D), where M is the message to be hashed, L is the desired output length, and D is the domain separator in the range $[0 \times 01, 0 \times 7f]$.

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [BCP 14 RFC 2119 RFC 8174](#) when, and only when, they appear in all capitals, as shown here.

A.2.2 Syntax

We use pseudocode resembling a mix of Rust and Python. Variables declared with `let mut` are mutable. Function definitions are preceded with `fn`, each function input name is followed by a colon, then the type signature, and each function is followed by an arrow `→` then the return type. Ranges are denoted `a..b`, and indicate the range $[a, b)$ (i.e., including a , excluding b). When s is a sequence type (e.g., a `bitstring` or `bytestring`), `s[a..b]` is used to denote the subsequence starting at index a (0-indexed), and ending at and excluding index b . We use underscores in the LHS of assignments to denote the elision of a value that is normally bound, e.g., `let (a, _) = f()` where `f()` returns two values. We use ellipses to denote the elision of multiple values, e.g., `let (a, ...) = f()`, where `f()` returns three values. We use infix `||` to denote concatenation of `bytestrings`. We use infix `^` to denote integer exponentiation when the inputs are expressions. We write `[uK; N]` to mean an array of N many K -bit integers. Subtraction over `uK` is always defined as wrapping subtraction, e.g., `0u8 - 1u8 = 255u8`.

A.2.3 Mathematical Definitions

Let R be the negacyclic polynomial ring $\mathbb{Z}[X]/(X^{2^{56}} + 1)$. We denote by $R13$ the polynomial ring modulo 2^{13} , i.e., $R/2^{13}R$ (this is isomorphic to $(\mathbb{Z}/2^{13}\mathbb{Z})[X]/(X^{2^{56}} + 1)$). Similarly, $R10$ denotes $R/2^{10}R$ and $R1$ denotes $R/2R$.

The type MatRn refers to $R^n^{\ell \times \ell}$, i.e., $\ell \times \ell$ matrices of elements of Rn , for any n . Similarly, the type VecRn refers to R^n^ℓ , i.e., vectors of ℓ elements of Rn . For any n , we write make_rn to refer to the natural morphism from the space of coefficients $[\text{un}; 256]$ to Rn (input is interpreted lowest-degree-coefficient-first). make_matn takes a nested array $[[Rn; \ell]; \ell]$ and interprets it as a list of rows in a matrix in MatRn . make_vecn takes an array $[Rn; \ell]$ and interprets it as a column vector in VecRn , in the same order as make_matn , i.e., such that $\text{make_matn}([\text{eye}, \text{zeros}, \dots, \text{zeros}]) * \text{make_vecn}(\text{eye}) = \text{make_vecn}(\text{eye})$, where $\text{eye} = [1, 0, \dots, 0]$, and $\text{zeros} = [0, 0, \dots, 0]$ (interpreting the numbers as ring elements). When indexing into a vector $v: \text{VecRn}$ for some n , we do so in the same order that was used in its constructor, i.e., $v[i]$ equals $a[i]$ where a is the input to the make_vecn that constructed v . transpose transposes the given matrix or vector.

When we write r as U for some type U , we mean to invoke either the natural injection or projection of r into/onto U . For example, if r is in VecR10 and U is VecR13 , then it is the natural injection, and if vice-versa, then it is the natural projection by quotienting by $2^{10}R13$.

For any modulus N , we say the canonical form of an element of $\mathbb{Z}/N\mathbb{Z}$ is its representative integer in $[0, N)$. For any n and a ring element r we say its *canonical coefficients* $\text{canonical_coeffs}(r: Rn) \rightarrow [\text{un}; 256]$ are the unique sequence of 256 un values $a_0, \dots, a_{255}$ such that $r = a_{255} X^{255} + \dots + a_1 X + a_0$.

For an element r in Rn and integer N , we define the right-shift $r \gg N$ as $\text{make_rn}([a_0 \gg N, a_1 \gg N, \dots, a_{255} \gg N])$ where a_i are the canonical coefficients of r . We define left-shift $r \ll N$ similarly, with coefficients reduced mod 2^n . We define right and left shift for elements of VecRn as operating element-wise.

We define $\text{to_bits_le}(n: \mathbb{Z}, k: \text{un}) \rightarrow [\text{bool}; n]$ to be the function that converts an n -bit integer to its bit representation, starting with the least significant bit. Similarly, we define $\text{from_bits_le}(n: \mathbb{Z}, \text{bits}: [\text{bool}; n]) \rightarrow \text{un}$ to interpret n bits as a un value, using $\text{bits}[0]$ as the least significant bit of the output, and so on.

A.3 Main Algorithms

We now define a **public key encryption (PKE) scheme**. For the purposes of this specification, the secret key space is simply $[\text{u8}; 32]$, and generating a fresh secret key amounts to generating a fresh uniform bytestring. We define a public key encryption scheme as the set of the following algorithms:

- $\text{SkToPk}(\text{sk}) \rightarrow \text{pk}$ — Computes the public key corresponding to the given secret key

- $\text{PkeEncrypt}(\text{randomness}, \text{pk}, \text{msg}) \rightarrow \text{ct}$ — Encrypts the given message msg to public key pk , using encryption randomness randomness
- $\text{PkeDecrypt}(\text{sk}, \text{ct}) \rightarrow \text{msg}$ — Decrypts the given ciphertext ct using the secret key sk , and unconditionally returns a message msg

We now define a **key encapsulation mechanism (KEM)**. Similar to the PKE above, secret keys are uniform elements of $[\text{u8}; 32]$. A KEM is defined as the set of the following algorithms:

- $\text{SkToPk}(\text{sk}) \rightarrow \text{pk}$ — Computes the public key corresponding to the given secret key
- $\text{KemEncap}(\text{randomness}, \text{pk}) \rightarrow (\text{k}, \text{ct})$ — Encapsulates a shared secret k to public key pk , using randomness randomness . Outputs a k and a ciphertext ct .
- $\text{KemDecap}(\text{sk}, \text{ct}) \rightarrow \text{k}$ — Decrypts the ciphertext ct using the secret key sk , and unconditionally returns a shared secret k

For real-world usage of these algorithms, the randomness parameters above **MUST** be sampled uniformly.

An implementation note: the purpose of Kopsis is to be used in settings that require constant-time operations. Thus, any implementation of the above functions **MUST** be constant-time with respect to all inputs.

A.3.1 Parameters

The following variables represent security parameters, and depend on the security level being instantiated:

- ℓ — The public matrix dimension. This impacts the size of public keys and ciphertexts
- μ — The binomial parameter used for secret generation. This is always even.
- t — The base-2 logarithm of the modulus of the space of compressed ring elements

A.3.2 Constants

We define the constants used in our implementation. The sizes in bytes of our secret keys, public keys, and ciphertexts are functions of the parameters above:

- $\text{SK_SIZE} = 32$
- $\text{PK_SIZE} = 256 \cdot \ell \cdot 10/8 + 32$
- $\text{CT_SIZE} = 256 \cdot t/8 + 256 \cdot \ell \cdot 10/8$

We also require domain separators for all our TurboSHAKE invocations:

- DOMSEP_KGEXPAND = 0×01
- DOMSEP_GENMAT = 0×02
- DOMSEP_GENSEC = 0×03
- DOMSEP_PKHASH = 0×04
- DOMSEP_F0 = 0×05
- DOMSEP_NOREJECT = 0×06

A.3.3 PKE

We define key generation, encryption, and decryption for the Kopsis IND-CPA-secure PKE scheme. We will define the helper functions later.

```

fn SkToPk(sk: [u8; 32]) -> [u8; PK_SIZE]:
  let (_, _, pk, _) = ExpandSecretKey(sk)
  return pk

fn PkeEncrypt(
  randomness: [u8; 32],
  pk: [u8; PK_SIZE],
  msg: [u8; 32]
) -> [u8; CT_SIZE]:
  let vec_b = deserialize_vec(10, pk[..256*10/8])
  let mat_seed = pk[256*10/8..]
  let m = deserialize_elem(1, msg)

  let mat_A = GenMat(mat_seed)
  let vec_sprime = GenSecret(randomness)
  let vec_bprime = CompressToR10(mat_A * vec_sprime)
  let vprime = transpose(vec_b) * (vec_sprime as VecR10)
  let cm = CompressToRt(vprime - ((m as R10) << 9))

  let ct = serialize_vec(10, vec_bprime) || serialize_elem(t, cm)
  return ct

fn PkeDecrypt(sk: [u8; 32], ct: [u8; CT_SIZE]) -> [u8; 32]:
  let (vec_s, ...) = ExpandSecretKey(sk)
  let vec_bprime = deserialize_vec(10, ct[..256*10/8])
  let cm = deserialize_elem(t, ct[256*10/8..])
  let v = transpose(vec_bprime) * (vec_s as VecR10)
  let cm10 = (cm as R10) << (10 - t)
  let mprime = DecodeMsg(v - cm10)
  return serialize_elem(1, mprime)

```

A.3.4 KEM

We define the IND-CCA-secure Kopis KEM below. The SkToPk function is identical to the one given in the PKE above.

```
fn KemEncap(
  randomness: [u8; 32],
  pk: [u8; PK_SIZE]
) -> ([u8; 32], [u8; CT_SIZE]):
  # Derive the output key and encryption randomness
  let pkh = TurboSHAKE256(pk, 32, DOMSEP_PKHASH)
  let b = TurboSHAKE256(randomness || pkh, 64, DOMSEP_F0)
  let (k, r) = (b[..32], b[32..])

  # Encrypt to pk. `randomness` is itself the message
  let ct = PkeEncrypt(r, pk, randomness)

  return (k, ct)

fn KemDecap(sk: [u8; 32], ct: [u8; CT_SIZE]) -> [u8; 32]:
  let (_, z, pk, pkh) = ExpandSecretKey(sk)

  let randomness = PkeDecrypt(sk, ct)
  let b = TurboSHAKE256(randomness || pkh, 64, DOMSEP_F0)
  let (k, rprime) = (b[..32], b[32..])
  let cprime = PkeEncrypt(rprime, pk, randomness)

  if ct == cprime:
    return k
  else:
    return TurboSHAKE256(z || ct, 32, DOMSEP_NOREJECT)
```

We note again that, along with all other top-level functions, `KemDecap` MUST be constant time with respect to its inputs. In particular, an implementer MUST perform the ciphertext equality check in constant time.

For efficiency, implementers MAY internally cache the expanded decapsulation key. But this expanded key SHOULD NOT be persisted anywhere.

A.3.5 Auxiliary Functions

We now define the auxiliary functions used in the schemes above:

```
fn ExpandSecretKey(
  sk: [u8; 32]
) -> (VecR13, [u8; 32], [u8; PK_SIZE], [u8; 32]):
  let randomness = TurboSHAKE256(sk || (ℓ as u8), 96, DOMSEP_KGEXPAND)
  let mat_seed = randomness[..32]
```

```

let secret_seed = randomness[32..64]
let z = randomness[64..]

let mat_A = GenMat(mat_seed)
let vec_s = GenSecret(secret_seed)
let vec_b = CompressToR10(transpose(mat_A) * vec_s)
let pk = serialize_vec(10, vec_b) || mat_seed
let pkh = TurboSHAKE256(pk, 32, DOMSEP_PKHASH)

return (vec_s, z, pk, pkh)

fn CompressToR10(v: VecR13) -> VecR10:
  let h1 = make_r13([4u13; 256])
  let h = make_vec13([h1; 1])
  let s = (v + h) >> 3
  return (s as VecR10)

fn CompressToRt(r: R10) -> Rt:
  let h1 = make_r10([4u10; 256])
  let s = (r + h1) >> (10 - t)
  return (s as Rt)

fn DecodeMsg(r: R10) -> R1:
  let h2 = make_r10([2^8 - 2^(10-t-1) + 4; 256])
  let s = (r + h2) >> 9
  return (s as R1)

fn GenMat(seed: [u8; 32]) -> MatR13:
  let A: [[R13; 1]; 1]
  for i in 0u8..1:
    for j in 0u8..1:
      let buf = TurboSHAKE128(seed || i || j, 256*13/8, DOMSEP_GENMAT)
      A[i][j] = deserialize_elem(13, buf)
  return make_mat13(A)

fn GenSecret(seed: [u8; 32]) -> VecR13:
  let s: [R13; 1]
  for i in 0u8..1:
    let buf = TurboSHAKE256(seed || i, 256*13/8, DOMSEP_GENSEC)
    let vals = bit_slices(buf)
    let r: [u13; 256]
    for k in 0..256:
      # Recall subtraction is wrapping
      r[k] = hamming(vals[2*k]) - hamming(vals[2*k+1])
    s[i] = make_r13(r)
  return make_vec13(s)

```

```

# Serializes an element of Rn (for any choice n=13,10,1,t)
fn serialize_elem(n:  $\mathbb{Z}$ , r: Rn) -> [u8; n*256/8]:
  let a = canonical_coeffs(r)

  let all_bits: [bool; n*256]
  for i in 0..256:
    all_bits[n*i..n*(i+1)] = to_bits_le(n, a[i])

  let out: [u8; 32*n]
  for i in 0..32*n:
    out[i] = from_bits_le(8, all_bits[8*i..8*(i+1)])
  return out

# Serializes an element of VecRn (for any choice n=13,10,1,t)
fn serialize_vec(n:  $\mathbb{Z}$ , v: VecRn) -> [u8;  $\ell$ *n*256/8]:
  let out: [u8;  $\ell$ *n*32]
  for i in 0.. $\ell$ :
    out[n*32*i..n*32*(i+1)] = serialize_elem(n, v[i])
  return out

# Deserializes an element of Rn (for any choice n=13,10,1,t)
fn deserialize_elem(n:  $\mathbb{Z}$ , bytes: [u8; n*256/8]) -> Rn:
  let all_bits: [bool; n*256]
  for i in 0..n*32:
    all_bits[8*i..8*(i+1)] = to_bits_le(8, bytes[i])

  let coeffs: [un; 256]
  for i in 0..256:
    coeffs[i] = from_bits_le(n, all_bits[n*i..n*(i+1)])
  return make_rn(coeffs)

# Deserializes an element of VecRn (for any choice n=13,10,1,t)
fn deserialize_vec(n:  $\mathbb{Z}$ , bytes: [u8;  $\ell$ *n*256/8]) -> VecRn:
  let elems: [Rn;  $\ell$ ]
  for i in 0.. $\ell$ :
    elems[i] = deserialize_elem(n, bytes[n*32*i..n*32*(i+1)])
  return make_vecn(elems)

# Reinterprets a bytestring as a sequence of bitstrings of length  $\mu/2$ 
fn bit_slices(bytes: [u8;  $\mu$ *256/8]) -> [[bool;  $\mu/2$ ]; 512]:
  let all_bits: [bool;  $\mu$ *256]
  for i in 0.. $\mu$ *32:
    all_bits[8*i..8*(i+1)] = to_bits_le(8, bytes[i])

  let out: [[bool;  $\mu/2$ ]; 512]

```

```

for i in 0..512:
    out[i] = all_bits[i*μ/2..(i+1)*μ/2]
return out

# Returns the number of set bits in b
fn hamming(b: [bool; μ/2]) -> u13:
    let mut weight = 0u13
    for i in 0..μ/2:
        if b[i]:
            weight += 1
    return weight

```

A.4 Parameter Sets

We define three security levels for Kopis: Kopis-512, Kopis-768, and Kopis-1024, referring to the dimension of the public key vector over $\mathbb{Z}/2^t\mathbb{Z}$:

Name	Parameters	PK_SIZE	CT_SIZE	SK_SIZE
Kopis-512	$\ell=2$ $t=3$ $\mu=10$	672	736	32
Kopis-768	$\ell=3$ $t=4$ $\mu=8$	992	1088	32
Kopis-1024	$\ell=4$ $t=6$ $\mu=6$	1312	1472	32

B Implementing High-Assurance Code with an LLM

The majority of our **Kopis** Rust implementation was written by an LLM. Nonetheless, we claim this is high-assurance code, due to the strict constraints we placed around the code production process, using unit tests, known-answer tests, linters, and, most importantly, formal verification in Lean. We built our implementation incrementally, beginning with short, human-audited (LLM-written) Rust code and Lean theorems and ending with a codebase with 6.7 kloc of Rust and 76 kloc of Lean.

We describe our design and implementation process below, as this is of potential independent interest and may inform future work. The LLMs used were Claude Opus versions 4.8 and 5.

B.1 Building the Trusted Base

We describe the trusted starting points from which we built the rest of the **Kopis** codebase.

English Specification. In order to build the high-assurance codebase, we first needed to define a source of truth. We wrote a Markdown specification for **Kopis**, reusing as much structure as possible from **Saber**. We also specified all functions with explicit randomness, to make testing easier later on. We performed multiple passes over the specification, both with and without the help of an LLM, tweaking language until we were satisfied that all ambiguities were resolved.

Lean Specification. We used an LLM to produce a Lean specification of **Kopis** from the Markdown specification. We reused the **Keccak** primitive from **SymCRust** [Res26], a recently published Lean-verified implementation of **ML-KEM-768**. The LLM-produced specification of **Kopis** required just 209 lines of Lean code to be human-audited, which was done without much difficulty.

B.2 Building Bridges

At this point, we still had not run any code. So we produced test vectors to test on both implementations. We had an LLM produce a small Python implementation of **Kopis** and output known-answer test (KAT) vectors, covering each security level, honest executions, executions with mauled ciphertexts,¹⁷ and executions with values that are malformed (e.g., wrong ciphertext length). We spot-checked the Python code but did not do a close reading.

Rust KAT runner. To consume the vectors and start our **Kopis** Rust implementation, we had an LLM modify a pre-existing human-written and well-tested **Saber** implementation in Rust. We then gave an LLM the specification and a description of the KAT file (importantly, *not* the Python implementation itself),

¹⁷These were missed in the first iteration and added later on.

and asked it to look at the pre-existing Saber Rust codebase and modify it until it implemented **Kopis**. We reviewed the resulting diff of +104/−128 manually.¹⁸ With the diff merged, we prompted the LLM to make one more change: to implement KATs using test vectors in the format described above. The diff was +182 lines of Rust, which we manually reviewed.

Lean KAT runner. At the same time, we also had an LLM build a known-answer test harness using a description of the KAT file. This PR was less than 100 lines of Lean code, which we spot-checked but did not do a close reading of.

With all test harnesses set up, we passed the test vectors into the Lean and Rust implementations and found that all passed.

B.3 Formally Verifying the Rust

With trustworthy Rust and Lean implementations of **Kopis** in hand, we could then begin proving their equivalence. This was a relatively hands-off process.

Proving Rust. We used Aeneas [HP22] to extract our Rust implementation to Lean, and began prompting the LLM to prove equivalence of the extracted Lean to our Lean specification from the previous section. As a side benefit, a proof of correctness against our Lean specification also proves that the Rust code is panic-free.

To aid the LLM, we gave it access to **SymCRust** to copy techniques if any were relevant. We started by prompting for proofs of equivalence for ring addition, then for serialization and deserialization. These proofs succeeded and brought to our attention that our **TurboSHAKE** dependency needed to be axiomatized, since we do not implement it directly in our crate. Similarly, our dependency on **subtle**, a crate for constant-time selection operations (necessary for the Fujisaki-Okamoto transform), was axiomatized. Finally, since they were not yet covered by Aeneas, some basic Rust intrinsics were axiomatized, namely `count_ones` and `wrapping_neg`.

We then moved on to ring multiplication and `expand_decap_key`, then on to PKE encryption and decryption. No issues arose in any of these processes, so we asked the LLM to just complete the proof for the KEM, which it did.

Making theorems auditable. Lean proofs are not useful unless we can verify the theorems. By the end of the above process, there were about 14 kloc of Lean code, not including comments. We asked the LLM to extract the top-level theorems and place them in their own file, so we could manually review them. After several rounds of rephrasing theorems, choosing what counted as top-level, and reformatting for readability, we ended up with 18 theorems (many of them duplicates because of the three security levels) describing the correspondence between the extracted Rust API and the Lean implementation. These theorems were placed in a separate file `TopLevelTheorems.lean` and manually reviewed.

¹⁸The deleted code included easily visually skippable functions such as repetitive hash computations and serialization/deserialization routines for Saber KEM and PKE secret keys. The remaining changes in the PR were variable renames and deletion of Saber KATs, both easily visually skippable using a side-by-side word-diffing UI.

Checking for sorry. The top-level theorems are useless if there are hidden axioms in the proofs or proofs using `sorry` (a tactic which tells the prover that no proof is present). To check this, we had an LLM produce a file called `TrustBase.lean`, which dumps all axioms used in the project and matches them to a list of approved axioms. If any axiom is unapproved, an error is raised. We checked this axiom list to ensure it was what we encountered in the above steps.

B.4 (Mostly) Fearless Optimizing

Once we had a functioning Lean pipeline for updating proofs, it became extremely easy to let the LLM perform arbitrary optimizations. For example, the LLM produced code specializing serialization and deserialization for specific moduli. We simply let it produce the code, verified that it improved benchmarks, and let it fix the Lean proofs. All we needed to check was that the top-level theorem file was unchanged.

Using NTT. We asked the LLM to make high-impact optimizations to the crate. The route it chose was the number-theoretic transform (NTT) described in [CHK⁺21, Section 4], specifically the one which uses a semiprime modulus just greater than 2^{25} , so that all `Kopis` operations fit without overflow. This modification improved our encapsulation and decapsulation runtimes by nearly $2\times$.

The cost was that our internal representation of the public and secret keys, now stored in the NTT domain, no longer matched the Lean specification. To accommodate this change, we modified the top-level theorems to be independent of internal representation. For example, rather than showing that an internal representation of `pk` matches the Lean specification, it suffices to show that (1) deserializing a public key and encapsulating to it produces the same ciphertext and shared secret as when the same is done in Lean, and (2) deserializing a public key and serializing it is the identity.

Adding SIMD. Next, we had an LLM create SIMD backends. We chose the same NTT for all backends and introduced compile-time flags to enable or disable them at compile time (with runtime detection if otherwise unspecified). We produced an AVX2 backend and a NEON backend (supporting only CPUs which also have SHA-3 hardware acceleration). The top-level theorems remained the same for all backends.

Like the NTT optimization, the new backends required changes to the trust base. During proving, we discovered that Rust AVX2 and NEON instructions are not covered by Aeneas. And so every instruction we used needed to be axiomatized. We had the LLM create a list of these (48 NEON and 55 AVX2), axiomatize their definitions, produce Rust programs that produce test vectors, and produce Lean code to check the test vectors against the axioms.

Parallelizing TurboSHAKE. In our AVX2 backend, we found that we achieved a significant speedup if we implemented a 4-way parallel TurboSHAKE squeezing algorithm. To prove this correct, we needed a Lean specification for TurboSHAKE. Since Keccak was already provided by `SymCRust`, a TurboSHAKE implementa-

tion was not far off. We had an LLM produce a Lean specification of TurboSHAKE, requiring just 45 lines of Lean code to be human-audited. In non-AVX2 backends, an external crate was still used for TurboSHAKE computations, so those calls remained axiomatized.

Checking for constant-timeness. At this point, most of the code in the crate had been produced by an LLM. The guardrails of the Lean proofs mean that we can be assured of correctness. But Lean proofs cannot tell us whether our implementations are properly constant-time. There are two primary sources of non-constant-timeness: an explicit branch over a secret value, and use of a non-constant-time operation, such as integer division.

We had an LLM produce checks for both kinds of constant-timeness. The first is a Valgrind test harness which performs taint-testing, and the second is a manual scan through the compiled binaries to look for non-constant-time instructions. The taint-testing was done by marking secret bytes as undefined, and running Valgrind’s memcheck. When this happens, any branching based on undefined values will throw a fatal error. Thus, this catches secret-based branching. For the manual scan, we had an LLM produce a list of plausible instructions that are not constant-time in the various architectures (7 instructions total), and search for them in the compiled binaries. We also introduced an allowlist for specific instances where instructions are used in a benign way (e.g., division involving two public values).

B.5 Cortex-M4 implementation

We built our ARM Cortex-M4 implementation on top of the `pqm4` library [SK18], a collection of the current state-of-the-art implementations of KEMs and signature algorithms for the Cortex-M4.

The first thing we did was recover the **Saber** implementation. It was removed from `pqm4` after it was eliminated from the NIST competition. So we copied the last known implementation (commit 3743a66). We note that each **Saber** security level has two implementations: one optimized for speed and one optimized for stack usage. For the rest of this section, when we say “implementation,” we are referring to both.

To prepare a **Kopis** implementation, we copied the **Saber** implementation to its own folder. Next, we told an LLM to read the **Kopis** Markdown spec and convert that folder into a **Kopis-768** implementation, optimizing for a minimal diff. We manually reviewed the resulting diff, which was +344/−102 lines.

Finally, we extended the implementations to **Kopis-512** and **Kopis-1024**. Like the **Saber** implementation, these are identical, except for a two-line difference in the parameters file (ℓ changes, and the remaining parameters are already defined as a function of ℓ). The rest of the files are symlinks to the corresponding files in the **Kopis-768** implementation.

As a final check, we had an LLM create a test harness for the code. It produced code to run all **Kopis** implementations and produce test vectors. We also had an LLM extend the Rust implementation to consume these KATs.

These commits were +144/-0 and +179/-0, respectively. We reviewed these by hand. All KATs passed.