Masked Key Wrapping and Mask Compression

Markku-Juhani O. Saarinen$^{[0000-0002-2555-235X]}$

PQShield Ltd. Oxford, UK
mjos@pqshield.com

Abstract. Side-channel secure implementations of public-key cryptography algorithms must be able to load and store their secret keys safely. We describe WrapQ, a masking-friendly key management technique and encoding format for Kyber and Dilithium Critical Security Parameters (CSPs). WrapQ protects secret key integrity and confidentiality with a Key-Encrypting Key (KEK) and allows the keys to be stored on an untrusted medium. Importantly, its encryption and decryption processes avoid temporarily collapsing the masked asymmetric secret keys (which are plaintext payloads from the viewpoint of the wrapping primitive) into an unmasked format. We demonstrate that a masked Kyber or Dilithium private key can be loaded any number of times from a compact WrapQ format without updating the encoding in non-volatile memory. We also consider the keys-in-RAM use case (without the write-back restriction) and introduce Mask Compression, a technique that leverages fast, unmasked deterministic samplers. Mask compression saves working memory while reducing the need for true randomness and is especially useful when higher-order masking is applied in lattice cryptography. The techniques have been implemented in a side-channel secure hardware module. Kyber and Dilithium wrapping and unwrapping functions were validated with 100K traces of TVLA-type leakage assessment.

Keywords: Side-Channel Security · Masking Countermeasures · Key Wrapping · Post-Quantum Cryptography · Kyber · Dilithium

1 Introduction

With the standardization of CRYSTALS suite algorithms Kyber [3] and Dilithium [4] as the preferred NIST Post-Quantum Cryptography (PQC) algorithms for key agreement and digital signatures [1], their secure and efficient implementation has become one of the most important engineering challenges in cryptography. NSA has also selected these two algorithms for the CNSA 2.0 suite for protecting classified information in National Security Systems [40].

Among other applications, Kyber and Dilithium will be gradually replacing older RSA and Elliptic Curve systems in systems where it is a requirement that the secret information held by a device (such as a mobile phone, payment card, or an authentication token) does not leak even if an adversary has access to the physical device (or its close proximity.) A related application is *platform security*, where cryptographic methods protect system integrity and system (firmware) updates against unauthorized modification and other attacks.

Side-Channel Attacks (SCA) use external physical measurements to derive information about the data being processed. Some of the most important ones are Timing Attacks (TA) [30], Differential Power Analysis (DPA) [31], and Differential Electromagnetic Analysis (DEMA) [42]. The attacks are powerful, and almost any implementation can be rapidly attacked if appropriate countermeasures are not in place.

1.1 Side-Channel Countermeasures for Lattice Cryptography

Masking countermeasures have emerged as the most prominent and effective way to secure lattice-based cryptography against side-channel attacks. Masking is based on randomly splitting all secret variables into two or more shares.

Definition 1. *Order-d masked encoding* $[[x]]$ *of a group element* $x \in G$ *consists of a tuple of* $d+1$ *shares* $(x_0, x_1, \cdots, x_d), x_i \in G$ *with* $x_0 + x_1 + \cdots + x_d \equiv x$.

The addition operation can be defined in an arbitrary finite group G; Boolean masking uses the exclusive-or operation $\oplus$, while arithmetic masking uses modular addition. Vectors, matrices, and polynomials can also be used as shares.

A fundamental security requirement is that the shares are randomized so that all $d+1$ shares are required to reconstruct x, and any subset of only d shares reveals no statistical information about x itself. *Mask refreshing* refers to a re-randomization procedure that maps $[[x]]$ to another encoding $[[x]]'$ of x.

Computation on shares is organized so that it can be performed without reconstructing the full secret variable at any point. It can be shown that the amount of side channel information required to learn a secret grows exponentially in d, the number of additional shares, while any circuit can be transformed to use masking with at most $O(d^2)$ overhead [12,26].

Several abstract models have been proposed for the purpose of providing theoretical proofs of side-channel security of masked implementations, including the Ishai-Sahai-Wagner probing model [26] and Prouff-Rivain noisy leakage model [44,41]. Duc et al. provide a reduction from the latter to the first [18].

In addition to theoretical soundness, an essential advantage of PQC masking countermeasures is that they are defined at a higher algorithmic level and are generally less dependent on the physical details of the implementation when compared to logic-level techniques such as dual-rail countermeasures [2]. However, it is essential to experimentally verify that the leakage properties of the implementation match the theoretical expectations.

Designers often proceed by describing a set of generic “gadgets” that make up the secured portion of the algorithm and then providing analysis for the composition. There already exists a large body of works discussing the masking of lattice cryptography schemes, including GLP [6], Dilithium [35] and Kyber [11,24]. The issue of key management is generally not addressed in these works.

An additional security notion in the probing model is SNI (Strong Non-Interference) [5], which allows better composability: Almost any combination of SNI gadgets is SNI too. However, the ease of theoretical composability and analysis comes at a cost; SNI gadgets may require additional randomness. SNI provides theoretical assurance, but the relationship with actual attack mitigation is sometimes unclear. We leave SNI analysis for future work.

1.2 Private Keys and Secret Variables

Side-channel leakage can be exploited in any component that handles secret key material. Hence the key management processes must meet the same security requirements as cryptographic private key operations. Often the "zeroth" step of a private-key operation is "load private key."

Secure, non-volatile key storage is a limited resource. Standard-format Kyber1024 private keys are 25,344 bits, while Dilithium5 secret keys are 38,912 bits (Table 3.) This is an order of magnitude more than typical RSA keys and two orders of magnitude more than the keys of Elliptic Curve Cryptography schemes.

Key Wrapping [19,46] refers to a process where Authenticated Encryption (AE) is used to protect the confidentiality and integrity of other key material, such as asymmetric keys. Key wrapping reduces much of the problem of secure key storage to that of protecting the shorter AE keys.

Masking side-channel countermeasures can significantly increase the secret key storage requirement. Masking requires that the shares are refreshed (re-randomized) every time they are used. A trivial solution is to write back the refreshed keys to non-volatile memory after each usage, which risks corruption of the keys. It is preferable if long-term secret keys can be stored statically.

1.3 Outline of this work and Our Contributions

Section 2 describes the *side-channel secure key wrapping* problem and outlines the WrapQ key import and export methods. We discuss its properties, including the classification and protection of Kyber and Dilithium Critical Security Parameters (CSPs). The technique allows compact storage of long PQC secret keys on an untrusted medium and their side-channel secure loading for use.

In Section 3 we introduce *compressed masking*, a straightforward method for encoding long vector shares as compact cryptographic keys. This method can be used to compress temporary variables and also to make (ephemeral) internal key management more efficient.

Appendix A describes an FPGA implementation and "TVLA" experiments on importing and exporting Kyber and Dilithium keys using WrapQ. The module is found not to leak in 100K traces. Power and t-traces for Kyber1024 and Dilithium5 are supplied.

2 WrapQ: Masked Key Import and Export

Most works on side-channel secure implementations of symmetric ciphers (such as AES) focus on protecting the symmetric key; in a standard model, the attacker can observe and even choose both plaintext and ciphertext. For Key Wrapping, we have an additional goal: its "plaintext" (i.e., the wrapped asymmetric key payload) also remains invisible to side-channel measurements.

An approach that first decrypts a standard serialization of a secret key and only then splits it into randomized shares (Definition 1) will leak information in repeat observations; even partial information about coefficients can be used to accelerate attacks. One can also consider encrypting the individual masked shares, which significantly increases the size of the key blob. However, when importing the same key blob multiple times, the decrypted masked key is static (not uniform), potentially leaking information. From the attackers' viewpoint, a secret key in static shares is not much different from an unmasked key; it is just a longer "expanded key". A standard solution would be to write a refreshed, re-encrypted secret key back every time the key is used, but this approach has severe practical disadvantages in addition to a much larger key blob, such as reliability risks.

2.1 Masked Key Wrapping

WrapQ implements masked Key Wrapping [19] for lattice cryptography with a special type of Authenticated Encryption with Associated Data (AEAD) [45] mechanism. An abstract high-level interface is:

$$\textit{Key Wrapping}: \qquad C \leftarrow \mathsf{WrapQ}(\ [[K]], [[P]], AD\ ) \tag{1}$$

$$\textit{Key Unwrapping}: \qquad \{\ [[P]], \mathsf{FAIL}\ \} \leftarrow \mathsf{WrapQ}^{-1}(\ [[K]], C, AD\ ) \tag{2}$$

Where

[[**K**]] Symmetric key(s) for integrity and confidentiality protection. Supplied as Boolean shares.

[[**P**]] Payload: Asymmetric key material to be encrypted. A set of masked (Boolean/Arithmetic) quantities.

AD Authenticated Associated Data: Public values that only require integrity protection.

C Wrapped key blob containing encrypted P, authentication information for AD and P, and auxiliary information such as nonces (IV.)

In addition to standard AEAD security goals, the primitive guarantees that the payload secret key material $[[P]]$ is not "collapsed" into leaking unmasked format during the wrapping or unwrapping processes. Each unwrapping call WrapQ^{-1} produces a fresh, randomized masking representation for $[[P]]$ variables, or FAIL in case of authentication (integrity) failure.

2.2 Design Choices

WrapQ has several design choices motivated by its particular use case; a side-channel secure hardware module that implements lattice-based cryptography. It is not intended as a universal encoding for private keys.

Key Loading. The term "import" does not necessarily imply interaction with external devices. The import function simply prepares and loads a private key from static storage to be used by the cryptographic processor. Importing may occur during device start-up or if there is a change of keys. Key export is required when new keys are generated or if KEK changes. Side-channel considerations are equally important in these use cases.

Key Encryption Key. We primarily wanted to secure the process of local, automatic, unsupervised loading of secret keys for immediate use. Some hardware devices may use a device-unique key or a Physically Unclonable Function (PUF) to derive the KEK, with the idea that keys exported to a less trusted storage can only be imported back into the same physical module [33]. Since the main goal is side-channel security, the storage format may be modified to accommodate implementation-specific requirements.

Non-Deterministic. Rogaway and Shrimpton [46] argue that a key wrapping operation should be fully deterministic; the inputs K, P, A fully determine C without randomization. Their motivation is that removing the nonce IV will save some bandwidth. We prioritize side-channel security and observe that randomization helps to eliminate leakage in the export function.

Secondary Encryption. WrapQ only encrypts critical portions of the key material. It is a "feature" that algorithm identifiers and the public key hash are unencrypted; this makes it possible to retrieve a matching public key before validating the secret key blob. WrapQ key blobs do not have complete confidentiality properties, such as indistinguishability from random. However, the resulting blob is much safer to handle as critical variables are encrypted; a secondary confidentiality step can use arbitrary mechanisms to re-encrypt it.

Not (necessarily) a key interchange format. Export can also occur between devices; sometimes, the term "Key Exchange Key" is used to export a key from one HSM to another or from an on-premises system to the cloud [34]. In such "one-off" manual use cases, side-channel protections may be less critical, and mechanisms such as PKCS #12 [36] can be used (after additional authorization).

2.3 Method Outline

Masked XOF. WrapQ uses a masked XOF (extensible output function [37]) as a building block for all of its side-channel secure cryptographic functionality.

Definition 2. *An Order-d masked extensible output function* $[[h]] \leftarrow \mathsf{XOF}_n([[m]])$ *processes arbitrary-length masked input* $[[m]]$ *into n-byte output shares* $[[h]]$ *while maintaining Order-d security (under some applicable definition.)*

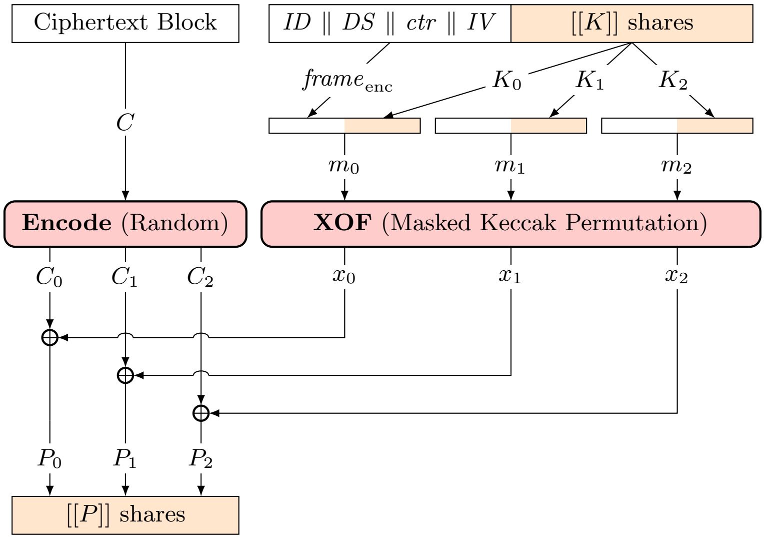

Fig. 1. WrapQ import uses a masked XOF in counter mode to decrypt ciphertext blocks C into randomized Boolean shares $[[P]]$. A masked SHAKE (1600-bit Keccak Permutation, pictured here with three shares) is a common requirement for side-channel secure implementations of Dilithium and Kyber, so it can be expected to be available.

Instantiation. The XOF can be instantiated with a masked Keccak[1600] [37] permutation. Note that a masked SHA3/SHAKE (and hence a masked Keccak permutation) is required to process secret variables in Kyber (G, PRF, KDF) and in Dilithium (H, $\mathsf{ExpandS}$, $\mathsf{ExpandMask}$). Hence this masked primitive can be expected to be available in masked Kyber and Dilithium implementations.

For first-order security one can use the Threshold Implementation (TI) approach of [9,15] to implement Keccak. Note that a TI Keccak internally uses three shares to obtain first-order resistance. The additional share is randomized before use (affecting the other two shares). Collapsing the result back to $d + 1$ shares also requires a refresh. For higher-order XOF, one can use higher-order TI [10] or the techniques of [5].

Domain Separation. We construct a non-secret frame header for all XOF inputs from four fixed-length components:

$$frame = (ID \parallel DS \parallel ctr \parallel IV) \tag{3}$$

- *ID*: 32-bit identifier for algorithm type, parameter set, authentication frame structure, key blob structure, WrapQ version; all serialization details.
- *DS*: 8-bit Domain Separation identifier which specifies frame purpose.
- *ctr*: A 24-bit block index $0, 1, 2, \ldots$ for encrypting multi-block material. Set to 0 for authentication (unless the authentication process is parallelized).

- IV: 256-bit Initialization Vector, chosen randomly for the key blob.

The main security property of the frame header is that it never repeats:

- For a fixed secret key, this is due to the randomization of IV. There is a birthday bound of 2^{128} wrapping operations for a given key.
- Within a key blob (fixed IV, key), thanks to (DS, ctr) uniquenuess.
- Across versions. Any functional change in the algorithm or its WrapQ serialization requires a new ID. This identifier unambiguously defines the structure of the key blob, the interpretation of the contents, the frame header, etc.

There are predefined domain separation bytes; DS_{hash} and DS_{mac} for authentication (Algorithm 1) and DS_{enc} for encryption/decryption (Algorithms 2 and 3.) Frame headers with these domain separation fields are denoted $\mathit{frame}_{\mathrm{hash}}$, $\mathit{frame}_{\mathrm{mac}}$ and $\mathit{frame}_{\mathrm{enc}}$.

Integrity. Algorithm 1 describes the authentication tag computation process. In the terminology of [8], WrapQ is an Encrypt-then-MAC (EtM) scheme; ciphertext is authenticated rather than plaintext. The authentication tag is always checked before any decryption is performed. Upon a mismatch between the calculated T' and the tag T, a FAIL is returned – no partial decrypted payload.

Since WrapQ is an Authenticated Encryption with Associated Data (AEAD) [45] scheme, A includes data items that do not need to be decrypted in addition to ciphertext C. Unambiguous serialization must be used to guarantee domain separation between data items. In our case, the ID identifier in *frame* defines the contents and ordering of fixed-length fields in A.

For performance reasons, we first use a non-masked hash function $\mathsf{Hash}()$ to process A (Step 1), and only use a masked XOF to bind the hash result h to the (masked) authentication key $[[K]]$ and other variables (Step 2.) Furthermore, randomized hashing [23] with a frame header containing the IV is used to make the security of h less dependent on strong collision resistance. The random prefix IV is included in the *frame* construction (Eq. (3)) and used again in the masked key binding step. It is domain-separated via DS from encryption/decryption frames in case the same $[[K]]$ is used. After this single masked step, $[[K]]$ is refreshed, and the authentication tag $[[T]]$ can be unmasked (collapsed) into T.

Gadgets. For first-order security, we use trivial refresh gadgets $\mathsf{Refresh}([[x]]) = (x_0 \oplus r, x_1 \oplus r)$ with $r = \mathsf{Random}()$ and $\mathsf{Encode}(x) = (x \oplus r, r)$ with $r = \mathsf{Random}()$. For quasilinear-time SNI higher-order refresh gadgets, see [32,22]. The function $\mathsf{Decode}([[x]]) = x_0 \oplus x_1 \oplus \cdots x_d = x$ simply unmasks x.

Confidentiality. We use the masked XOF in "counter mode" to encrypt/decrypt data. Algorithms 2 and 3 outline the process for a single block. SHAKE256 has a data rate of $(1600 - 2 * 256)/8 = 136$ bytes for each permutation, while SHAKE128 has a 168-byte rate. When choosing block sizes, one generally wants to minimize the number of permutation invocations.

Algorithm 1: $T = \mathsf{AuthTagM}(\ A, [[K]], ID, ctr, IV\ )$

```
Input: $A$, Authenticated data, including ciphertext (not masked.)
Input: $[[K]]$, MIK – Message Integrity Key (Boolean masked.)
Input: $ID, ctr, IV$: Used to construct frame headers (not masked.)
Output: $T$, Authentication tag (not masked.)
1: $h \leftarrow \mathsf{Hash}(\ \mathit{frame}_{\mathrm{hash}} \parallel A\ )$      ▷ Prefix containing $IV$, use unmasked hash.
2: $[[T]] \leftarrow \mathsf{XOF}_{|T|}(\ \mathit{frame}_{\mathrm{mac}} \parallel [[K]] \parallel h\ )$      ▷ Bind to the masked key.
3: $[[K]] \leftarrow \mathsf{Refresh}([[K]])$      ▷ Refresh MIK for next invocation.
4: return $T = \mathsf{Decode}([[T]])$      ▷ Collapse the authentication tag.
```

Confidentiality of C follows from the random-indistinguishability and one-wayness of the XOF function (as it would without masking), assuming that the frame identifier ($\mathit{frame}_{\mathrm{enc}}$) never repeats for the same secret key K.

A necessary feature of the block decryption (import) function (Algorithm 3) is that the ciphertext C is first converted into masked encoding (Step 1). The secret cover $[[x]]$ is also in randomized shares (Step 2). Hence decryption occurs in masked form (Step 3), avoiding collapsing $[[P]]$. This is illustrated in Fig. 1.

Algorithm 2: $C = \mathsf{EncryptBlockM}(\ [[P]], [[K]], ID, ctr, IV\ )$

```
Input: $[[P]]$, Payload block (Boolean masked.)
Input: $[[K]]$, KEK – Key Encryption Key (Boolean Masked).
Input: $id, ctr, IV$: Used to construct header $\mathit{frame}_{\mathrm{enc}}$ (not masked.)
Output: $C$, Ciphertext block (not masked).
1: $[[x]] \leftarrow \mathsf{XOF}_{|P|}(\ \mathit{frame}_{\mathrm{enc}} \parallel [[K]]\ )$      ▷ Output shares have the length of $P$.
2: $[[C]] \leftarrow [[P]]\ \oplus\ [[x]]$      ▷ $C_i = P_i \oplus x_i$, for $i = 0, 1, \cdots, d$.
3: $[[K]] \leftarrow \mathsf{Refresh}([[K]])$      ▷ Refresh KEK for next invocation.
4: $[[P]] \leftarrow \mathsf{Refresh}([[P]])$      ▷ Alternatively destroy $[[P]]$, if not reused.
5: return $C = \mathsf{Decode}([[C]])$      ▷ Masking can be removed after encryption.
```

Algorithm 3: $[[P]] = \mathsf{DecryptBlockM}(\ C, [[K]], ID, ctr, IV\ )$

```
Input: $C$, Ciphertext block (not masked).
Input: $[[K]]$, KEK – Key Encryption Key (Boolean Masked).
Input: $ID, ctr, IV$: Used to construct header $\mathit{frame}_{\mathrm{enc}}$ (not masked.)
Output: $[[P]]$, key material payload (Packed, Boolean masked.)
1: $[[C]] \leftarrow \mathsf{Encode}(C)$      ▷ Split into randomized shares.
2: $[[x]] \leftarrow \mathsf{XOF}_{|P|}(\ \mathit{frame}_{\mathrm{enc}} \parallel [[K]]\ )$      ▷ Output shares have the length of $P$.
3: $[[P]] \leftarrow [[C]] \oplus [[x]]$      ▷ $P_i = C_i \oplus x_i$, for $i = 0, 1, \cdots, d$.
4: $[[K]] \leftarrow \mathsf{Refresh}([[K]])$      ▷ Refresh KEK for next invocation.
5: return $[[P]] = \mathsf{Refresh}([[P]])$      ▷ Return masked payload.
```

2.4 Variable Classification for Kyber and Dilithium

Cryptographic module security standards (FIPS 140-3 [39] / ISO 19790 [27]) expect that implementors classify all variables based on the impact of their potential compromise.

- **CSP** (Critical Security Parameter): Security-related information whose disclosure or modification can compromise the security of a cryptographic module. CSPs require both integrity and confidentiality protection.
- **PSP** (Public Security Parameter): Security-related public information whose modification can compromise the security of a cryptographic module. PSPs require only integrity protection (authentication).
- **SSP** (Sensitive Security Parameter): Either a CSP or PSP, or a mixture of both. Essentially all variables in a cryptographic module are SSPs.

In the FIPS 140-3 / ISO 19790 context, the (non-invasive) side-channel leakage protection requirement only applies to CSPs, not PSPs. We use this more economical approach in our design. Asymmetric cryptographic algorithms have both public and secret key material. The parts of secret key material whose disclosure can compromise cryptographic security are CSPs. Additionally, all internally derived or temporary variables whose leakage will compromise security are CSPs.

The standard serialization formats of Kyber and Dilithium keys complicate this picture somewhat as secret keys duplicate data that is also contained in public keys.

CRYSTALS-Kyber. Table 1 contains a classification of Kyber key variables. WrapQ encrypts and authenticates the masked CSPs $(\mathbf{s}, z)$ and only authenticates the rest of the parameters. For the underlying MLWE problem $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$ the public key consists of $\mathbf{t}$ and the secret key is $\mathbf{s}$. In Kyber, the $\mathbf{A}$ matrix is represented by a SHAKE128 seed ρ that deterministically generates it.

The standard encoding stores $\mathbf{s}$ in the NTT-domain representation $\hat{\mathbf{s}}$. To conserve storage space and also Boolean-to-Arithmetic transformation effort, we instead store normal-domain $\mathbf{s}$, where coefficients are in the range $[-\eta, \eta]$ and would fit into 3 bits (in Kyber, we have $\eta \in \{2, 3\}$, depending on the security level.) However, WrapQ uses 4 bits per coefficient for decoding convenience.

The z variable is a secret quantity used to generate a deterministic response to an invalid ciphertext in the Fujisaki-Okamoto transform. The security proofs assume it to be secret (we implement the entire FO transform as masked); hence, this 256-bit quantity is handled as a Boolean masked secret.

The standard encoding secret key contains a full copy of the public key, likely due to the reference API [38], which did not allow the passing of the public key to signature decapsulation functions. It also contains $H(pk)$ as a performance optimization. We retain and authenticate the $H(pk)$ quantity as it can be used to authenticate the public key that is separately supplied.

Table 1. Kyber "standard serialization" public and secret key component variable classification. WrapQ uses the same encoding for the public key but allows masked implementations to access the secret variables without side-channel leakage.

Standard Encoding		**Public Key**	**Secret Key**
CRYSTALS-Kyber [3]		$pk = (\hat{\mathbf{t}}, \rho)$	$sk = (\underline{\hat{\mathbf{s}}}, pk, H(pk), \underline{z})$
Variable	**Bits**	**Description**	
$\hat{\mathbf{t}}$	$k \times 12 \times 256$	PSP: Public vector, NTT domain.	
ρ	256	PSP: Seed for public $\mathbf{A}$.	
$\hat{\mathbf{s}}$	$k \times 12 \times 256$	CSP: Secret vector, NTT domain.	
pk	$\lvert\hat{\mathbf{t}}\rvert + 256$	PSP: Full public key.	
$H(pk)$	256	PSP: Hash of the public key.	
z	256	CSP: Fujisaki-Okamoto rejection secret.	

WrapQ Secret Key Blob		**CRYSTALS-Kyber**
Field	**Bits**	**Description**
ID	32	Algorithm and serialization type identifier.
T	256	Authentication tag (Algorithm 1).
IV	256	Nonce/randomization seed.
pkh	256	Public key hash $\mathsf{SHA3}(pk)$, authenticated only.
z	256	Encrypted: FO Transform secret (Boolean).
$\mathbf{s}$	$k \times 4 \times 256$	Encrypted: Secret key polynomials (Boolean).

CRYSTALS-Dilithium. Table 2 contains a classification of Dilithium public and secret key variables. WrapQ encrypts $(K, \mathbf{s}_1, \mathbf{s}_2)$ and authenticates the rest of the parameters. For the underlying equation $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ the public key is $(\mathbf{A}, \mathbf{t})$ and the secret key is $(\mathbf{s}_1, \mathbf{s}_2)$. In Dilithium, the $\mathbf{A}$ matrix is represented with a short SHAKE128 seed ρ, and the $\mathbf{t}$ quantity is split into two halves to minimize the size of the public key, with $\mathbf{t}_1$ placed in the public key and the $\mathbf{t}_0$ in private key (as it is not required for verification). The security proofs treat the entire $\mathbf{t}$ as public, as we do.

The tr quantity is a 256-bit hash of the public key $tr = \mathsf{SHAKE256}(\rho \parallel \mathsf{t}_1)$. Only the hash is required for signature generation, but since it is an authenticated part of the key blob, we use this quantity to verify that a supplied public key matches the secret key.

The distribution of both $\mathbf{s}_1$ and $\mathbf{s}_2$ is uniform in $[-\eta, +\eta]$. Depending on the security parameters, we have $\eta \in \{2, 4\}$, resulting in 5 or 9 distinct values. While the standard encoding uses bit-packed $d_\eta = \lceil \log_2(2\eta + 1) \rceil$ bits (either 3 or 4). WrapQ uses 4 bits per coefficient in both cases.

The K variable is a secret "seed" value used in deterministic signing (when a signature should be a deterministic, non-randomized function of the private key and the message to be signed). We treat K as a Boolean-masked quantity. However, from a side-channel security perspective, it is preferable always to randomize the signing process, in which case K is not used.

Table 2. Dilithium "standard serialization" public and secret key component variable classification, and the WrapQ encoding format for secret keys

Standard Encoding	**Public Key**	**Secret Key**
CRYSTALS-Dilithium [4]	$pk = (\rho, \mathbf{t}_1)$	$sk = (\rho, \underline{K}, tr, \underline{\mathbf{s}_1}, \underline{\mathbf{s}_2}, \mathbf{t}_0)$

Variable	**Bits**	**Description**
ρ	256	PSP: Seed for public $\mathbf{A}$.
$\mathbf{t}_1$	$k \times 10 \times 256$	PSP: Upper half of public $\mathbf{t}$.
K	256	CSP: Seed for deterministic signing.
tr	256	PSP: Hash of public key $tr = H(\rho \parallel \mathbf{t}_1)$.
$\mathbf{s}_1$	$\ell \times d_\eta \times 256$	CSP: Secret vector 1, coefficients $[-\eta, \eta]$.
$\mathbf{s}_2$	$k \times d_\eta \times 256$	CSP: Secret vector 2, coefficients $[-\eta, \eta]$.
$\mathbf{t}_0$	$k \times 13 \times 256$	PSP: Lower half of public $\mathbf{t}$.

WrapQ Secret Key Blob	**CRYSTALS-Dilithium**

Field	**Bits**	**Description**
ID	32	Algorithm and serialization type identifier.
T	256	Authentication tag (Algorithm 1).
IV	256	Nonce/randomization seed.
ρ	256	Public seed for $\mathbf{A}$, authenticated only.
K	256	Encrypted: Determ. signing seed (Boolean).
tr	256	Hash $tr = \mathsf{SHAKE256}(pk)$, authenticated only.
$\mathbf{t}_0$	$k \times 13 \times 256$	Lower half of public $\mathbf{t}$, authenticated only.
$\mathbf{s}_1$	$\ell \times 4 \times 256$	Encrypted: Secret vector 1 (Boolean).
$\mathbf{s}_2$	$k \times 4 \times 256$	Encrypted: Secret vector 2 (Boolean).

2.5 Encoding Details

The proposal is entirely built from SHA3 / Keccak components; the $\mathsf{XOF}()$ masked permutation (Definition 2) and its non-masked counterpart $\mathsf{Hash}()$ (Section 2.3). We note that the masked permutation function is substantially more complex than the non-masked version; a straightforward hardware implementation of a first-order threshold implementation is roughly three times larger [9] than the unmasked one, and the complexity grows quadratically with the masking order [5]. Other operations in the process are related to mask refreshing or trivial ones such as linear XORs, packing of bits, etc.

Algorithm 1 requires $\lceil (|frame| + |A| + |\text{padding}|)/r \rceil$ unmasked Keccak permutations to compute h with $\mathsf{Hash}()$, where r is the block rate. For SHAKE256, we have $r = 136$ bytes. Additionally, there is a single invocation of masked $\mathsf{XOF}()$ permutation to compute $[[T]]$.

Algorithms 2 and 3 require $\lceil |P|/r \rceil$ invocations of the masked permutation in $\mathsf{XOF}()$. This is also the minimum when computation is organized in a "counter mode" fashion where $[[P]]$ is split into block-sized chunks and ctr is used as an input index. In a way, the encryption process is "format-preserving" as the block size is arbitrary. It is not economical to encrypt blocks substantially smaller than r, as that will result in an increased number of permutations. However, for some parameters, we sacrifice optimality for the logical separation of data items, simplifying implementation.

Wrapping Process. In the key wrapping operation WrapQ (Eq. (1)) all CSPs are converted to Boolean shares. For arithmetic $\mathbf{s}$ shares, this involves Inverse-NTT operations since 4-bit packing is used, followed by an Arithmetic-to-Boolean conversion. We can then choose a random IV. Then the $[[P]]$ input, comprising of CSP data, is divided into blocks for Algorithm 2. For Dilithium and Kyber, we process one polynomial at a time since the resulting $(4 \times 256)/8 = 128$-byte block fits the 136-byte data rate of SHAKE256. This has the advantage of "random access" – each secret polynomial can be decrypted only when needed, reducing the RAM requirement. The 4-bit encoding is not optimal of all $[-\eta, +\eta]$ ranges present in these algorithms but is simpler to decode. The Boolean CSPs (K or z) have $ctr = 0$, block and polynomial CSPs are $1 \leq ctr \leq k$ with Kyber and $1 \leq ctr \leq k + \ell$ with Dilithium. The ciphertext blocks and the PSP data items are then combined into blob A; its serialization is the same as in Tables 1 and 2, although ID, T, IV are omitted from A. Finally, A is passed to Algorithm 1 to produce T; then the final WrapQ key blob is combined from (ID, T, IV, A).

Unwrapping Process. The unwrapping operation WrapQ^{-1} (Eq. (2) starts with consistency checks; we parse ID from the beginning of the blob and see if the size of the blob matches with it. We also check that the pkh (Kyber) or tr (Dilithium) fields match with a hash of the public key that is separately provided. We then extract IV and A (the remaining part after IV in the blob), and pass those to Algorithm 1 to obtain a check value T'. If we have a mismatch $T \neq T'$ then we return FAIL and abort. After this, we proceed to decrypt CSP fields into payload shares $[[P]]$ using Algorithm 3 and selecting ctr as in the wrapping process. The conversion of arithmetic CSPs follows an inverse route; Boolean-to-Arithmetic conversion, perhaps followed by an NTT transform.

Size Metrics. The NIST standardization process will likely bring some changes to Kyber 3.02 [3] and Dilithium 3.1 [4]. Furthermore, WrapQ is not necessarily an "interchange" key format; details are subject to change from one instantiation to another. Table Table 3 summarizes the sizes of both standard encodings for Kyber and Dilithium keypairs. We observe that even a single randomized arithmetic CSP share would be larger than the WrapQ format. For several parameter sizes, the WrapQ size could be further reduced by encoding the $[-\eta, +\eta]$ coefficients in less than 4 bits, but this would complicate implementation somewhat.

3 Compressed Masking

If we can "write back" and refresh secret keys or other sensitive variables every time they are used, using non-masked seed expanders becomes possible. While information theory implies that at least one "base" share x_0 must have complete

Table 3. Kyber 3.02 [3] and Dilithium 3.1 [4] "standard serialization" public and secret key sizes in bytes, and the size of the WrapQ secure secret key blob (Tables 1 and 2.) Note that a Kyber WrapQ structure requires a standard-format public key to be separately provided. Also included is the size of a single (packed) masked arithmetic share; $k \times 12 \times 256$ bits for Kyber, $(k + \ell) \times 23 \times 256$ bits for Dilithium.

Algorithm			**Standard Encoding**		**Masking** per	**WrapQ** Safe
Parameters	k	ℓ	Pub. Key	Priv. Key	Arith. Share	Private Key
Kyber512	2		800	1,632	768	**388**
Kyber768	3		1,184	2,400	1,152	**516**
Kyber1024	4		1,568	3,168	1,536	**644**
Dilithium2	4	4	1,312	2,528	5,888	**2,852**
Dilithium3	6	5	1,952	4,000	8,096	**4,068**
Dilithium5	8	7	2,592	4,864	11,040	**5,412**

encoding, each additional share $x_i, i > 0$ is represented by a temporary key z_i. We introduce a symmetric cryptography primitive $\mathsf{Sample}_G(z)$ that maps finite-length keys z to elements in G. This function itself does *not* need to be masked; input or output variables are mask components, not masked variables.

Definition 3. *The function* $x \leftarrow \mathsf{Sample}_G(z)$ *uses the input seed* $z \in \{0,1\}^k$ *to deterministically sample an element* $x \in G$. *The function is cryptographically secure; the computational task of distinguishing* x *from a random element in set* G *is not substantially easier than an exhaustive search (advantage* $< 2^{-k+1}$*).*

Practical instantiation. We implement $\mathsf{Sample}_G(z)$ with an extendable output function (XOF) such as SHAKE 128/256 [37]. The function could also be instantiated with a stream cipher or a block cipher (in counter mode). If a mapping from XOF output to non-binary distributions is required, one may use rejection sampling since each $\mathsf{XOF}(z)$ defines an arbitrarily long bit sequence.

Examples of sampled $|G| \gg 2^k$ include large-degree polynomials that are ring elements $\mathbb{Z}_q[x]/(x^n + 1)$ in Kyber [3] and Dilithium [4]. Note that implementations Kyber and Dilithium already have subroutines that generate uniform polynomial coefficients in $\mathbb{Z}/q\mathbb{Z}$ from XOF output via rejection sampling. An efficient (unmasked) method for this task is required to create $\mathbf{A}$ polynomials on the fly, so we implemented the seed expander directly in the hardware.

Discussion. The shares x are relatively long (polynomials of thousands of bits) in the lattice cryptography use case; hence, an instantiation of a standard, secure cryptographic algorithm as a sampler is possible. Works such as [13,25] concern the probing of local wires and reducing the number of random bits required to construct a much smaller masked circuit such as AES.

Definition 4. *Compressed masking consists of a tuple* $[[x]]^z = (x_z, z_1, \cdots, z_d)$ *satisfying* $x \equiv x_z + \sum_{i=1}^{d} \mathsf{Sample}_G(z_i)$ *with* $x_z \in G$ *and* $z_i \in \{0,1\}^k$ *for* $i \in [1,d]$.

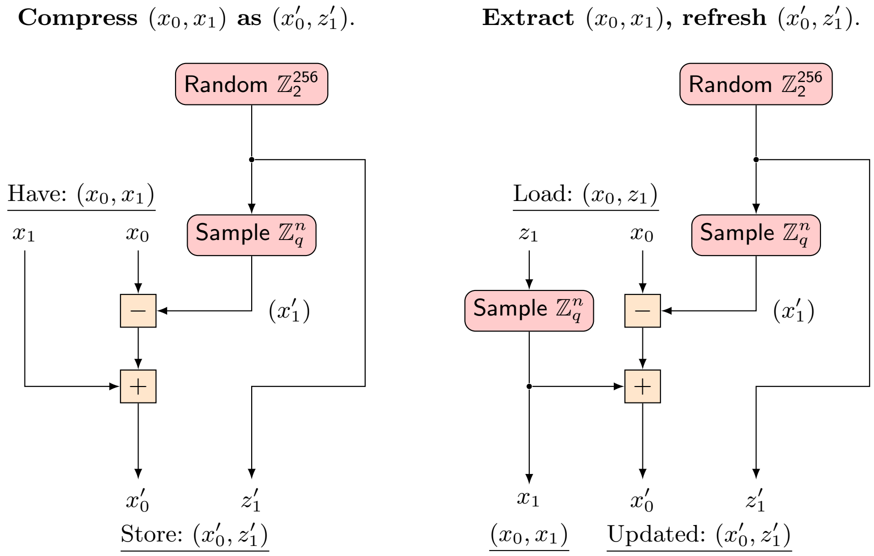

Fig. 2. Illustrating first-order mask compression. Let $[[x]] = (x_0, x_1)$ consist of a pair of degree-n polynomials ($n = 256$ for Kyber, Dilithium) with integer coefficients $\in \mathbb{Z}_q$. Function $\mathsf{Sample}_{\mathbb{Z}_q^n}(z)$ takes a 256-bit key z and uniformly samples a polynomial from it (similarly to $\mathsf{ExpandA}(z)$ in Dilithium and $\mathsf{Parse}(\mathsf{XOF}(z))$ in Kyber.) On the left-hand side, a "compression" gadget (analogous to Algorithm 4) creates a 256-bit random z'_1 and samples a random polynomial x'_1 using it. It then subtracts x'_1 from x_0 and then adds x_1 to the result, producing x'_0. This construction is exactly like a trivial first-order refresh gadget, except that instead of (x'_0, x'_1) we store (x'_0, z'_1), which has a significantly smaller since z'_1 is only 256 bits. While $x'_1 \leftarrow \mathsf{Sample}_{\mathbb{Z}_q^n}(z'_1)$ would suffice for decompression (once), on the right-hand side, we present a simultaneous refresh mechanism (analogous to Algorithm 5) that allows repeated extractions.

Theorem 1. *It is computationally infeasible to determine information about x from any subset of d elements in compressed masking tuple $[[x]]^z$.*

Proof. If x_z is not known, x can be any value. If one of z_i is unavailable, the indistinguishability property of $\mathsf{Sample}_G(z_i)$ makes x similarly indistinguishable.

Encoding Size. From Theorem 1 we observe that the compressed masking inherits the basic security properties of regular masked encoding. However, the size of the representation is only $\log_2 |G| + dk$ bits, while a regular representation requires $(d+1)\log_2 |G|$ bits. In the case of Kyber, polynomials are typically packed in $12 * 256 = 3072$ bits, while Dilithium ring elements require $23 * 256 = 5888$ bits. In compressed masking, this is the size of the x_0^z element only, while z_i variables are $k = 256$ bits.

Algorithm 4: $[[\mathrm{x}]]^z = \mathsf{MaskCompress}([[x]])$ *(NI / Not composable.)*

```
Input: Masking $[[x]] = (x_0, x_1, \cdots, x_d)$.
Output: Compressed masking $[[x]]^z$ with $x_0^z + \sum_{i=1}^{d} \mathsf{Sample}_G(z_i) = \sum_{i=0}^{d} x_i$.
$x_0^z = x_0$
for $i = 1, 2, \cdots, d$ do
    $z_i \leftarrow \mathsf{Random}(k)$        ▷ Random Bit Generator, $k$ bits.
    $x_0^z \leftarrow x_0^z - \mathsf{Sample}_G(z_i)$
    $x_0^z \leftarrow x_0^z + x_i$
return $[[x]]^z = (x_0^z, z_1, z_2, \cdots, z_d)$
```

Conversions. We obtain a trivial mapping from compressed encoding $[[x]]^z$ to the general masked encoding $[[x]]$ by setting $x_0 = x_0^z$ and $x_i = \mathsf{Sample}_G(z_i)$ for $i \in [1, d]$. Security follows from the observation that this conversion is "linear" in the sense that there is no interaction between shares.

Mapping from regular to compressed format requires interaction between the shares since Sample_G is not invertible. Algorithm 4 MaskCompress presents one way of performing this conversion. We note its resemblance to the RefreshMasks gadget of Rivain and Prouff ([44, Algorithm 4]); its NI security follows similarly. While it is secure if used appropriately, combining it with other gadgets may expose leakage, as demonstrated in [14]. Depending on requirements, it can be combined with additional refresh steps to build an SNI [5] gadget.

Algorithm 5: $x_i = \mathsf{LoadShare}([[x]]^z, i)$ *(NI / Not composable.)*

```
Input: Compressed masking $[[x]]^z$ satisfying $x = x_0^z + \sum_{i=1}^{d} \mathsf{Sample}_G(z_i)$
Input: Index $i$ for the share to be accessed.
Output: If read in order, $i = 0, 1, \cdots d$, the returned $\{x_i\}$ is a fresh masking $[[x]]$.
if $i = 0$ then
    $x_i \leftarrow x_0^z$        ▷ Should be accessed first, the rest $i > 0$ only once.
else
    $x_i \leftarrow \mathsf{Sample}_G(z_i)$        ▷ Expand the current $z_i$.
    $z_i \leftarrow \mathsf{Random}(k)$        ▷ Update $z_i$ with a Random Bit Generator.
    $x_0^z \leftarrow x_0^z - \mathsf{Sample}_G(z_i)$
    $x_0^z \leftarrow x_0^z + x_i$
return $x_i$
```

Computing with Compressed Masking A key observation for memory conservation is that one does not need to uncompress all of the shares to perform computations with the compressed masked representation. One can decompress a single share, perform a transformation on it, compress it, and proceed to the next one. Masked lattice cryptography implementations generally operate sequentially on each share, performing complex linear operations such as Number

Theoretic Transforms (NTT) on individual shares without interaction with others. Furthermore, they require individual masked secret key shares only once (or a limited number of times) during a private key operation.

Algorithm 5, $\mathsf{LoadShare}([[x]]^z, i)$ decodes a share $x_i \in G$ from a compressed masking $[[x]]^z$. If the shares are accessed in the sequence $i = 0, 1, 2, \cdots, d$, it is easy to show that their sum will satisfy $x = \sum_{i=0}^{d} x_i$. The compressed masking is refreshed simultaneously (albeit not necessarily in an SNI-composable manner). Subsequent accesses to the same indices will return a different encoding $[[x]]'$.

4 Conclusions and Open Problems

When building side-channel secure implementations of asymmetric algorithms, it is easy to sidestep the key management problem. Academic works have generally focused on protecting the private key operations, assuming that refreshed key shares can be kept in working memory. However, many real-life devices do not have the option of having refreshable non-volatile memory for keys.

WrapQ is a method for transferring masked secret key material between a hardware security module and potentially untrusted storage. The encryption, decryption, and authentication modes can manage wrapped key material in masked format, significantly increasing their resistance to side-channel attacks.

We discuss the initial version of WrapQ, which supports the CRYSTALS-Kyber 3.02 Key Encapsulation Mechanism and CRYSTALS-Dilithium 3.1 signature scheme. The implementation leverages a masked implementation of FIPS 202 / SHAKE256 (the Keccak permutation) in a mode that prevents leakage even when an attacker can acquire thousands of side-channel measurements from importing and exporting secret keys and also access the resulting WrapQ data itself. We have performed a leakage assessment validation of its implementation for Kyber and Dilithium concerning the protected CSP variables and the encryption key with a TVLA testing of up to 100K traces.

When working with trusted temporary storage, *compressed masking* (Section 3) can be used to reduce the amount of secure memory required. This simple technique allows a set of d-order mask shares to have a storage requirement equivalent to a single share and d symmetric keys. In practice, this almost halves the memory requirement for first-order Kyber and Dilithium, and its benefits are more significant if higher-order masking is required.

Our experimental work has focused on first-order protections. However, the file format works also with higher-order masking. As the masking order grows, so does the complexity of all nonlinear operations and refresh gadgets. We acknowledge that the construction of higher-order gadgets for both WrapQ (Section 2) and compressed masking (Section 3) requires further investigation.

Acknowledgements The author wishes to thank Ben Marshall for running the leakage assessment tests and Oussama Danba and Kevin Law for helping to make the FPGA test target operational. Further thanks to Thomas Prest, Rafael del Pino, and Melissa Rossi for the technical and theoretical discussions. The author is to blame for all errors and omissions.

Bibliography

1. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D.: Status report on the third round of the NIST post-quantum cryptography standardization process. Interagency or Internal Report NISTIR 8413, National Institute of Standards and Technology (July 2022). https://doi.org/10.6028/NIST.IR.8413
2. Alioto, M., Bongiovanni, S., Djukanovic, M., Scotti, G., Trifiletti, A.: Effectiveness of leakage power analysis attacks on DPA-resistant logic styles under process variations. IEEE Transactions on Circuits and Systems I: Regular Papers **61**(2), 429–442 (2014). https://doi.org/10.1109/TCSI.2013.2278350
3. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-kyber: Algorithm specifications and supporting documentation (version 3.02). NIST PQC Project, 3rd Round Submission Update (August 2021), https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf
4. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-dilithium: Algorithm specifications and supporting documentation (version 3.1). NIST PQC Project, 3rd Round Submission Update (February 2021), https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf
5. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P., Grégoire, B., Strub, P., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 116–129. ACM (2016). https://doi.org/10.1145/2976749.2978427, http://dl.acm.org/citation.cfm?id=2976749
6. Barthe, G., Belaïd, S., Espitau, T., Fouque, P., Grégoire, B., Rossi, M., Tibouchi, M.: Masking the GLP lattice-based signature scheme at any order. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II. Lecture Notes in Computer Science, vol. 10821, pp. 354–384. Springer (2018). https://doi.org/10.1007/978-3-319-78375-8_12, https://eprint.iacr.org/2018/381
7. Becker, G., Cooper, J., DeMulder, E., Goodwill, G., Jaffe, J., Kenworthy, G., Kouzminov, T., Leiserson, A., Marson, M., Rohatgi, P., Saab, S.: Test vector leakage assessment (TVLA) methodology in practice (2013), presented at International Cryptography Module Conference – ICMC 2013
8. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. J. Cryptol. **21**(4), 469–491 (2008). https://doi.org/10.1007/s00145-008-9026-x
9. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Building power analysis resistant implementations of Keccak (August 2010), https://csrc.nist.gov/Events/2010/The-Second-SHA-3-Candidate-Conference

10. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Higher-order threshold implementations. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 326–343. Springer (2014). https://doi.org/10.1007/978-3-662-45608-8_18
11. Bos, J.W., Gourjon, M., Renes, J., Schneider, T., van Vredendaal, C.: Masking kyber: First- and higher-order implementations. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 173–214 (2021). https://doi.org/10.46586/tches.v2021.i4.173-214
12. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener [50], pp. 398–412. https://doi.org/10.1007/3-540-48405-1_26
13. Coron, J., Greuet, A., Zeitoun, R.: Side-channel masking with pseudo-random generator. In: Canteaut, A., Ishai, Y. (eds.) Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12106, pp. 342–375. Springer (2020). https://doi.org/10.1007/978-3-030-45727-3_12, https://eprint.iacr.org/2019/1106
14. Coron, J., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. In: Moriai, S. (ed.) Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8424, pp. 410–424. Springer (2013). https://doi.org/10.1007/978-3-662-43933-3_21
15. Daemen, J.: Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10529, pp. 137–153. Springer (2017). https://doi.org/10.1007/978-3-319-66787-4_7
16. Ding, A.A., Zhang, L., Durvaux, F., Standaert, F., Fei, Y.: Towards sound and optimal leakage detection procedure. In: Eisenbarth, T., Teglia, Y. (eds.) Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10728, pp. 105–122. Springer (2017). https://doi.org/10.1007/978-3-319-75208-2_7
17. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2. Submission to NIST (Lightweight Cryptography Project) (May 2021), https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf
18. Duc, A., Dziembowski, S., Faust, S.: Unifying leakage models: From probing attacks to noisy leakage. J. Cryptol. **32**(1), 151–177 (2019). https://doi.org/10.1007/s00145-018-9284-1
19. Dworkin, M.: Recommendation for block cipher modes of operation: Methods for key wrapping. NIST Special Publication SP 800-38F (December 2012). https://doi.org/10.6028/NIST.SP.800-38F
20. Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for sidechannel resistance validation. CMVP & AIST Non-Invasive Attack Testing Workshop

(NIAT 2011) (September 2011), https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf

21. Goubin, L.: A sound method for switching between boolean and arithmetic masking. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2162, pp. 3–15. Springer (2001). https://doi.org/10.1007/3-540-44709-1_2
22. Goudarzi, D., Prest, T., Rivain, M., Vergnaud, D.: Probing security through input-output separation and revisited quasilinear masking. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(3), 599–640 (2021). https://doi.org/10.46586/tches.v2021.i3.599-640
23. Halevi, S., Krawczyk, H.: Strengthening digital signatures via randomized hashing. In: Dwork, C. (ed.) Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4117, pp. 41–59. Springer (2006). https://doi.org/10.1007/11818175_3
24. Heinz, D., Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels, D.: First-order masked Kyber on ARM Cortex-M4. IACR ePrint 2022/058 (2022), https://eprint.iacr.org/2022/058
25. Ishai, Y., Kushilevitz, E., Li, X., Ostrovsky, R., Prabhakaran, M., Sahai, A., Zuckerman, D.: Robust pseudorandom generators. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D. (eds.) Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7965, pp. 576–588. Springer (2013). https://doi.org/10.1007/978-3-642-39206-1_49, https://eprint.iacr.org/2013/671
26. Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2729, pp. 463–481. Springer (2003). https://doi.org/10.1007/978-3-540-45146-4_27
27. ISO: Information technology – security techniques – security requirements for cryptographic modules. Standard ISO/IEC 19790:2012(E), International Organization for Standardization (2012)
28. ISO: Information technology – security techniques – testing methods for the mitigation of non-invasive attack classes against cryptographic modules. Standard ISO/IEC 17825:2016, International Organization for Standardization (2016), https://www.iso.org/standard/82422.html
29. ISO: Information technology – security techniques – testing methods for the mitigation of non-invasive attack classes against cryptographic modules. Working Draft ISO/IEC WD 17825:2021(E), International Organization for Standardization (2021)
30. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996). https://doi.org/10.1007/3-540-68697-5_9
31. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener [50], pp. 388–397. https://doi.org/10.1007/3-540-48405-1_25

32. Mathieu-Mahias, A.: Securisation of implementations of cryptographic algorithms in the context of embedded systems. Ph.D. thesis, Université Paris-Saclay (2021), https://tel.archives-ouvertes.fr/tel-03537322
33. Menhorn, N.: External secure storage using the PUF. Application Note: Zynq UltraScale+ Devices, XAPP1333 (v1.2) (April 2022), https://docs.xilinx.com/r/en-US/xapp1333-external-storage-puf
34. Microsoft: Bring your own key specification. Online documentation: Azure Key Vault / Microsoft Learn. Accessed 2022-Oct-12 (February 2022), https://learn.microsoft.com/en-us/azure/key-vault/keys/byok-specification
35. Migliore, V., Gérard, B., Tibouchi, M., Fouque, P.: Masking dilithium - efficient implementation and side-channel evaluation. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11464, pp. 344–362. Springer (2019). https://doi.org/10.1007/978-3-030-21568-2_17
36. Moriarty, K.M., Nystrom, M., Parkinson, S., Rusch, A., Scott, M.: PKCS #12: Personal information exchange syntax v1.1. IETF RFC 7292 (July 2014). https://doi.org/10.17487/RFC7292
37. NIST: SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication FIPS 202 (August 2015). https://doi.org/10.6028/NIST.FIPS.202
38. NIST: PQC – API notes. Example Files, Official Call for Proposals, National Institute for Standards and Technology (September 2017), https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/example-files/api-notes.pdf
39. NIST: Security requirements for cryptographic modules. Federal Information Processing Standards Publication FIPS 140-3 (March 2019). https://doi.org/10.6028/NIST.FIPS.140-3
40. NSA: Announcing the commercial national security algorithm suite 2.0. National Security Agency, Cybersecurity Advisory (September 2022), https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF
41. Prouff, E., Rivain, M.: Masking against side-channel attacks: A formal security proof. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7881, pp. 142–159. Springer (2013). https://doi.org/10.1007/978-3-642-38348-9_9
42. Quisquater, J., Samyde, D.: Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In: Attali, I., Jensen, T.P. (eds.) Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2140, pp. 200–210. Springer (2001). https://doi.org/10.1007/3-540-45418-7_17
43. Rambus: Test vector leakage assessment (TVLA) derived test requirements (DTR) with AES. Rambus CRI Technical Note (February 2015), https://www.rambus.com/wp-content/uploads/2015/08/TVLA-DTR-with-AES.pdf
44. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Mangard, S., Standaert, F. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010.

Proceedings. Lecture Notes in Computer Science, vol. 6225, pp. 413–427. Springer (2010). https://doi.org/10.1007/978-3-642-15031-9_28

45. Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002. pp. 98–107. ACM (2002). https://doi.org/10.1145/586110.586125, http://dl.acm.org/citation.cfm?id=586110
46. Rogaway, P., Shrimpton, T.: A provable-security treatment of the key-wrap problem. In: Vaudenay, S. (ed.) Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4004, pp. 373–390. Springer (2006). https://doi.org/10.1007/11761679_23
47. Saarinen, M.J.O.: WiP: Applicability of ISO standard side-channel leakage tests to NIST post-quantum cryptography. In: IEEE International Symposium on Hardware Oriented Security and Trust (HOST). June 27–30, 2022 Washington DC, USA. pp. 69–72. IEEE (2022). https://doi.org/10.1109/HOST54066.2022.9839849, https://eprint.iacr.org/2022/229
48. Schneider, T., Moradi, A.: Leakage assessment methodology - A clear roadmap for side-channel evaluations. In: Güneysu, T., Handschuh, H. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9293, pp. 495–513. Springer (2015). https://doi.org/10.1007/978-3-662-48324-4_25
49. Whitnall, C., Oswald, E.: A critical analysis of ISO 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11923, pp. 256–284. Springer (2019). https://doi.org/10.1007/978-3-030-34618-8_9
50. Wiener, M.J. (ed.): Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings, Lecture Notes in Computer Science, vol. 1666. Springer (1999). https://doi.org/10.1007/3-540-48405-1

A Implementation and Leakage Assessment Experiments

WrapQ grew out of a need to be able to manage Kyber and Dilithium private keys in a commercial side-channel secure hardware module. For leakage testing, the hardware platform was instantiated on an FPGA target.

Additionally, a simple conversion program was written in Python (that used library Keccak functions rather than masked ones) for interoperability testing.

A.1 FPGA Platform Overview

A first-order implementation of the WrapQ mechanism was tested with an FPGA module that also implements first-order masked Dilithium and Kyber. We outline its relevant components.

- A 64-bit RISC-V control processor.
- Lattice accelerator that can support Kyber and Dilithium $\mathbb{Z}_q$ and NTT ring arithmetic. The unit can also perform vectorized bit manipulation operations for tasks such as masking conversions (A2B, B2A).
- Ascon-based [17] mask generator used by the lattice unit for refreshing Boolean and Arithmetic $(\bmod\ q)$ shares. This RNG is for masking only, not for key generation. It can be continuously seeded from an entropy source.
- Fast (unmasked) 1600-bit Keccak permutation used for A public value processing, and also to compute PSP hashes (e.g., value h in Algorithm 1).
- A compact first-order, three-share Threshold Implementation [9,15] of the Keccak. See discussion in Section 2.3.

A.2 Algorithmic Overview

The implementation supported all main versions of Kyber and Dilithium (Table 3). In the internal representation, the algorithms hold two copies of the variables in Tables 1 and 2 either in compressed or uncompressed format. Kyber polynomials were decoded into 16 bits per coefficient for arithmetic operations, while Dilithium polynomials used 32 bits. Hence a two-share unpacked Kyber1024 $[[\mathbf{s}]]$ requires 4 kB of internal storage while Dilithium5 $([[\mathbf{s}_1]], [[\mathbf{s}_2]])$ needs 30 kB. These polynomials are handled using $(\bmod\ q)$ arithmetic masking. The 256-bit quantities z (Kyber) and K (Dilithium) were Boolean masked.

Key generation for Kyber (CBD functions) and Dilithium (small-range uniform rejection sampling) was implemented in the bitwise Boolean-masked domain. Hence the wrapping operation does not necessarily require an Arithmetic-to-Boolean (A2B) transform – if invoked from the key generation. The implementation had an A2B function for this case, however.

First-order Boolean-to-Arithmetic (B2A) transform in the unwrapping function was based on the efficient method of Goubin [21], with additional masked manipulation steps to transform secret key quantities from $(\bmod\ 2^n)$ arithmetic masking to $(\bmod q)$ arithmetic masking.

Conversion from the two-share representation to the three-share input required for the TI Keccak was done with the help of the fast masking random generator. Two random vectors were used. Conversion into another direction involves collapsing two of the three shares together (keeping the third intact) and then refreshing the result with a single random vector.

The confidentiality algorithm used in the test target matches the details of Algorithms 2 and 3 in Section Section 2. Authentication was enabled in the import and export functions, but the tests were performed using a “platform security” parameterization; 128-bit IV and T fields, and a slightly different arrangement of hashes is Algorithm 1. Masking compression (Section 3) was implemented and tested in the firmware but was not within the “trigger” of the import and export TVLA tests presented here.

A.3 Leakage Assessment: Fixed-vs-Random Tests

Our methodology broadly follows the ISO/IEC WD 17825:2021(E) "General Testing Procedure," [29, Figure 7] with statistical corrections [16,49]. This, in turn, was based on Test Vector Leakage Assessment (TVLA) proposed by CRI / Rambus in 2011 [20] and refined in subsequent works [48,16].

In TVLA testing, two sets of trace waveforms, A and B, each with L synchronized time points, are compared. The tests we use are of the "Fixed-vs-Random" type, where set A has a CSP (such as a secret key set) to a fixed value, while set B has that CSP set to random values. Welch's t-test is applied to each time point to see if the averages of A and B sets differ significantly. A significant difference is a distinguishing feature between the sets, implying leakage from the CSP.

The first step is determining the required sample size $N = N_A + N_B$ and t-test threshold C from the experiment parameters.

α: Significance level (probability) of false positives / type I error.
β: Significance level (probability) of false negatives / type II error.

Traditionally ([7,43,28]) a critical value C of ± 4.5 has been used for $L = 1$, which matches an $\alpha < 10^{-5}$ in that case. Since we have long traces (large L), this choice would cause false positives. We adjust the critical value C based on L using the Mini-p procedure from Zhang et al. [16]. Let $\alpha_L = 1 - (1 - \alpha)^{(1/L)}$ be the adjusted significance level. Since the degrees of freedom are very large, we can approximate using the normal distribution: $C = \mathsf{CDF}^{-1}(1 - \frac{\alpha_L}{2})$.

Older (2016) versions of ISO 17825 [28] set the number of traces to $N = 10,000$ at FIPS 140-3 security level 3 and $N = 100,000$ at the highest level 4. Newer draft versions [29] derive N using an experimentally-derived Cohen's statistical effect size d, an approach suggested in [49]. The $d \in \{0.01, 0.04\}$ values were based on AES key recovery experiments. We don't have experimental key-extraction data to justify a specific d selection, so we collect as many traces as practically possible (in a day or two.) We adopt a best-effort approach to leakage detection and attempt to minimize physical or methodological sources of interference that might negatively impact the leakage assessment process.

1. Collect Subsets A and B and compute their pointwise averages (μ_A, μ_B) and standard deviations (σ_A, σ_B).
2. Compute the pointwise Welch t-test statistic vector
$$T = \frac{\mu_A - \mu_B}{\sqrt{\frac{\sigma_A^2}{N_A} + \frac{\sigma_B^2}{N_B}}}.$$
3. If at any point $|T| > C$, the test results in a FAIL. If the threshold is not crossed, the test is a PASS.

KEK Leakage Testing. A (relatively) straightforward fixed-vs-random test is used in relation to the symmetric Key Encryption Key (KEK) K. This 256-bit secret variable is used for decryption in import (Algorithm 3), encryption in export (Algorithm 3), and to compute integrity check values (Algorithm 1) in both cases.

The test aims to find leakage from the key K itself, and its set-up is similar to “fixed-vs-random key” TVLA tests performed on block ciphers such as AES [29,43]. Set A has a fixed K, while set B has a random K. Note that the plaintext payload data (i.e., Kyber and Dilithium keys) is randomized in this test; only the symmetric keys are manipulated.

CSP Leakage Testing. For fixed-vs-random testing, confidentiality (encryption) is only provided in WrapQ for CSP (actually non-public) variables. Kyber has two CSPs: ring vector $\mathbf{s}$ (decryption key), and FO secret z (Table 1) while Dilithium’s CSPs are the ring vectors $\mathbf{s}_1, \mathbf{s}_2$ (signing key) and the deterministic seed K (Table 2.) All other variables are PSPs (public.)

Since public components are not protected, they would, of course “leak.” In order to capture leakage from these specific CSP variables during the import/export function, we construct *synthetic keys* [47] where CSPs have been randomized, but other components (such as public seed ρ) are unmodified. Such masked keys $[[P]]$ would not be valid for encapsulation/decapsulation or signing/verifying; they are merely artifacts used in side-channel testing of key import and export functions.

Trace Acquisition. The experiments were performed with XC7A100T2FTG256 Artix 7 FPGA chip on a ChipWhisperer CW305-A100 board, clocked at 50 Mhz. The processor and coprocessor bitstreams were synthesized with Xilinx Vivado 2021.2. All firmware was in C and complied GCC, under `-Os` size optimization and `-mabi=lp64 -march=rv64imac` architectural flags.

Signal acquisition was performed with Picoscope 6434E oscilloscopes with a 156.25 MHz sampling rate connected to the SMA connectors on the CW305 board. The DUT generated a cycle-precise trigger.

Table 4 summarizes the various Fixed-vs-Random tests performed on the implementation. The tests were carried out on all three proposed security levels of Kyber and Dilithium, but we only include graphs for the highest Category 5 proposals, Kyber1024 and Dilithium5.

The functions passed the tests with 100,000 traces. Even though the critical value C has been adjusted for long traces (as discussed above), by looking at the figures referenced in Table 4, we can see that the t values are generally bound at a much smaller range. The target unit also performs side-channel secure Kyber and Dilithium operations, but those tests are out of scope for the present work.

Table 4. Summary of Random-vs-Fixed tests on WrapQ key import and export functions. The tests were designed to test leakage from the KEK (key) and the payload CSPs. Each test consisted of 100K traces.

Func Under Test	Set A	Set B	Both A&B	Traces
Kyber Import	Fixed CSP	Random CSP	Fixed KEK	Fig. 3
Kyber Export	Fixed CSP	Random CSP	Fixed KEK	Fig. 4
Kyber Import	Fixed KEK	Random KEK	Random CSP	Fig. 5
Kyber Export	Fixed KEK	Random KEK	Random CSP	Fig. 6
Dilithium Import	Fixed CSP	Random CSP	Fixed KEK	Fig. 7
Dilithium Export	Fixed CSP	Random CSP	Fixed KEK	Fig. 8
Dilithium Import	Fixed KEK	Random KEK	Random CSP	Fig. 9
Dilithium Export	Fixed KEK	Random KEK	Random CSP	Fig. 10

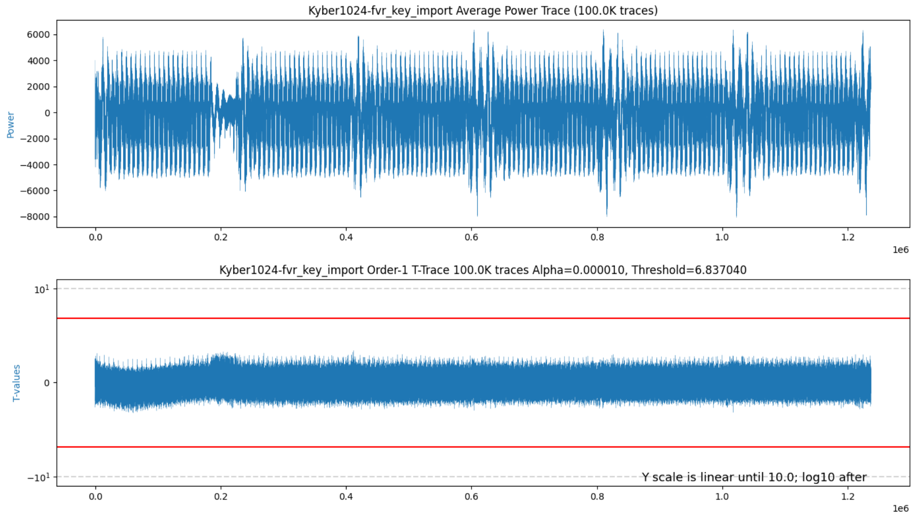

Fig. 3. Kyber1024 WrapQ Import Random-vs-Fixed CSP, 100K Traces.

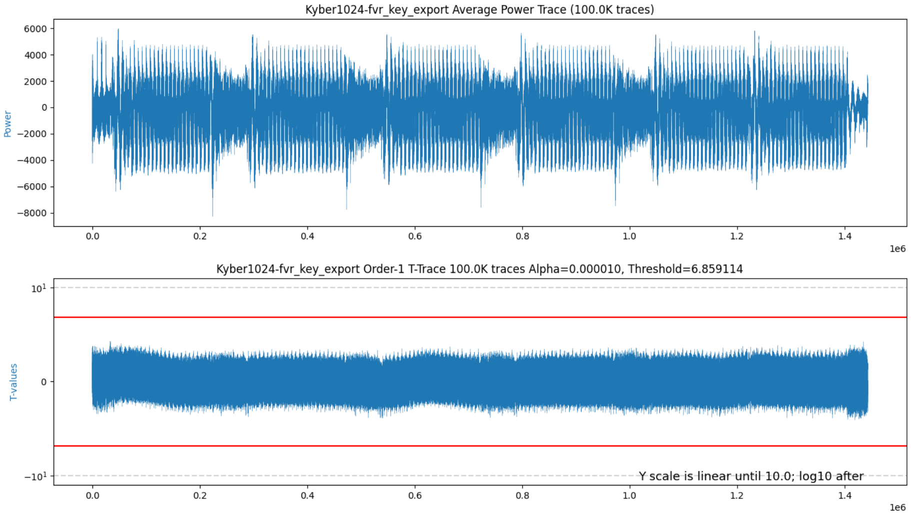

Fig. 4. Kyber1024 WrapQ Export Random-vs-Fixed CSP, 100K Traces.

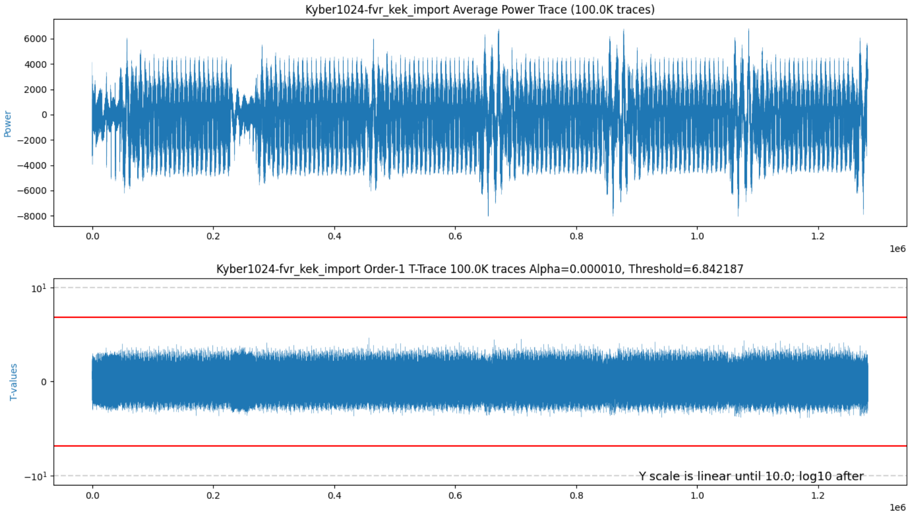

Fig. 5. Kyber1024 WrapQ Import Random-vs-Fixed KEK, 100K Traces.

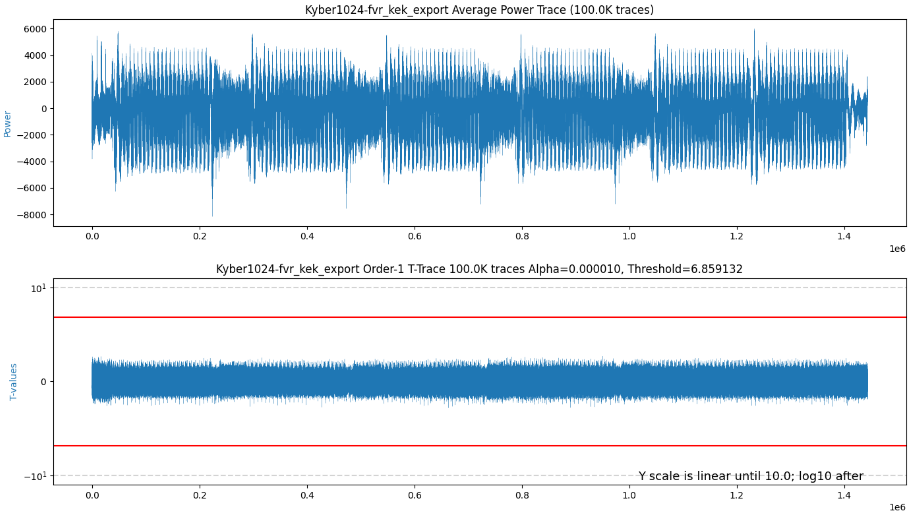

Fig. 6. Kyber1024 WrapQ Export Random-vs-Fixed KEK, 100K Traces.

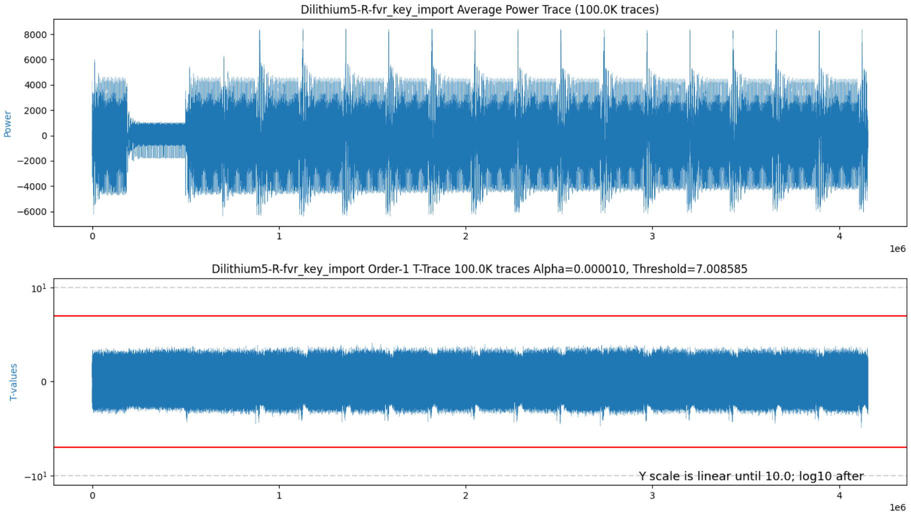

Fig. 7. Dilithium5 WrapQ Import Random-vs-Fixed CSP, 100K Traces.

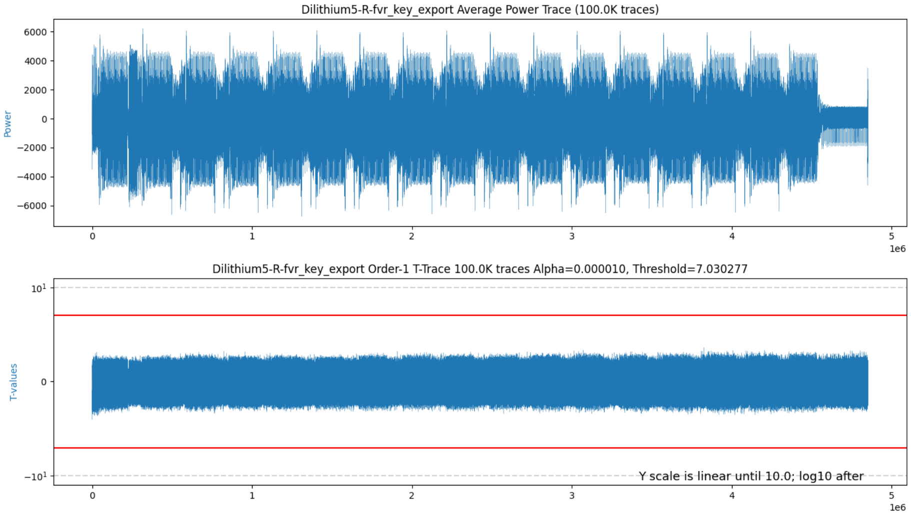

Fig. 8. Dilithium5 WrapQ Export Random-vs-Fixed CSP, 100K Traces.

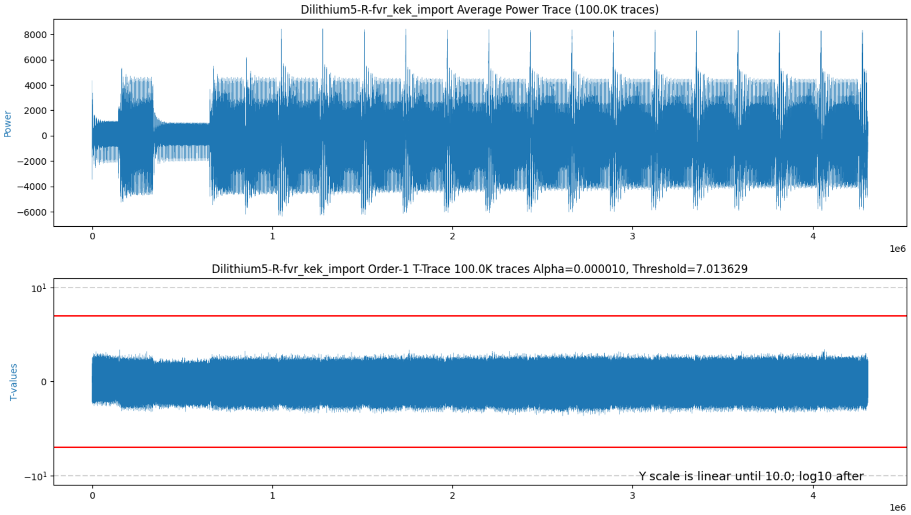

Fig. 9. Dilithium5 WrapQ Import Random-vs-Fixed KEK, 100K Traces.

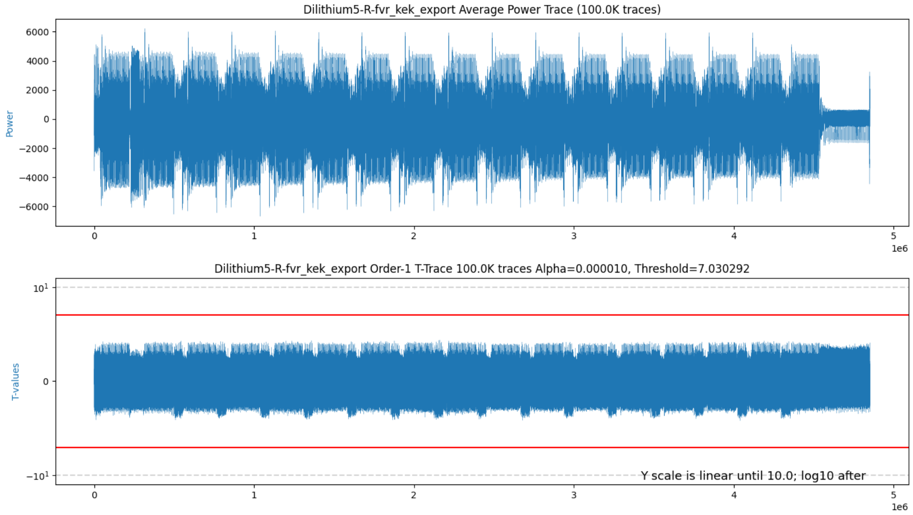

Fig. 10. Dilithium5 WrapQ Export Random-vs-Fixed KEK, 100K Traces.