Mask Compression: High-Order Masking on Memory-Constrained Devices

Markku-Juhani O. Saarinen[1] and Mélissa Rossi[2]

[1] PQShield Ltd., Oxford, UK, mjos@pqshield.com
[2] ANSSI, France, melissa.rossi@ssi.gouv.fr

Abstract. Masking is a well-studied method for achieving provable security against side-channel attacks. In masking, each sensitive variable is split into d randomized shares, and computations are performed with those shares. In addition to the computational overhead of masked arithmetic, masking also has a storage cost, increasing the requirements for working memory and secret key storage proportionally with d.

In this work, we introduce mask compression. This conceptually simple technique is based on standard, non-masked symmetric cryptography. Mask compression allows an implementation to dynamically replace individual shares of large arithmetic objects (such as polynomial rings) with κ-bit cryptographic seeds (or temporary keys) when they are not in computational use. Since κ does not need to be larger than the security parameter (e.g., $\kappa = 256$ bits) and each polynomial share may be several kilobytes in size, this radically reduces the memory requirement of high-order masking. Overall provable security properties can be maintained by using appropriate gadgets to manage the compressed shares. We describe gadgets with Non-Inteference (NI) and composable Strong-Non Interference (SNI) security arguments.

Mask compression can be applied in various settings, including symmetric cryptography, code-based cryptography, and lattice-based cryptography. It is especially useful for cryptographic primitives that allow quasilinear-complexity masking and hence are practically capable of very high masking orders. We illustrate this with a $d = 32$ (Order-31) implementation of the recently introduced lattice-based signature scheme Raccoon on an FPGA platform with limited memory resources.

Keywords: Side-Channel Security · Mask Compression · Raccoon Signature Scheme · Post-Quantum Cryptography

1 Introduction

Physical side-channel attacks exploit sensitive information leaked by a cryptography system via externally observable characteristics such as Timing [20], Power consumption (SPA/DPA) [21,22], and Electromagnetic emissions [30].

Currently, NIST and the cryptographic community are engaged in a wide-reaching transition effort to use Post-Quantum Cryptography (PQC) algorithms

such as Kyber [2] (a lattice-based key establishment scheme) and Dilithium [4] (a lattice-based signature scheme) to replace older quantum-vulnerable RSA and Elliptic Curve based cryptography [1,26]. In many prominent use cases, this transition requires physical side-channel security from PQC implementations: Authentication tokens, Mobile / IoT device platform security (secure boot, firmware update, attestation), smart cards, and other secure elements.

Masking. Masking is a general technique to attain side-channel security by splitting sensitive variables into d randomized shares, where $t = d - 1$ is the *masking order*. Each share individually appears uniformly random, and all d shares are required to determine their sum, which is the actual masked quantity. We write $[\![x]\!]$ to denote a masked representation of x. The relationship may be either an exclusive-or operation ("Boolean masking") or modular ("Arithmetic masking"):

$$\text{Boolean masking: } [\![x]\!] = x_0 \oplus x_1 \oplus \ldots \oplus x_t \tag{1}$$

$$\text{Arithmetic masking: } [\![x]\!] = x_0 + x_1 + \ldots + x_{d-1} \pmod{q}. \tag{2}$$

PQC algorithm side-channel countermeasures are primarily based on masking. For example, see [6,14] for details about masking Kyber, and [24,3] for Dilithium. High-order computation on the shares is relatively complex in the case of these two algorithms, requiring both Boolean and Arithmetic masking.

Complexity of Attack and Defence. In addition to practicality, one main advantage of masking over more ad-hoc approaches is that it allows one to prove side-channel security properties of implementations. In pioneering work, Chari et al. [7] showed that in the presence of Gaussian noise, the number of side-channel observations required to determine x from individual bits grows exponentially with the number of shares d. The understanding of this exponential relationship has since been made more precise both theoretically and in practice [13,23,18].

In [15], Ishai et al. introduced the probing model: the notion of t-probing security states that the joint distribution of any set of at most t internal intermediate values should be independent of any of the secrets. Thus, a circuit is t-probing secure iff it is secure against observations of $t = d - 1$ wires. Reductions from the probing model to the noisy leakage model [29,12] exist and allow to link t-probing security with realistic leakage models.

In addition, [15] showed that any circuit can be transformed into a t-probing secure circuit of size $O(nt^2)$. It has since been demonstrated that quasilinear $O(t \log t)$ masking complexity can be achieved for some primitives, including the Lattice-based signature scheme Raccoon [27,28].

Structure of this Paper and Our Contributions. The mask compression technique is introduced in Section 2, which also discusses how it can be applied in practice. Further security discussion is given in Section 3, including requirements for composability (strong non-interference). Section 4 gives a practical example of a very high-order PQC scheme (Raccoon [28] with $d = 32$ shares) implemented with Mask Compression on FPGA with 128kB of physical SRAM, instead of several megabytes that would be required without it.

2 Mask Compression

Mask compression in a group G (Eqs. 1 or 2) requires a symmetric cryptography primitive $\mathsf{Sample}_G(z)$ that maps short binary keys z to elements in G. The function is used to manipulate sensitive variables, but thanks to the way it is used, $\mathsf{Sample}_G(z)$ itself does *not* need to be masked. Its input and output variables are generally ephemeral (single-use) individual shares.

Definition 1. *(Informal.) The function $x \leftarrow \mathsf{Sample}_G(z)$ uses the input seed $z \in \{0,1\}^\kappa$ to deterministically sample a pseudorandom element $x \in G$. We assume that Sample_G is cryptographically secure under a suitable definition.*

For a technical discussion of pseudorandomness, see [19, Section 3] (the definitions offered for binary strings can be easily extended to other uniform distributions.) Intuitively, we assume that the task of distinguishing x from a uniformly random element in set G is computationally hard. Typically key size κ is selected to match the overall security level of the system. In this case, distinguishing x should not be substantially easier than an exhaustive search for z.

Practical instantiation. We can implement $\mathsf{Sample}_G(z)$ with an extendable output function (XOF) such as SHAKE[25][3] The function can also be instantiated with a stream cipher or a block cipher (in counter mode). If a mapping from XOF output to non-binary uniform distributions is required, one may use rejection sampling since each $\mathsf{XOF}(z)$ defines an arbitrarily long bit sequence.

Examples of sampled $|G| \gg 2^\kappa$ include large-degree polynomials that are ring elements $\mathbb{Z}_q[x]/(x^n + 1)$ in Kyber [2] 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. In common lattice algorithms, an efficient (unmasked) method for this task is required to create polynomials for $\mathbf{A}$ generator matrix on the fly. This is the reason why a PQC hardware implementation (such as the one discussed in Section 4) will often have an efficient instance of $\mathsf{Sample}_G(z)$ available.

Definition 2 (Compressed Mask Set). *A compressed mask set consists of a tuple $[\![x]\!]^z = (x_0, z_1, \cdots, z_t)$ satisfying $x \equiv x_0 + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i)$ with $x_0 \in G$ and $z_i \in \{0,1\}^\kappa$ for $i \in [1,t]$.*

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

Proof. If x_0 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.

[3] FIPS 202 presents a SHAKE specifically as an extensible output function (XOF), which is defined as a hash function with arbitrary-length input and output.

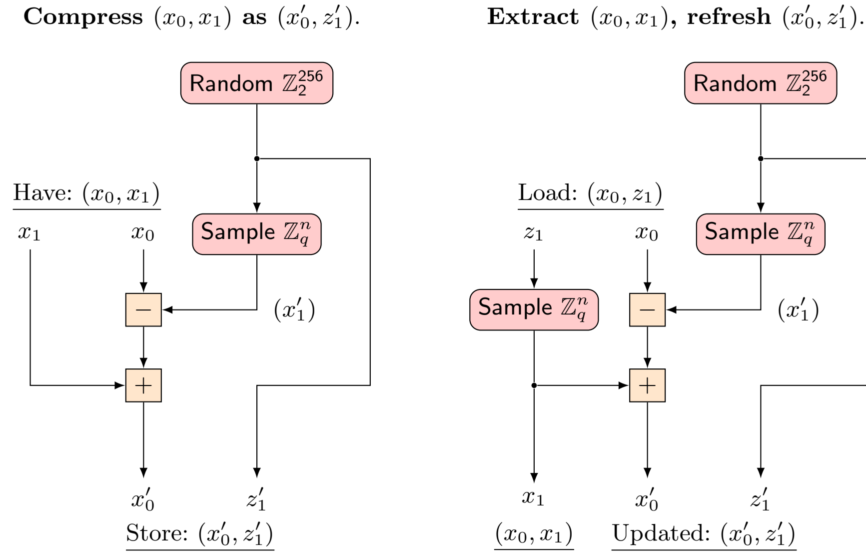

Fig. 1. Illustrating first-order ($t = 1$,$d = 2$) mask compression. Let $[\![x]\!] = (x_0, x_1)$ consist of a pair of degree-n polynomials ($n = 256$ for Kyber, Dilithium) with integer coefficients $\in \mathbb{Z}_q$. Function $\mathsf{Sample}_{\mathbb{Z}_q^n}(z)$ takes a 256-bit key z and uniformly 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" algorithm (analogous to Algorithm 1) 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 algorithm, 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 2) that allows repeated extractions.

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 $\log_2 |G| + d\kappa$ 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 $\kappa = 256$ bits.

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 1 $\mathsf{MaskCompress}$ presents one

Algorithm 1: $[\![\mathsf{x}]\!]^{\mathsf{z}} = \mathsf{MaskCompress}([\![x]\!])$ *(Proved t-NI in Th. 2)*

Input: Masking $[\![x]\!] = (x_0, x_1, \cdots, x_t)$.
Output: Compressed masking $[\![x]\!]^z$ with $x_0^z + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i) = \sum_{i=0}^{t} x_i$.
1: $x_0^z = x_0$
2: **for** $i = 1, 2, \cdots, t$ **do**
3: $z_i \leftarrow \mathsf{Random}(\kappa)$ ▷ Random Bit Generator, κ bits.
4: $x_0^z \leftarrow x_0^z - \mathsf{Sample}_G(z_i)$
5: $x_0^z \leftarrow x_0^z + x_i$
6: **return** $[\![x]\!]^z = (x_0^z, z_1, z_2, \cdots, z_t)$

way of performing this conversion. We note its resemblance to the RefreshMasks algorithm of Rivain and Prouff ([31, Algorithm 4]); its NI security follows similarly (see Section 3 for more details). While it is secure if used appropriately, combining it with other algorithms may expose leakage, as demonstrated in [9]. Depending on requirements, it can be combined with additional refresh steps to build an SNI [5] algorithm (also see Section 3 for more details).

Algorithm 2: $x_i = \mathsf{LoadShare}([\![x]\!]^z, i)$

Input: Compressed masking $[\![x]\!]^z$ satisfying $x = x_0^z + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i)$
Input: Index i for the share to be accessed.
Output: If read in order, $i = 0, 1, \cdots t$, the returned $\{x_i\}$ is a fresh masking $[\![x]\!]$.
1: **if** $i = 0$ **then**
2: $x_i^{\text{out}} \leftarrow x_0^z$ ▷ Should be accessed first, the rest $i > 0$ only once.
3: **else**
4: $x_i^{\text{out}} \leftarrow \mathsf{Sample}_G(z_i)$ ▷ Expand the current z_i.
5: $z_i \leftarrow \mathsf{Random}(\kappa)$ ▷ Update z_i with a Random Bit Generator.
6: $x_0^z \leftarrow x_0^z - \mathsf{Sample}_G(z_i)$
7: $x_0^z \leftarrow x_0^z + x_i^{\text{out}}$ ▷ Update x_0^z accordingly.
8: **return** x_i^{out}

Algorithm 3: $[\![x]\!] = \mathsf{FullLoadShare}([\![x]\!]^z)$ *(Proved t-NI Th. 3)*

Input: Compressed masking $[\![x]\!]^z$ satisfying $x = x_0^z + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i)$
Input: Index i for the share to be accessed.
Output: If read in order, $i = 0, 1, \cdots t$, the returned $\{x_i\}$ is a fresh masking $[\![x]\!]$.
1: **for** $i = 0, 1, \cdots, t$ **do**
2: $x_i \leftarrow \mathsf{LoadShare}([\![x]\!]^z, i)$
3: **return** $(x_0, x_1, \cdots, x_t)$

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 2, $\mathsf{LoadShare}([\![x]\!]^z, i)$ decodes a share $x_i \in G$ from a compressed masking $[\![x]\!]^z$. If the shares are accessed in the sequence $i = 0, 1, 2, \cdots, t$, like presented in Algorithm 3, it is easy to show that their sum will satisfy $x = \sum_{i=0}^{t} x_i$. 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 $[\![x]\!]'$.

3 Security arguments

Let us introduce some standard, intermediate security properties used in security proofs [31,9,5].

Definition 3 (t-Non Interference [5]). *An algorithm is said to be t-non-interfering (written t-NI for short) iff any set of at most t observed internal intermediate variables can be perfectly simulated from at most t shares of each input.*

One can see that t-non interference implies t-probing security. Such a precise definition allows simulation proofs for sequential compositions of non-interferent parts. Note that stronger security notions have been introduced in [5] like the t-strong non-interference to handle more than sequential compositions.

Definition 4 (t-Strong Non Interference [5]). *An algorithm is said t-strongly-non-interfering (written t-SNI for short) if and only if any set of at most $t = t_{\text{int}} + t_{\text{out}}$ observed variables where t_{int} are made on internal data and t_{out} are made on the outputs can be perfectly simulated from at most t_{int} shares of each input.*

We observe that t-strong non-interference implies t-non interference. Any non-interferent algorithm can achieve strong non-interference with an extra mask refreshing of its output [5].

Considering MaskCompress (Algorithm 1) , we propose the following Theorem and prove it below.

Theorem 2. *Algorithm 1 is d-Non Interferent under the Pseudorandom Function hypothesis on the Sample_G function (Definition 1). Hence, it is also t-probing secure.*

Let us first assume that there exists an index $i^* \in \{1, ..., d\}$ such that both the seed z_{i^*} and the input x_{i^*} are left unobserved by the probing attacker. With a hybrid argument, under the pseudorandomness hypothesis on the $\mathsf{Sample}_G(z_{i^*})$ function, $\mathsf{Sample}_G(z_{i^*})$ may be replaced by a uniform random value in G, denoted y^*. Hence, all the intermediate values that intervene in the i^*-th iteration can be simulated with uniform random. Therefore, the distribution of the observations can be simulated with at most t shares of the input (x_i for $i \neq i^*$) under the computational assumption.

Now assume that it is not possible to find such an index $i^* \in \{1, ..., d\}$. In that case, all the t observations are made on a combination of x_i for $i \in \{1, t\}$ and z_i for $i \in \{1, t\}$. Let us note that in that case, the input x_0 is always left unobserved. The distribution of x_0^z over all iterations is then statistically indistinguishable from uniform random in G. Hence, the distribution of the observations can be simulated with at most t shares of the input (x_i for $i \in \{1, t\}$).

Algorithm 4: $[\![\mathsf{x}]\!]^{\mathsf{z}} = \mathsf{SNIMaskCompress}([\![x]\!])$ *(Proved t-SNI in Th. 4)*

Input: Masking $[\![x]\!] = (x_0, x_1, \cdots, x_t)$.
Output: Compressed masking $[\![x]\!]^z$ with $x_0^z + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i) = \sum_{i=0}^{t} x_i$.

```
1: x_0^z = x_0
2: for i = 1, 2, ··· , t do
3:     z_i ← Random(κ)
4:     x_0^z ← x_0^z − Sample_G(z_i)
5:     x_0^z ← x_0^z + x_i          ▷ Compared to Alg .6, x_i is directly accessed
6: for j = 1, 2, ··· , t do
7:     for i = 1, 2, ··· , t do
8:         x_i ← Sample_G(z_i)
9:         z_i ← Random(κ)
10:        x_0^z ← x_0^z − Sample_G(z_i)
11:        x_0^z ← x_0^z + x_i
12: return [[x]]^z = (x_0^z, z_1, z_2, ··· , z_t)
```

Let us now consider the $\mathsf{LoadShare}$ algorithm. As noted above, the full version of Algorithm 2, presented in Algorithm 3, is very similar to the Non-Interferent $\mathsf{RefreshMasks}$ algorithm introduced in [31]. Hence, we introduce the following Theorem.

Theorem 3. *Algorithm 3 is d-Non Interferent and thus t-probing secure under the pseudorandomness hypothesis on the* Sample_G *function (Definition 1).*

Since there are $t+1$ iterations and at most t observations, there exists an index $i^* \in \{0, ..., t\}$ designating an iteration that is left unobserved by the probing attacker. Hence both the input seed z_{i^*} and the value x_0^z (of the i^*-th iteration) are left unobserved. In that case, all the subsequent updates of x_0^z can

be replaced with uniform random under the same pseudorandomness hypothesis of Sample_G. Finally, all the attacker's observations may be simulated with $(x_0, (z_i)_{i \neq i^*})$ if $i^* \neq 0$ and all the (z_i) otherwise. There are no more than t shares of the input, which concludes the proof.

Algorithm 5: $x_i = \mathsf{SNILoadShare}([\![x]\!]^z, i)$

Input: Compressed masking $[\![x]\!]^z$ satisfying $x = x_0^z + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i)$
Input: Index i for the share to be accessed.
Output: If read in order, $i = 0, 1, \cdots t$, the returned $\{x_i\}$ is a fresh masking $[\![x]\!]$.

```
1: if i = 0 then
2:     x_i^out ← x_0^z          ▷ Should be accessed first, the rest i > 0 only once.
3: else
4:     x_i^out ← Sample_G(z_i)                            ▷ Expand the current z_i.
5: for j = 1, 2, ··· , t do
6:     x_j ← Sample_G(z_j)
7:     z_j ← Random(κ)                ▷ Update z_i with a Random Bit Generator.
8:     x_0^z ← x_0^z − Sample_G(z_j)
9:     x_0^z ← x_0^z + x_j                              ▷ Update x_0^z accordingly.
10: return x_i^out
```

Strong non interference Our mask compression design does not immediately reach the strong non-interference security notion; thus, it cannot be directly composed in complex designs. As outlined above, for safe composition properties, applying a Strong Non-Interferent mask refreshing like introduced in [8] is important. We present in Algorithm 6 an SNI refresh procedure on compressed masks. Applying Algorithm 6 at the beginning of $\mathsf{FullLoadShare}$ and at the end $\mathsf{MaskCompress}$ allows one to easily reach the strong Non-Interference property.

However, it is also possible to slightly save some randomness and directly transform both our algorithms such that they reach the SNI property. We introduce them in Algorithms 4, 5 and 7.

Please note that these three SNI gadgets will not be used in Section 4 for Raccoon but they are provided here for potential other applications.

Theorem 4. *Algorithms 4, 6 and 7 are d-Strongly Non-Interferent under the Pseudorandomness hypothesis of* Sample_G *function (Definition Definition 1). They may be safely composed in complex designs.*

In Algorithms 4 and 6, since there are $t+1 = d$ iterations (with one outside of the loop with index j for Algorithm 4) and t observations, at least one iteration is left unobserved. All the observations (including the observations on the output) performed after the unobserved iteration can be simulated with uniform random (under the same pseudorandomness hypothesis). All the observations performed

Algorithm 6: $[\![x]\!]^z = \mathsf{SNIRefresh}([\![x]\!])$ *(Proved t-SNI in Th. 4)*

Input: Compressed masking $[\![x]\!]^z$
Output: Compressed masking $[\![x]\!]^z$ with fresh shares.
1: **for** $j = 0, 1, \cdots, t$ **do**
2: **for** $i = 1, 2, \cdots, t$ **do**
3: $x_i \leftarrow \mathsf{Sample}_G(z_i)$
4: $z_i \leftarrow \mathsf{Random}(\kappa)$
5: $x_0^z \leftarrow x_0^z - \mathsf{Sample}_G(z_i)$
6: $x_0^z \leftarrow x_0^z + x_i$
7: **return** $[\![x]\!]^z = (x_0^z, z_1, z_2, \cdots, z_t)$

Algorithm 7: $[\![x]\!] = \mathsf{FullLoadShare}([\![x]\!]^z)$ *(Proved t-SNI Th. 4)*

Input: Compressed masking $[\![x]\!]^z$ satisfying $x = x_0^z + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i)$
Input: Index i for the share to be accessed.
Output: If read in order, $i = 0, 1, \cdots t$, the returned $\{x_i\}$ is a fresh masking $[\![x]\!]$.
1: **for** $i = 0 \cdots, t$ **do**
2: $x_i \leftarrow \mathsf{SNILoadShare}([\![x]\!]^z, i)$
3: **return** $(x_0, x_1, \cdots, x_t)$

before the unobserved iteration can be simulated with at most t shares of the input (inherited from the NI property of Algorithm 1). For Algorithm 7, one can switch the loops for i and j and apply the same reasoning.

4 Experiment: Order-31 Lattice Signatures

We illustrate Mask Compressions with Raccoon[4] at very high masking order 31 (number of shares $d = t + 1 = 32$) [28]. The unit performs all of the masked arithmetic in $\mathsf{KeyGen}()$, and $\mathsf{Sign}()$, and also implements $\mathsf{Verif}()$. We will focus on the masked signing process, reproduced in Algorithm 8.

Overview of the hardware. The FPGA implementation contains an RV32C controller, a 24-cycle Keccak accelerator, and a lattice unit with direct memory access via a 64-bit interface. The lattice unit has hard-coded support for Raccoon's mod q arithmetic. It can perform arbitrary-length vector arithmetic operations such as polynomial addition, coefficient multiplication, NTT butterfly operations, and shifts on 64-bit words. The FPGA implementation has a 5-cycle modular multiplier with a 64-to-49 bit fixed-modulus reduction circuit. All variants of Raccoon utilize the same modulus q, allowing "hard-coded" reduction circuitry to be used to implement them all.

Since the implementation is designed for masking, the circuitry also has a fast "random fill" function that generates non-deterministic masking random rapidly.

[4] The discussion applies to the version of Raccoon published at IEEE S&P 2023 [28]. There are differences to the Raccoon version submitted to the NIST PQC Call [27].

In a production implementation, this function would require special attention to guarantee that the randomness used in each share is genuinely independent, but trivial entropy sources with simple ASCON [11] -based mixing function was used in the prototype.

4.1 $\mathsf{Sample}_G(z)$ in Hardware

Crucially, the hardware can directly perform mod q rejection sampling from streaming SHAKE output to memory. Since a full Keccak round is implemented in hardware, it produces output at a very high rate, theoretically a full block (136 bytes for SHAKE-256) every 24 cycles. This function works in parallel with other operations. We found that bus access and arithmetic steps tend to be the performance bottlenecks rather than the rejection sampling component.

In addition to implementing $\mathsf{Sample}_G(z)$, the rejection sampler eliminates perhaps the most significant performance bottleneck in microcontroller lattice-based PQC implementations: It was initially intended to generate the $k \times \ell$ polynomial matrix $\mathbf{A}$ on the fly (Lines 2 and 12 in Algorithm 8, similar requirement in key generation and verification functions.) Such on-the-fly generation of $\mathbf{A}$ is also required in Kyber and Dilithium implementations. Hence a rejection sampler of this type can be expected to be available in dedicated PQC hardware.

Share Access Gadgets For this implementation, we used gadgets based on Algorithms 1 and 2, implemented as a library call with an "API" for loading and storing mask sets consistently (so that leakage characteristics would be uniform). Note that while the introduced SNI gadgets are not used in Raccoon – this would violate the quasilinear complexity requirement – the NI gadgets in Algorithms 1 and 2 suffice to ensure the probing security of Raccoon. One polynomial (x_0 in Definition 2) was held as full 64-bit integers to facilitate fast hardware arithmetic, while the rest ($t = 31$ shares) were stored as $\kappa = 256$ bit seeds. Note that each arithmetic step utilized the shares one at a time (thanks to the requirements of the quasilinear lattice cryptography) $i = 0, 1, \cdots, t$. When a share i was required for arithmetic, an implementation of Algorithm 2 gadget was called. For storing $i = 0, 1, \cdots, t$, the share $i = 0$ was stored in full, while the rest utilized Lines 3-6 of Algorithm 1 to update it. The implementation of Decode function does not require simultaneous refresh, so it is sufficient to simply compute $x_0 + \sum_{i=1}^{t} \mathsf{Sample}_G(z_i)$.

Memory Footprint Algorithm 8 has been annotated with the share-access gadgets used in each stage, which allow the implementation to use mask compression on each sensitive variable. All of these are vectors of polynomial rings $\mathcal{R}_q$, with dimension depending on the security level $\lambda_{\mathsf{target}} \in \{128, 192, 256\}$ [28, Table 3]. Focusing on the "Category 1" Raccoon-128 parameter sets the vector length is either $\ell = 3$ (for $[\![\mathbf{r}]\!]$, $[\![\mathbf{s}]\!]$, and $[\![\mathbf{z}]\!]$) or $k = 8$ (for $[\![\mathbf{u}]\!]$ and $[\![\mathbf{w}]\!]$.) For the masked variables, only the secret key $[\![\mathbf{s}]\!]$ needs to be retained for repeated use. Not all internal variables are used concurrently, and hence e.g., $[\![\mathbf{u}]\!]$ and $[\![\mathbf{w}]\!]$

can occupy the same memory as $[\![\mathbf{r}]\!]$ and $[\![\mathbf{z}]\!]$. Hence this Raccoon implementation requires $\ell = 3$ masked polynomials for persistent storage (secret key), and additional $\ell + k = 11$ for working memory.

To estimate the minimum memory requirement at $t = 31$ without mask compression, we assume that each polynomial coefficient is bit-packed into $\lceil \log_2 q \rceil = 49$ bits; hence a masked polynomial requires $d \times n \times 49 = 802,816$ bits. For both secret key and working memory, this comes to roughly 1.4 megabytes (close to 2 MB if coefficients are stored in an access-friendly manner as 64-bit integers.)

With mask compression, the size of each masked polynomial drops to $n \times 49 + t \times \kappa = 33,024$ bits, or 4.1% of the uncompressed mask size. This is only a 31.6% increase over completely unmasked implementation, even for the very high masking order of 31; one can well say that the storage cost of masking becomes negligible with mask compression.

The physical FPGA implementation operated well with 128 kB of SRAM, while at least 2000 kB would have been required without compression. The secret key $[\![\mathbf{s}]\!]$ size also shrunk from 294 kB to 12.1 kB, which is important as non-volatile storage can be more scarce than working memory.

Algorithm 8: $\mathsf{Sign}([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{msg})$: "IEEE SP '23" Raccoon signing [28, Algorithm 7] with applicable mask compression gadgets annotated in the comments. (Note: There are differences to the "NIST" version [27].)

Input: A masked signing key $[\![\mathsf{sk}]\!]$, a message msg
Output: A signature sig of msg under sk

1: $[\![\mathbf{r}]\!] \leftarrow (\mathcal{R}_q^\ell)^d$ ▷ In the implementation: A random mask set!
2: $[\![\mathbf{u}]\!] := \mathbf{A} \cdot [\![\mathbf{r}]\!]$ ▷ Access: NI Alg. 1,2 or SNI Alg. 4,5.
3: $[\![\mathbf{u}]\!] \leftarrow \mathsf{Refresh}([\![\mathbf{u}]\!])$ ▷ Implicit with NI or SNI with Alg. 6.
4: $[\![\mathbf{w}]\!] := \mathsf{ApproxShift}_{q \to q_w}([\![\mathbf{u}]\!])$ ▷ Access: NI Alg. 1,2 or SNI Alg. 4,5.
5: $\mathbf{w} := \mathsf{Decode}([\![\mathbf{w}]\!])$ ▷ Commitment. NI: Alg. 3 or SNI Alg. 7.
6: $\mathbf{c}_{\mathsf{hash}} := H(\mathbf{w}, \mathsf{msg})$ ▷ Challenge hash. (Not masked.)
7: $\mathbf{c}_{\mathsf{poly}} := \mathsf{ChalPoly}(\mathbf{c}_{\mathsf{hash}})$ ▷ Challenge polynomial. (Not masked.)
8: $[\![\mathbf{s}]\!] \leftarrow \mathsf{Refresh}([\![\mathbf{s}]\!])$ ▷ Implicit with NI or SNI with Alg. 6.
9: $[\![\mathbf{r}]\!] \leftarrow \mathsf{Refresh}([\![\mathbf{r}]\!])$ ▷ Implicit with NI or SNI with Alg. 6.
10: $[\![\mathbf{z}]\!] := \mathbf{c}_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$ ▷ Access: NI Alg. 1,2 or SNI Alg. 4,5.
11: $\mathbf{z} := \mathsf{Decode}([\![\mathbf{z}]\!])$ ▷ Response. NI: Alg. 3 or SNI Alg. 7.
12: $\mathbf{y} := \mathbf{A} \cdot \mathbf{z} - p_t \cdot \mathbf{c}_{\mathsf{poly}} \cdot \mathbf{t}$ ▷ (The rest is not masked.)
13: $\mathbf{y}^{\mathsf{top}} := \lfloor \mathbf{y} \rceil_{q \to q_w}$
14: $\mathbf{h} := \mathbf{w} - \mathbf{y}^{\mathsf{top}}$ ▷ Hint.
15: **if** $(\|\mathbf{h}\|_2 > B_2)$ **or** $(\|\mathbf{h}\|_\infty > B_\infty)$ **then**
16: **goto** Line 1 ▷ Check the hint's norms.
17: **return** $\mathsf{sig} := (\mathbf{c}_{\mathsf{hash}}, \mathbf{z}, \mathbf{h})$

4.2 Implementation Details and Basic Leakage Assessment

On an XC7A100T (Xilinx Artix 7) FPGA target, this size-optimized design (including a control Core, Keccak unit, lattice coprocessor, masking random num-

ber generator, and communication peripherals) was 10,638 Slice LUTs (16.78%), 4,140 Slice registers / Flip Flops, (3.26%) and only 3 DSPs (as logic was used for multipliers – the design is ASIC-oriented). The design was rated for 78.3 MHz. Table 1 summarizes its performance at various masking levels.

Table 1. FPGA cycle counts at various side-channel security levels.

Algorithm	Shares	Keygen()	Sign()	Verif()
Raccoon-128	$d = 2$	1,366,000	2,402,000	1,438,000
Raccoon-128	$d = 4$	2,945,000	3,714,230	1,433,034
Raccoon-128	$d = 8$	6,100,000	6,345,000	1,389,000
Raccoon-128	$d = 16$	12,413,000	11,605,000	1,389,000
Raccoon-128	$d = 32$	25,073,000	22,160,000	1,393,000

Leakage Assessments. We ran a TVLA/17825:2022(E) [17] type leakage assessment on all orders from $d = 2$ up to $d = 32$, with $N = 200,000$ traces at $d = 2$ showing no leakage. Such detection mechanisms are generally limited to first-order leakage, so testing a high-order implementation can be seen as unnecessary. However, in this particular case, there is an additional risk that the mask compression gadgets themselves would be leaking.

Fixed vs. Random test. A non-specific t-test [32] was conducted on the signing function to assess leakage of secret key $[\![\mathbf{s}]\!]$. The fixed set of traces consisted of signing operations using synthetic keypairs where the secret $[\![\mathbf{s}]\!]$ component was fixed (but refreshed for every operation), and the public $\mathbf{A}$ was randomized. For the signing operation, a synthetic $\mathbf{t}$ is derived with the fixed $[\![\mathbf{s}]\!]$ and randomized $\mathbf{A}$. The second random set of traces used completely random keypairs. The message to be signed was constant in both tests.

Critical Value. At order $d = 32$, the leakage assessment was carried out with $N = 20,000$ full traces and passed well under a threshold value matching $\alpha = 10^{-5}$. As noted by several authors, for example, Ding et al. [10] and Oswald et al. [33], the common "TVLA" threshold value 4.5 needs to be adjusted for long traces (the overall false positive rate with millions of points would be close to 1.) The threshold value corresponding to significance level $\alpha = 10^{-5}$ with $l = 2.59 \times 10^6$ time points is $C = 6.94$, using the methodology of [10].

Signal acquisition and post-processing. Power signal was acquired from the FPGA chip on the CW305 board [16, Sect. C.3] with a PicoScope 2208B oscilloscope. The test was run with a 24ns (41.7 MHz) clock cycle. Power samples were gathered at the same rate. Each trace of the signature operation contained more than 22 million samples at $d = 32$. The DUT generated a cycle-precise trigger. Random delays and other non-masking countermeasures were disabled.

We applied post-processing steps to improve detection. The waveforms were computationally normalized so that each 1 ms sliding window had $\mu = 0$ and $\sigma^2 = 1$ (effectively, a 1 kHz high-pass filter and dynamic amplitude control). This allowed the traces to match more closely on the vertical axis. The traces were also aligned horizontally using the start and end triggers.

Results. At N=20,000 traces, the maximum t-value was 5.55 (Fig. 2), well under the threshold and corresponding to P-value 0.47. At N=10,000 traces, the test result was $t = 5.43$. We also verified that leakage detection is functional by disabling countermeasures in various ways; spikes rapidly appear in those cases.

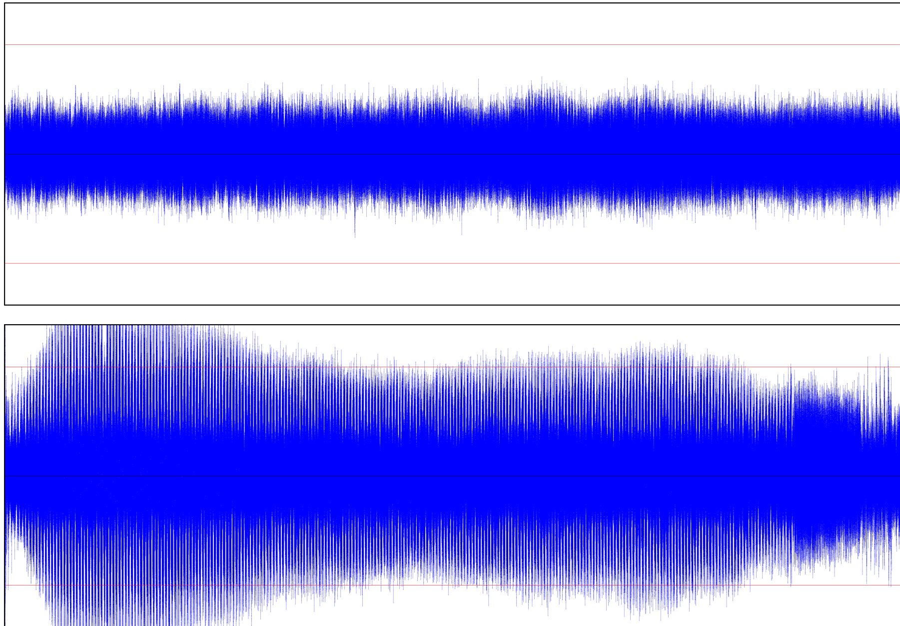

Fig. 2. On top, t-trace of Raccoon-128 ($d = 32$) signature function from $N = 20,000$ waveforms, each with 22.16×10^6 measurements (time on the horizontal axis). No leakage spikes were detected; the t-statistic values are within the critical value boundaries (thin red lines). This test only detects first-order leakage, so it is merely offered as additional evidence related to the implementation of the mask compression gadgets. The bottom figure has $N = 500$ traces of the same implementation with mask randomization disabled; this simply demonstrates that leakage detection was operational.

5 Conclusions and Open Problems

We have introduced *Mask Compression*, a method to reduce the memory cost of high-order masking side-channel countermeasures using non-masked symmetric

cryptography. This simple technique allows a set of t-order mask shares to have a storage requirement equivalent to a single share and t symmetric keys. Its benefits are most significant in higher-order masking, but it also nearly halves the memory requirement for first-order Kyber and Dilithium. We present security arguments in the well-known NI and SNI frameworks.

To illustrate the technique's utility, we describe an Order-31 implementation of the Raccoon signature scheme [28] where the size of the secret keys is reduced from 294kB to 12kB. The overall memory requirement is reduced from two megabytes to 128 kB, allowing the scheme to be implemented on a resource-constrained FPGA target while maintaining a quasilinear masking complexity and a high level of non-invasive side-channel security, but with NI gadgets only. As an open problem, we are working on closing SNI composability gaps for some of the components and providing SNI gadgets with quasilinear complexity.

References

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-upd1, National Institute of Standards and Technology (September 2022). https://doi.org/10.6028/NIST.IR.8413-upd1, https://csrc.nist.gov/publications/detail/nistir/8413/final
2. 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
3. Azouaoui, M., Bronchain, O., Cassiers, G., Hoffmann, C., Kuzovkova, Y., Renes, J., Schönauer, M., Schneider, T., Standaert, F.X., van Vredendaal, C.: Leveling Dilithium against leakage: Revisited sensitivity analysis and improved implementations. IACR ePrint 2022/1406 (2022), https://eprint.iacr.org/2022/1406, fourth PQC Standardization Conference, NIST (Virtual) 29 Nov – 1 Dec 2022
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.) CCS '16: 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. 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

7. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener [34], pp. 398–412. https://doi.org/10.1007/3-540-48405-1_26
8. Coron, J.: Higher order masking of look-up tables. In: Nguyen, P.Q., Oswald, E. (eds.) Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8441, pp. 441–458. Springer (2014). https://doi.org/10.1007/978-3-642-55220-5_25
9. 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
10. 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
11. 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
12. 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
13. Duc, A., Faust, S., Standaert, F.: Making masking security proofs concrete - or how to evaluate the security of any leaking device. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 401–429. Springer (2015). https://doi.org/10.1007/978-3-662-46800-5_16, https://eprint.iacr.org/2015/119, extended version is available as IACR ePrint Report 2015/015
14. 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
15. 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
16. ISO: IT security techniques – test tool requirements and test tool calibration methods for use in testing non-invasive attack mitigation techniques in cryptographic modules – part 2: Test calibration methods and apparatus. Standard ISO/IEC 20085-2:2020(E), International Organization for Standardization (2020), https://www.iso.org/standard/70082.html
17. ISO: Information technology – security techniques – testing methods for the mitigation of non-invasive attack classes against cryptographic modules. Draft In-

ternational Standard ISO/IEC DIS 17825:2022(E), International Organization for Standardization (2023)
18. Ito, A., Ueno, R., Homma, N.: On the success rate of side-channel attacks on masked implementations: Information-theoretical bounds and their practical usage. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. pp. 1521–1535. ACM (2022). https://doi.org/10.1145/3548606.3560579, https://eprint.iacr.org/2022/576
19. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Third Edition. CRC Press (2021). https://doi.org/10.1201/9781351133036
20. 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
21. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener [34], pp. 388–397. https://doi.org/10.1007/3-540-48405-1_25
22. Kocher, P.C., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to differential power analysis. Journal of Cryptographic Engineering **1**(1), 5–27 (2011). https://doi.org/10.1007/s13389-011-0006-y
23. Masure, L., Rioul, O., Standaert, F.: A nearly tight proof of duc et al.'s conjectured security bound for masked implementations. In: Buhan, I., Schneider, T. (eds.) Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13820, pp. 69–81. Springer (2022). https://doi.org/10.1007/978-3-031-25319-5_4, https://eprint.iacr.org/2022/600
24. 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
25. 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
26. 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
27. del Pino, R., Espitau, T., Katsumata, S., Maller, M., Mouhartem, F., Prest, T., Rossi, M., Saarinen, M.J.O.: Raccoon: A side-channel secure signature scheme. Submission to NIST Standardization Call for Additional PQC Signature Schemes (June 2023), https://github.com/masksign/raccoon/blob/main/doc/raccoon.pdf
28. del Pino, R., Prest, T., Rossi, M., Saarinen, M.J.O.: High-order masking of lattice signatures in quasilinear time. In: 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, 22-25 May 2023. pp. 1168–1185. IEEE (2023). https://doi.org/10.1109/SP46215.2023.00160

29. 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
30. 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
31. 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
32. 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
33. 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
34. 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