

ML-DSA masking sweetened with SUCRE: Shuffle-and-Unmask Countermeasure for REjection sampling

Sonia Belaïd¹, Ryad Benadjila¹, Julien Devevey², Morgane Guerreau^{3*},
Thomas Legavre^{4,5}, Ange Martinelli², Thomas Ricosset⁴, Matthieu Rivain¹
and Mélissa Rossi¹

¹ CryptoExperts, Paris, France, firstname.lastname@cryptoexperts.com

² ANSSI, firstname.lastname@ssi.gouv.fr

³ PQShield, firstname.lastname@pqshield.com

⁴ Thales, firstname.lastname@thalesgroup.com

⁵ Sorbonne Université, CNRS, LIP6, firstname.lastname@lip6.fr

Abstract. We present SUCRE, a novel countermeasure designed to physically protect the rejection sampling step of ML-DSA, one of the post-quantum signature schemes standardized by NIST. At the core of SUCRE is a masking gadget that securely un-masks a vector while simultaneously applying a random permutation of its coefficients. This lightweight mechanism preserves the vector’s infinity norm, enabling rejection sampling to proceed as usual without requiring any complex mask conversions. We formally prove that a d -probing adversary can learn at most some permuted rejected values—information which, we show, should remain insufficient to endanger the security of the signature scheme. This security argument relies on a new variant of the Module Learning with Rounding (MLWR) assumption, for which we provide a dedicated concrete security analysis to assess its hardness relative to the standard MLWR assumption.

Our implementation of SUCRE achieves a significant performance improvement over previous masked non-bitsliced implementations of rejection sampling—delivering four to six times faster execution than Coron et al. (TCHES 2024)—albeit at the cost of increased memory usage. Since the rejection step accounts for approximately 25% of the total runtime in masked ML-DSA implementations, and given the expected adoption of ML-DSA on embedded platforms, this speedup could significantly enhance efficiency in real-world applications.

Keywords: ML-DSA · Dilithium · Masking · Module Learning With Rounding · SCA countermeasure · Lattices · Permutation

1 Introduction

The new generation cryptographic algorithms has been designed with the looming threat of an incoming quantum computer. For digital signature schemes, NIST has selected several algorithms for standardization whose security is based either on hash-based constructions—such as XMSS [BDH11], LMS, SLH-DSA [NIS24b] (formerly known as SPHINCS+ [HBD⁺22])—or on structured lattice problems, namely Falcon [PFH⁺22] and ML-DSA [NIS24a] (formerly known as Crystals-Dilithium [LDK⁺22]). Among these, ML-DSA stands out as the most promising candidate for widespread deployment, thanks

*Work done while the author was at CryptoNext Security.

to its conceptual simplicity and excellent overall performance. Its design follows the Fiat-Shamir with Aborts transform, as introduced in [Lyu09], which relies on the use of rejection sampling to prevent any leak of information about the secret key during the signing process.

The anticipated widespread adoption of ML-DSA brings to the forefront the critical issue of secure implementations and, in particular, of side-channel protection. Side-channel attacks (SCA), introduced by Kocher *et al.* in their seminal work [KJJ99], exploit physical leakages—such as power consumption or electromagnetic emanations—to recover intermediate values during the execution of a cryptographic primitive. Today, SCAs represent one of the most significant practical threats against cryptographic implementations on embedded devices. The introduction of new post-quantum cryptographic schemes, especially lattice-based ones, has spurred a new wave of highly effective side-channel attacks [BDE⁺18, UMTS24, GMRR22, ZLYW23, GR24], highlighting the urgent need for dedicated countermeasures in this context.

One of the most prevalent techniques to thwart side-channel attacks is masking, introduced independently in 1999 by Chari, Jutla, Rao, and Rohatgi [CJRR99], and by Goubin and Patarin [GP99]. The core idea behind masking is to split each sensitive variable in an implementation into $d + 1$ shares: d shares are sampled uniformly at random, and the final share is computed to ensure that the combination of all shares reconstructs the original value (typically using a group operation—for instance, addition modulo q for arithmetic masking, or bitwise XOR for Boolean masking). The rationale is to increase the attack’s difficulty: instead of learning directly from the sensitive variable, the adversary must recover information on all the shares, which is significantly more challenging in presence of noise.

To formally analyze the security of masking schemes, it is crucial to model the side-channel leakage an attacker can exploit, which leads to the concept of leakage models. One of the most well-known models is the d -probing model, introduced by Ishai, Sahai, and Wagner in [ISW03], where the attacker is allowed to directly observe the exact value of up to d intermediate variables during a computation. The underlying intuition is that even though all computations might leak, the attacker typically faces an exponential complexity in the number of measurements required to combine noisy observations—a phenomenon first formalized in [CJRR99].

Building masking countermeasures for large circuits, even under the simple d -probing model, remains a challenging task. A common approach is to decompose the circuit into smaller blocks called gadgets, which implement basic masked operations. Secure composition principles then allow these gadgets to be assembled into larger circuits, provided that the gadgets meet certain security properties. In their seminal paper [ISW03], Ishai, Sahai, and Wagner introduced the first secure gadgets for field addition and multiplication, which can be composed arbitrarily. However, their construction requires $2d + 1$ shares for a security level d , which imposes a significant overhead. Subsequent works have largely focused on optimizing this to the minimal number of shares, $d + 1$, which is now considered the standard. Another milestone in this direction was the introduction of the d -(strong) non-interference property (d -(S)NI) in [BBD⁺16]. This property provides a composability framework: by carefully arranging d -NI and d -SNI gadgets within a circuit, one can ensure that the whole masked implementation remains secure in the d -probing model. Affine operations can typically be implemented through efficient d -NI gadgets, and d -SNI secure multiplications require a number of random values and operations quadratic in d .

In the case of lattice-based signature schemes, the design of masking countermeasures becomes even more challenging, as the computations go far beyond simple affine operations and multiplications. These schemes often involve more complex manipulations, which require carefully crafted dedicated gadgets to limit the overhead induced by masking. Moreover, several schemes—including ML-DSA—sometimes unmask intermediate values

during the signing process to improve efficiency. For example, [BBE⁺18], [GR20] and [ABC⁺23] chose to unmask the commitment value in masked schemes following the Fiat-Shamir with Aborts paradigm in order to avoid computing the hash function in masked form. This creates an additional complication from a security perspective: the adversary gains access to information that is not part of the usual output. In such cases, securing the masked implementation not only requires proving that the leakage (including the values observed via probes) can be simulated with at most d shares and these unmasked values, but also demands an additional argument: one must ensure that the exposure of these intermediate values (referred to as public outputs) does not compromise the underlying security guarantees of the scheme, such as sUF-CMA for ML-DSA. This subtle but critical point was first addressed in [BBE⁺18], where the authors analyzed the leakage of the commitment value in ML-DSA’s predecessor, which is revealed to the adversary during the signing process, and demonstrated how to formally integrate this into the security proof.

In essence, signing with ML-DSA consists in sampling a short nonce vector $\mathbf{y}$. This is turned into a commitment $\mathbf{w}_1$, which is hashed along with the message to sign to get a so-called *challenge*. A response $\mathbf{z}$ to this challenge is computed using the signing key, and returned if short enough. Otherwise, the signer restarts this process. While we know that revealing rejected $\mathbf{z}$ leaks the signing key with as few as 2^{10} signatures for ML-DSA-44 [ASY22, Section 3.3], it has not been investigated until now whether some small amount of leakage on $\mathbf{z}$ can be tolerated by the scheme.

1.1 Technical Overview of our Contributions

We now present our contributions, highlighting the key observations and providing a brief overview of our results.

The SUCRE countermeasure for rejection sampling. In ML-DSA, the rejection sampling mechanism ensures that the signature components $\mathbf{z}$ and $\mathbf{r}_0$ —which are sensitive vectors derived from secret-dependent computations—satisfy specific infinity norm bounds. Existing masked implementations often rely on heavy gadgets or expensive arithmetic-to-Boolean conversions. While significant progress has been made on this topic, these conversions still severely degrade performance. We refer to [CGV14, SPOG19, BC22, CGMZ22] for details on these conversion techniques. Our first observation is that the infinity norms of $\mathbf{z}$ and $\mathbf{r}_0$ remain unchanged under any permutation of their coordinates. Leveraging this property, we propose a new gadget that allows one to unmask and check the infinity norm of a randomly permuted version of $\mathbf{z}$ or $\mathbf{r}_0$. As the permutation is randomly chosen for each call to the infinity norm check, this thwarts statistical attacks that rely on recovering aborted sensitive values. From a security perspective, our design also addresses the specific risk of unusually large coordinates in $\mathbf{z}$, which could reveal information about the secret key or the permutation. To mitigate this, we apply a modular reduction before the norm check, ensuring that the reduced $\mathbf{z}$ is uniformly distributed while preserving the correctness of the rejection sampling. With this new design, our countermeasure significantly outperforms existing non-bitsliced solutions in terms of speed performance (at the expense of memory usage) thanks to its lighter operations, avoiding the overhead of conversion gadgets while still providing strong protection against side-channel leakage. Its speed is also on par with state of the art bitslice implementations from [ABC⁺23] (see Table 1 for an extract of SUCRE performances). On the downside, our countermeasure requires more memory than existing solutions, as it needs to store the random permutations. However, we believe that this trade-off can be acceptable in practice, especially considering the significant speedup achieved.

Table 1: Performance comparison on Cortex-M4 (extracted from Table 6).

Speed performance (kilo-cycles) for ML-DSA-65					
Algorithm \ # shares	2	4	6	8	10
Classical (unprotected)	23	23	23	23	23
SecBoundCheck [ABC ⁺ 23]	4 354	15 472	31 234	48 376	—
RejectSampling [CGL ⁺ 24]	15 612	56 761	127 094	221 437	353 375
SUCRE (for $\mathbf{z}$)	2 398	9 907	25 549	43 982	79 188
SUCRE without modular reduction (for $\mathbf{r}_0$)	689	2 320	4 897	8 455	12 774

Existing physical attacks on rejection sampling. The rejection case in SUCRE gives rise to a leakage of intermediate values during the rejection sampling process, namely the tuple $(\mathbf{w}_1, \pi(\mathbf{z}), \pi(\mathbf{r}_0), c)$, where $\pi(\mathbf{z})$ denotes a permutation applied to $\mathbf{z}$. To analyze the potential impact of this leakage, we refer to recent works [AOVCG25, ZWSY25] that exploit the knowledge of $\mathbf{z}$ and c to formulate an Integer Linear Programming (ILP) instance in order to recover the secret $\mathbf{s}_1$. However, such attacks heavily rely on the assumption that the coordinate of $\mathbf{z}$ exceeding the bound can be associated with the corresponding row of c , when c is viewed as a negacyclic matrix. In the case of SUCRE, due to the permutation π , the coordinates of $\pi(\mathbf{z})$ are no longer aligned with those of c , and thus identifying this correspondence requires an exhaustive search over all possible coordinate associations. This significantly increases the attack complexity and renders such approaches impractical in the setting of SUCRE.

The Permuted Hint Module Learning with Rounding (π nt-MLWR) problem and its failed cryptanalysis. With our new countermeasure SUCRE, a d -probing attacker targeting a masked ML-DSA implementation can gain access to the permutations of $\mathbf{z}$ and $\mathbf{r}_0$, as well as the commitment $\mathbf{w}_1$, which is unmasked prior to the computation of the challenge c in existing implementations [ABC⁺23, CGL⁺24]. This observation naturally leads to the formalization of a new problem, which we call the *Permuted Hint Module Learning with Rounding* (π nt-MLWR) problem. This problem captures the adversary’s knowledge of both the public commitment $\mathbf{w}_1$ —which corresponds to a classical MLWR instance—and the additional side information stemming from the probing of $\mathbf{z}$ and $\mathbf{r}_0$. In this paper, we present an initial cryptanalysis of the π nt-MLWR problem, focusing on its search variant. Our investigation considers two strategies: (1) recovering the permutation using techniques from *Permuted Kernel Problem* (PKP) cryptanalysis, and (2) treating the permuted secret and error vectors as *perfect hints*, in the sense of the *LWE with Side Information* framework [DDGR20]. While preliminary, our analysis suggests only a minor reduction in security relative to the underlying MLWR assumption.

Remark 1. Another work [BAE⁺24] presents a SASCA (Soft Analytical Side-Channel Attack) targeting a shuffled variant of Dilithium. It demonstrates that even a single-bit leakage from the permuted vector $\pi(\mathbf{y})$, when combined with knowledge of $(\mathbf{z}, c)$ —regardless of whether the signature is rejected—can lead to key recovery. While the simplified π nt-MLWR problem might appear related, it differs in several key aspects: the unpermuted vector $\mathbf{z}$ is never revealed in the event of a rejection, and $\pi(\mathbf{y})$ is not exposed when a signature $(\mathbf{z}, c)$ is accepted. Moreover, the security of π nt-MLWR does not rely on the hardness of recovering $(\mathbf{s}_1, \mathbf{s}_2)$, since these values are assumed to be known in the problem formulation. In contrast, SUCRE specifically targets the protection of the rejection sampling process by masking $\mathbf{z}$ and $\mathbf{r}_0$. As such, any leakage occurring on $\mathbf{y}$ prior to the masking phase lies outside the threat model addressed by SUCRE.

Security proof of SUCRE in the d -probing model. Our countermeasure is built around a sequence of $d + 1$ random permutations applied to a shared input, which can be either a reduced $\mathbf{z}$ or $\mathbf{r}_0$. After each permutation step, the number of shares is reduced, and the sharing is refreshed. We formally prove that, under this process, a d -probing adversary can learn at most a uniform permutation of the reduced $\mathbf{z}$ or $\mathbf{r}_0$.

Although the structure of the countermeasure is algorithmically simple, its security proof deviates from classical masking proofs. In particular, for the sake of efficiency, the number of shares decreases at each step. This implies that a probing adversary might observe fully unmasked coefficients of reduced $\mathbf{z}$ or $\mathbf{r}_0$. However, the core idea behind our proof is to show that even in this case, the adversary cannot simultaneously obtain:

- the actual value of an unmasked coefficient *and*
- its original position before the permutation was applied.

The formal proof is carried out using the Non-Interference (NI) framework, which we recall and define in the Section 2. The corresponding simulation argument is naturally divided into two cases, aligned with the scenarios described above. All in all, the masking proof ensures that the only leakage that a probing adversary can get is a permutation of a reduced $\mathbf{z}$ or $\mathbf{r}_0$ with different unknown permutations.

Assessing the security loss due to the leakage. Once we have shown that an adversary may at most learn the value of $\mathbf{w}_1$ and a permutation of a reduced $\mathbf{z}$ or $\mathbf{r}_0$ for rejected signatures, we still have to show the unforgeability of the signature scheme. To take this into account, we recall the analysis of Fiat-Shamir with Aborts from [DFPS23], which requires the HVZK simulator to be able to simulate rejecting transcripts. If this simulation is computationally indistinguishable from real transcripts, this is called strong computationally honest verifier zero-knowledge (sc-HVZK), and their security reduction proves that the signing oracle in the unforgeability game can be simulated without the signing key, and can thus be removed. While in their framework the aborted transcripts are never explicitly given to the adversary, we note that they could be without loss of generality. All that remains is to exhibit such a simulator.

Security proof of SUCRE: simulating the leakage. Based on our previous contributions, we confirm our intuition by providing a sc-HVZK simulator for SUCRE which security relies on the $\pi\text{nt-MLWR}$ assumption. Its high-level idea consists in sampling a uniform $\tilde{\mathbf{z}}$ and applying rejection sampling on it. If it is accepted, then we generate an accepted signature following usual techniques for ML-DSA [KLS18, DFPS23, BBD⁺23]. If it is rejected, we are still able to generate the corresponding leakage, by permuting some reduced $\tilde{\mathbf{z}}$ after generating and permuting the corresponding $\mathbf{r}_0$, but we still have to generate a commitment $\mathbf{w}_1$: we sample it uniformly. Finally, we show that this simulator generates transcripts that are indistinguishable from real transcripts under the $\pi\text{nt-MLWR}$ assumption.

On the masking of $\mathbf{w}_1$ in standard ML-DSA. We take a small detour here to note that our approach generalizes to other types of leakage. Consider for instance the case of (standard) ML-DSA, where only $\mathbf{w}_1$ is not protected against side-channel leakage. The above simulator can be directly adapted, even without using the reduced $\mathbf{z}$, by cutting all leakage related to $\mathbf{z}$ and $\mathbf{r}_0$, except the bit of information indicating whether the signature was rejected or not. However, its sc-HVZK security relies on a variant of MLWR where the adversary gets an extra bit of information stating whether the secret is a nonce that leads to an accepted signature or a rejected one. While we do not provide a reduction from the decision version of MLWR to this variant, we note (see Remark 6) that such a reduction is trivial for the search version by guessing this bit. This provides strong evidence that this

variant is inherently hard, hence supporting the long-standing intuition that $\mathbf{w}_1$ does not require masking.

1.2 Related Work

Protecting lattice-based algorithms against side-channel attacks has been an active research topic in recent years. Since masking strategies are closely tied to the nature of the operations performed by a given algorithm, the protection techniques differ significantly between the different schemes. One of the first high-order masking approach for lattice-based signatures was proposed in [BBE⁺18], targeting the original GLP scheme [GLP12], which later inspired the design of ML-DSA. Higher-order masking for Crystals-Dilithium was first introduced in [MGTF19], though the proposed countermeasure did not cover the entire algorithm. A significant advancement was achieved in [ABC⁺23], where the authors presented a fully masked implementation of ML-DSA, relying on bitslicing techniques to achieve state-of-the-art performance on architectures well suited for this approach. More recently, Coron *et al.* [CGL⁺24] further improved the masking of both the nonce generation and the rejection sampling procedure by introducing a novel technique that eliminates the need for arithmetic-to-Boolean conversion. This conversion refers to the process of transforming an arithmetic masking scheme—where secret shares are combined via modular addition in $\mathbb{Z}_q$ —into a Boolean masking scheme, where the combination of shares relies instead on bitwise XOR operations or, equivalently, additions on $\mathbb{F}_2^k$. Avoiding this conversion significantly reduces the computational overhead. Their work builds on top of the gadget constructions introduced in [CGTZ23]. The masked rejection sampling algorithm from [CGL⁺24] currently offers, to the best of our knowledge, the lowest theoretical complexity among known solutions.

Beyond the question of masking techniques, another line of research has focused on simplifying or bypassing the rejection sampling step altogether. In particular, reducing $\mathbf{z}$ modulo a small integer to make it uniform—and thus easier to simulate—is an idea connected to the results of [BCM21], where the authors sample $\mathbf{y}$ uniformly over the entire space. This allows them to get rid of the rejection condition on $\mathbf{z}$, as even rejected $\mathbf{z}$ can be simulated and integrated in the security proof. However, this comes at the cost of restraining the parameter space to make one of the two security assumptions of ML-DSA statistically intractable. This ends up taking a toll on the scheme efficiency. Another research track investigated the possibility of modifying the core design of ML-DSA in order to end up with another scheme without any rejection sampling [dPPRS23, dPKPR24]. While efficient, this design diverges from the NIST standard that is expected to see widespread adoption soon.

In parallel, another interesting direction explores whether specific components of the signature scheme need to be masked at all. Our attempt to formally justify that, in general, $\mathbf{w}_1$ does not require masking resonates with recent results from [dPN25], where the authors show that for other variants of lattice-based Fiat-Shamir with Aborts signatures, $\mathbf{w}_1$ can be safely unmasked. Their results rely on the fact that these variants benefit from a reduction from the underlying sc-HVZK problem to the standard MLWE problem. In this context, a specific rewriting technique enables getting rid of the extra bit of information, a strategy that unfortunately fails to apply in the MLWR setting considered by ML-DSA.

1.3 Organization

The necessary background on lattice-based assumptions, masking notions, and the description of the ML-DSA scheme are recalled in Section 2. Our new countermeasure, SUCRE, is introduced in Section 3. The formal definition of the π_{nt} -MLWR problem, along with a discussion of its hardness against known cryptanalysis, is provided in Section 4. Section 5

gives our main security theorem, establishing the security of SUCRE within the global ML-DSA framework and the d -probing model. In Section 6, we evaluate the performance of SUCRE under various configurations and compare it to the best known state-of-the-art countermeasures. Finally, Section 7 concludes the paper and outlines potential directions for future research.

1.4 Changes with respect to the CHES version

This version of the article corrects a mistake in the definition of the $\pi\text{nt-MLWR}$ assumption.

Looking ahead, and using notations introduced later in this work and notations from ML-DSA, we found the following issue, which we explain for the “special case” of the assumption. Note that in ML-DSA, for any $x \in [0, q - 1]$, if $\text{LowBits}_q(x, 2\gamma_2) = -\gamma_2$, then it holds that $\text{HighBits}_q(x, 2\gamma_2) = 0$. The original definition of $\pi\text{nt-MLWR}$ was not taking this fact into account, which allowed an adversary to trivially win the decision game when $\pi(e(\mathbf{A}\mathbf{y}))_{-\alpha/2} > \pi(\lfloor \mathbf{u} \rfloor_p)_0$, as this happens only when $\mathbf{u}$ is uniform and not when $\mathbf{u} = \mathbf{A}\mathbf{y}$. This event has non-negligible probability, and we have to take it into account in the definition of the problem.

In this version of our work, we slightly alter the distribution of the uniform high bits, to exclude such weak cases, that do not reflect the hardness of the problem.

We apply this change only when necessary. As our countermeasure does not reveal the low bits when $\mathbf{z}$ is rejected, they are not revealed, and thus the high bits can be uniformly sampled as previously. The necessity for this distinction arises when reducing the sc-HVZK property of SUCRE to the $\pi\text{nt-MLWR}$ assumption.

2 Preliminaries

2.1 Notations

Throughout this paper, λ will always denote a security parameter. A probabilistic algorithm is said *ppt* (probabilistic polynomial time) if it runs in expected polynomial time.

Lattice sets and operations. We let $\mathcal{R}$ (resp. $\mathcal{R}_q$) denote the ring $\mathbb{Z}[x]/(x^n + 1)$ (resp. $\mathbb{Z}_q[x]/(x^n + 1)$) for n a power of two. For any integers r and α , we let $r \bmod^+ \alpha$ (resp. $r \bmod^\pm \alpha$) denote the unique integer r' in $[0, \alpha - 1]$ (resp. $(-\alpha/2, \alpha/2]$) such that $r = r' \bmod \alpha$. Given $\mathbf{z} \in \mathcal{R}^k$ for any integer k , we let $\mathbf{z}[i]$ denote the degree $i \bmod n$ coefficient of the representative of degree less than n of the $\lfloor i/n \rfloor$ polynomial in $\mathbf{z}$. This corresponds to the i -th entry of the concatenation of the coefficient vectors of the polynomials in $\mathbf{z}$. Any ℓ_p norm over $\mathbb{Z}^{nk}$ is extended to $\mathcal{R}^k$ by applying it to the concatenation of the coefficient vectors of the polynomials. We let B_τ denote the set of ternary polynomials having exactly τ nonzero elements. For any integers η, γ , we also define $S_\eta = \{w \in \mathcal{R} \mid \|w\|_\infty \leq \eta\}$ and $\tilde{S}_\gamma = \{w \bmod^\pm 2\gamma \mid w \in \mathcal{R}\}$.

The rounding function $\lfloor \cdot \rfloor : \mathbb{R} \rightarrow \mathbb{Z}$ maps a real x to the unique integer k such that

$$k - \frac{1}{2} \leq x < k + \frac{1}{2}.$$

Given two integers $p < q$, the rounding of $r \in [0, q - 1]$ with respect to p is defined as $\lfloor r \rfloor_p = \lfloor p/q \cdot r \rfloor \bmod p$, and the rounding error is given by $e(r) = r - \lfloor r \rfloor_p \bmod q$.

We define the *frequency mapping* π that on input a vector $\mathbf{y} \in \mathbb{Z}_q^k$ for any $k > 0$ returns a vector $\mathbf{p} \in \mathbb{Z}^q$ such that $\forall i \in \{1, \dots, q\}, p_i = \#\{j \in \{1, \dots, k\} \mid y_j = i\}$, i.e. p_i is the number of times i appears in $\mathbf{y}$.

Distributions. Given any set $\mathcal{X}$, the uniform distribution over this set is denoted $U(\mathcal{X})$. Given a distribution χ over $\mathcal{R}$ and an integer k , we let χ^k denote the distribution over $\mathcal{R}^k$ such that each coefficient is identically and independently distributed following χ . We let $X \leftarrow P$ denote that the random variable X follows the distribution P . We use $\stackrel{\text{id}}{=}$ to refer to the equality of the distributions of random variables.

Masking. We use the general notation of $\llbracket x \rrbracket = (x_0, \dots, x_d) \in \mathbb{Z}_q^{d+1}$ for an arithmetic sharing of a secret value x , and it extends to $\llbracket \mathbf{x} \rrbracket$ when considering the sharings of all the coefficients of $\mathbf{x} \in \mathcal{R}_q$.

2.2 Lattice Assumptions

We recall standard assumptions from lattice-based cryptography.

Definition 1 (Module Learning with Errors). Let n, q, k, ℓ be four integers with n a power of two. Let χ be a distribution over $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$. The advantage of an adversary $\mathcal{A}$ in solving the decisional $\text{MLWE}_{n,q,k,\ell,\chi}$ problem is defined as

$$\text{Adv}_{n,q,k,\ell,\chi}^{\text{MLWE}}(\mathcal{A}) = \left| \Pr(\mathcal{A}(\mathbf{A}, \mathbf{t}) \rightarrow 1 \mid \mathbf{A} \leftarrow U(\mathcal{R}_q^{k \times \ell}), \mathbf{t} \leftarrow U(\mathcal{R}_q^k)) \right. \\ \left. - \Pr(\mathcal{A}(\mathbf{A}, \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2) \rightarrow 1 \mid \mathbf{A} \leftarrow U(\mathcal{R}_q^{k \times \ell}), \mathbf{s}_1 \leftarrow \chi^\ell, \mathbf{s}_2 \leftarrow \chi^k) \right|.$$

Definition 2 (Module Learning with Rounding). Let n, p, q, k, ℓ be five integers with $p < q$ and n a power of two. Let χ be a distribution over $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$. The advantage of an adversary $\mathcal{A}$ in solving the decisional $\text{MLWR}_{n,p,q,k,\ell,\chi}$ problem is defined as

$$\text{Adv}_{n,p,q,k,\ell,\chi}^{\text{MLWR}}(\mathcal{A}) = \left| \Pr(\mathcal{A}(\mathbf{A}, \lfloor \mathbf{t} \rfloor_p) \rightarrow 1 \mid \mathbf{A} \leftarrow U(\mathcal{R}_q^{k \times \ell}), \mathbf{t} \leftarrow U(\mathcal{R}_q^k)) \right. \\ \left. - \Pr(\mathcal{A}(\mathbf{A}, \lfloor \mathbf{A}\mathbf{s} \rfloor_p) \rightarrow 1 \mid \mathbf{A} \leftarrow U(\mathcal{R}_q^{k \times \ell}), \mathbf{s} \leftarrow \chi^\ell) \right|.$$

This problem benefits from search-to-decision reductions as well as reductions to the more standard ring learning with errors problem [LW20].

2.3 Fiat-Shamir with Aborts Signatures

We recall the definition of signature schemes based on the Fiat-Shamir with Aborts framework [Lyu09].

Definition 3. A *Fiat-Shamir with Aborts signature scheme* is a triple of ppt algorithms (KeyGen, Sign, Verify) such that:

- **KeyGen:** on input 1^λ , it outputs a verification key vk and a signing key sk .
- **Sign:** on input a signing key sk and a message μ , it sets $z = \perp$ and does the following until $z \neq \perp$.
 1. Sample a value w according to a distribution D .
 2. Compute $c = H(w, \mu) \in \mathcal{C}$.
 3. Compute z depending on w and c .

It outputs a signature $\sigma = (w, c, z)^1$.

- **Verify:** on input a verification key vk , a signature σ and a message μ , it outputs a bit b , thus accepting or rejecting the signature.

¹In the case of ML-DSA, w is omitted from the signature as it is reconstructed during verification.

The set $\mathcal{C}$ is called the *challenge set*. A triple (w, c, z) is called a *transcript*. It is *rejected* if $z = \perp$, *accepted* otherwise. The min-entropy of D is called the *commitment min-entropy*. The probability that one iteration results in $z = \perp$ is called the *aborting probability*. The signature scheme is called *correct* if for any pair (vk, sk) in the range of KeyGen and any message μ , $\text{Verify}(vk, \text{Sign}(sk, \mu), \mu)$ accepts the signature.

We recall the usual sUF-CMA and UF-NMA security notions for signatures.

Definition 4 (Unforgeability). Let $\Sigma = (\text{KeyGen}, \text{Sign}, \text{Verify})$ be a Fiat-Shamir with Aborts signature. The unforgeability game asks a classical or quantum adversary $\mathcal{A}$ on input vk to produce a forgery (σ^*, μ^*) such that $\text{Verify}(vk, \sigma^*, \mu^*)$ is accepted.

- If $\mathcal{A}$ is given access to a signing oracle, it must output a pair that was not obtained via this oracle and its probability of winning is denoted $\text{Adv}^{\text{sUF-CMA}}(\mathcal{A})$.
- Otherwise, its probability of winning is denoted $\text{Adv}^{\text{UF-NMA}}(\mathcal{A})$.

In both cases, a hash function may be modeled as a random oracle and $\mathcal{A}$ has classical (ROM) or quantum (QROM) access to it.

On top of it, we consider the following two notions.

Definition 5 (sc-HVZK). Let Σ be a Fiat-Shamir with Aborts signature scheme. Let Trans be the ppt algorithm that takes as input sk and c , samples w , computes the corresponding z according to Σ , and returns the resulting transcript (w, c, z) . For any ppt algorithm Sim that takes as input vk and c and returns a transcript (w, c, z) , the advantage of an adversary $\mathcal{A}$ playing the *Strong Computational Honest Verifier Zero Knowledge* (sc-HVZK) game is defined as:

$$\text{Adv}_{\Sigma, \text{Sim}}^{\text{sc-HVZK}}(\mathcal{A}) = \left| \Pr \left(1 \leftarrow \mathcal{A}(vk, \text{Trans}(sk, c)) \right) - \Pr \left(1 \leftarrow \mathcal{A}(vk, \text{Sim}(vk, c)) \right) \right|$$

$$\left\{ \substack{(vk, sk) \leftarrow \text{KeyGen}(1^\lambda) \\ c \leftarrow U(\mathcal{C})} \right\} \quad \left\{ \substack{(vk, sk) \leftarrow \text{KeyGen}(1^\lambda) \\ c \leftarrow U(\mathcal{C})} \right\}$$

Definition 6 (CUR). Let Σ be a Fiat-Shamir with Aborts signature scheme. The *Computational Unique Response* game asks an adversary $\mathcal{A}$, given a verification key vk as input, to find (w, c, z, z') such that $z \neq z'$ and both (w, c, z) and (w, c, z') are valid signatures according to Σ for the verification key vk . We let $\text{Adv}^{\text{CUR}}(\mathcal{A})$ denote the probability of $\mathcal{A}$ being successful.

We give the following adapted formulation of the combination of [DFPS23, Theorem 6] and [DFPS23, Theorem 11], where we take (following their notations) $B = \lambda / \log_2(1/\beta)$ to get the following theorem.

Theorem 1 ([DFPS23, Theorems 6, 11]). Let $\Sigma = (\text{KeyGen}, \text{Sign}, \text{Verify})$ be a Fiat-Shamir with Aborts signature scheme with ω bits of commitment min-entropy and aborting probability $\rho = 1/\text{poly}(\lambda)$. Let Q_s (resp. Q_h) be a maximum amount of signature (random oracle) queries. There exists a bound $B_{\text{ROM}}(Q_s, Q_h) = \text{poly}(Q_s, Q_h)$ (resp. $B_{\text{QROM}}(Q_s, Q_h) = \text{poly}(Q_s, Q_h)$). For any ppt (resp. quantum) adversary $\mathcal{A}$ in the ROM (resp. QROM) against the sUF-CMA security of Σ , there exist three adversaries $\mathcal{B}_0, \mathcal{B}_1$ and $\mathcal{B}_2$ of the same type as $\mathcal{A}$ and running in essentially the same time such that:

$$\text{Adv}_{\Sigma}^{\text{sUF-CMA}}(\mathcal{A}) \leq \frac{\lambda \cdot Q_s}{\log_2(1/\rho)} \text{Adv}^{\text{sc-HVZK}}(\mathcal{B}_0) + \text{Adv}^{\text{CUR}}(\mathcal{B}_1) + 2^{-\omega/C} \cdot B_{(\text{Q})\text{ROM}}(Q_s, Q_h) + \text{Adv}^{\text{UF-NMA}}(\mathcal{B}_2) + \text{negl}(\lambda),$$

where $C = 1$ if $\mathcal{A}$ is classical, $C = 2$ if $\mathcal{A}$ is quantum.

2.4 Masking

2.4.1 Linear Sharing, Circuits, and Gadgets

In the following, the $(d+1)$ -linear decoding mapping, denoted by LinDec , refers to the function $\mathbb{Z}_q^{d+1} \rightarrow \mathbb{Z}_q$ defined as

$$\text{LinDec} : (x_0, \dots, x_d) \mapsto x_0 + \dots + x_d \pmod{q},$$

for any $d \in \mathbb{N}$ and $(x_0, \dots, x_d) \in \mathbb{Z}_q^{d+1}$. We shall further consider that, for every $d, \ell \in \mathbb{N}$, on input $(\llbracket x^0 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket) \in (\mathbb{Z}_q^{d+1})^\ell$ the $(d+1)$ -linear decoding mapping acts as

$$\text{LinDec} : (\llbracket x^0 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket) \mapsto (\text{LinDec}(\llbracket x^0 \rrbracket), \dots, \text{LinDec}(\llbracket x^{\ell-1} \rrbracket)).$$

Definition 7 (Linear Sharing from [BCP⁺20]). Let $d, \ell \in \mathbb{N}$. For any $x \in \mathbb{Z}_q$, an $(d+1)$ -sharing of x is a random vector $\llbracket x \rrbracket \in \mathbb{Z}_q^{d+1}$ such that $\text{LinDec}(\llbracket x \rrbracket) = x$. It is said to be *uniform* if for any set $I \subseteq [0, d]$ with $|I| \leq d$ the tuple $(x_i)_{i \in I}$ is uniformly distributed over $\mathbb{Z}_q^{|I|}$. A $(d+1)$ -linear encoding is a probabilistic algorithm LinEnc which on input a tuple $\mathbf{x} = (x^0, \dots, x^{\ell-1}) \in \mathbb{Z}_q^\ell$ outputs a tuple $\llbracket \mathbf{x} \rrbracket = (\llbracket x^0 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket) \in (\mathbb{Z}_q^{d+1})^\ell$ such that $\llbracket x^i \rrbracket$ is a uniform $(d+1)$ -sharing of x^i for every $i \in [0, \ell-1]$.

An *arithmetic circuit* on a finite field $\mathbb{Z}_q$ is a labeled directed acyclic graph whose edges are *wires* and vertices are *arithmetic gates* processing operations on $\mathbb{Z}_q$. A *randomized arithmetic circuit* includes a random gate that outputs a fresh uniform random value from $\mathbb{K}$. For a randomized circuit C , when we write $C(\rho, \cdot)$, this refers to the corresponding de-randomized circuit with the random tape $\rho \in \{0, 1\}^*$ as input.

In the following, we shall call an $((d+1)$ -share, ℓ -to- m) *gadget*, a randomized arithmetic circuit that maps an input $\llbracket \mathbf{x} \rrbracket \in (\mathbb{Z}_q^{d+1})^\ell$ to an output $\llbracket \mathbf{y} \rrbracket \in (\mathbb{Z}_q^{d+1})^m$ such that $\mathbf{x} = \text{LinDec}(\llbracket \mathbf{x} \rrbracket) \in \mathbb{Z}_q^\ell$ and $\mathbf{y} = \text{LinDec}(\llbracket \mathbf{y} \rrbracket) \in \mathbb{Z}_q^m$ satisfy $\mathbf{y} = g(\mathbf{x})$ for some function g .

2.4.2 Security Notions

A *wire-assignment sampler* takes as input a gadget G , a subset of wire labels W , an input $x \in \mathbb{Z}_q^\ell$, and an optional subset of wire labels Y dedicated to the public output (*i.e.* either output of G or intermediate variables that are revealed) together with an *admissible assignment* of these wires. We say that a $|Y|$ -tuple $(y_1, \dots, y_{|Y|}) \in \mathbb{Z}_q^{|Y|}$ is an admissible assignment of the public output with respect to input x if the probability that the public output be assigned to $(y_1, \dots, y_{|Y|})$ on input x is strictly non-zero (over the internal randomness of the gadget). The wire-assignment sampler outputs a $|W|$ -tuple $(w_1, \dots, w_{|W|}) \in \mathbb{Z}_q^{|W|}$ as

$$(w_1, \dots, w_{|W|}) \leftarrow \text{AssignWires}(G, W, x, Y, \underbrace{(y_1, \dots, y_{|Y|})}_{\text{optional}}),$$

where $w_1, \dots, w_{|W|}$ represent the values assigned to the wires in W during an evaluation of G with input x with uniform internal randomness given that the public output is assigned to $(y_1, \dots, y_{|Y|})$. When the gadget G does not have any public output, we omit the optional input $Y, (y_1, \dots, y_{|Y|})$ (and the internal randomness of G involved in the assignment is truly uniform).

Informally, as introduced in [BBD⁺16], a gadget is said to be d -non-interferent (d -NI) if the distribution of any set of d wires can be perfectly simulated using at most d shares of each input. The notion can be further strengthened: a gadget is said to be d -strong non-interferent (d -SNI) if, for any set of d wires—among which at most $d' \leq d$ are internal wires (*i.e.* wires that are not carrying output shares)—their distribution can be perfectly simulated using only d' shares of each input. A formal definition of both d -NI and d -SNI is given in Definition 8.

Definition 8. Let G be an $((d+1)$ -share, ℓ -to- m) gadget. G is d -non interferent (resp. d -strong non interferent) or d -NI (resp. d -SNI) if for every set W^i of internal wires of G and every set W^o of output wires of G such that $|W| \leq d$, with $W = W^i \cup W^o$, there exists a two-stage simulator $\text{Sim} = (\text{Sim}_1, \text{Sim}_2)$ such that for every input $(\llbracket x^0 \rrbracket, \llbracket x^1 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket) \in (\mathbb{Z}_q^{d+1})^\ell$,

$$\begin{aligned} \text{Sim}_1(G, W) &= (I_0, \dots, I_{\ell-1}) \text{ with } |I_0|, \dots, |I_{\ell-1}| \leq d \quad (\text{resp. } |I_0|, \dots, |I_{\ell-1}| \leq |W^i|), \\ \text{Sim}_2(G, W, ((x_i^0)_{i \in I_0}, \dots, (x_i^{\ell-1})_{i \in I_{\ell-1}})) &\stackrel{id}{=} \text{AssignWires}(G, W, (\llbracket x^0 \rrbracket, \llbracket x^1 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket)). \end{aligned}$$

Extending this notion, as proposed in [BBE⁺18, BBE⁺24], a gadget with public output is d -NIO if the distribution of any set of d wires can be perfectly simulated from the public output and at most d shares of each input. The formal definition is provided in Definition 9.

Definition 9. Let G be a $((d+1)$ -share, ℓ -to- m) gadget and Y be a subset of wires of G . The gadget G is d -non interferent with public output Y (d -NIO) if for every set W of wires of G such that $|W| \leq d$, there exists a two-stage simulator $\text{Sim} = (\text{Sim}_1, \text{Sim}_2)$ such that for all input sharings $(\llbracket x^0 \rrbracket, \llbracket x^1 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket) \in (\mathbb{Z}_q^{d+1})^\ell$ and every admissible assignment of the public output $(y_1, \dots, y_{|Y|}) \in \mathbb{Z}_q^{|Y|}$ with respect to $(\llbracket x^0 \rrbracket, \llbracket x^1 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket)$,

$$\begin{aligned} \text{Sim}_1(G, W, Y) &= (I_0, \dots, I_{\ell-1}) \text{ with } |I_0|, \dots, |I_{\ell-1}| \leq d, \\ \text{Sim}_2(G, W, Y, ((x_i^1)_{i \in I_1}, \dots, (x_i^\ell)_{i \in I_\ell}), (y_1, \dots, y_{|Y|})) &\stackrel{id}{=} \text{AssignWires}(G, W, (\llbracket x^0 \rrbracket, \llbracket x^1 \rrbracket, \dots, \llbracket x^{\ell-1} \rrbracket), Y, (y_1, \dots, y_{|Y|})). \end{aligned}$$

Remark 2. The public output refers to any value that is intentionally left unmasked and accessible to the adversary. This typically includes the final output of the algorithm—such as a signature—whenever it is released in an unmasked form. It can also include any intermediate value that becomes unmasked during the execution of the gadget, even if it is not part of the gadget’s official output.

This distinction is particularly relevant in scenarios such as rejection sampling, where some internal variables may be exposed in their unmasked form under certain conditions, despite not being part of the returned output.

For our general statement (Theorem 3 of Section 5), we use the definition of [BBE⁺24, Def. 1] that captures the fact that no ppt adversary with access to less than d probes on a key generation and signature oracle should be able to break sUF-CMA security. It is presented in the following definition.

Definition 10 (d -probing-sUF-CMA-security [BBE⁺24]). Let us consider a signature scheme $(\text{Ver}, \text{Sign}, \text{KeyGen})$ and $d \geq 1$ an integer. Let Q_s be a fixed maximum amount of signature queries. Let KeyUpdate be an algorithm that refreshes the secret key between signature queries². The signature scheme is d -probing-sUF-CMA-secure if any probabilistic polynomial time adversary has a negligible probability of winning the game presented in Figure 1.

Note that sUF-CMA corresponds to 0-probing-sUF-CMA (the adversary can only probe empty sets). Let us recall a useful and general result from [BBE⁺18, BBE⁺24], which establishes a reduction between two security settings. Informally, it shows that the security of a signature scheme against a d -probing adversary—even when the adversary has access to public outputs (such as unmasked intermediate)—can be reduced to the standard security of the same signature scheme, provided the public outputs are explicitly included in the values returned by the signing and key generation algorithms.

²As [BBE⁺24, Rem. 3], we assume that this algorithm cannot be probed by the attacker.

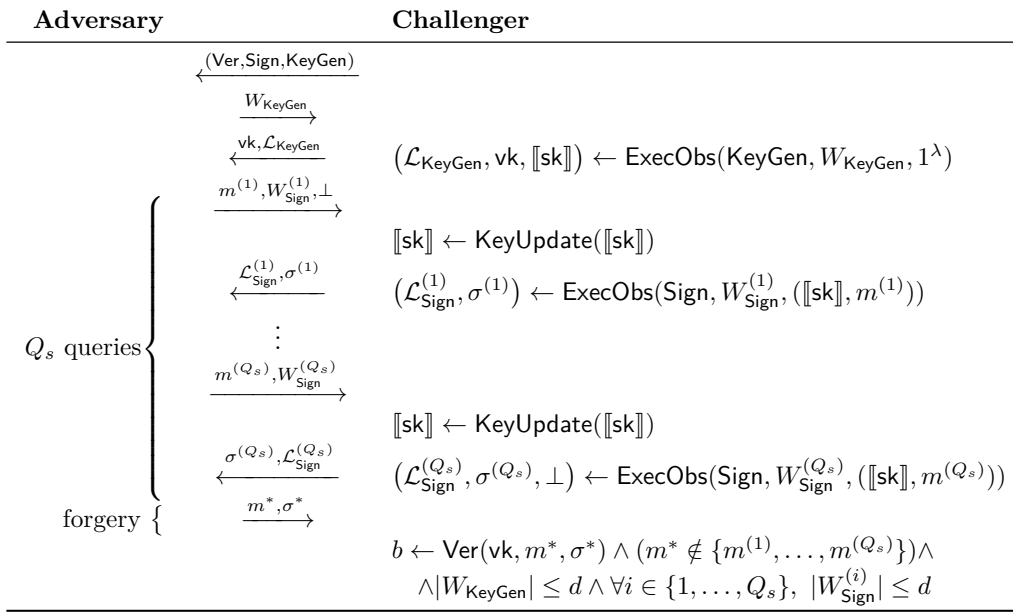


Figure 1: d -probing-sUF-CMA security game. We denote by ExecObs the extension of the AssignWires function that takes the same inputs and outputs a leakage $\mathcal{L} = (w_1, \dots, w_{|W|})$, the values of public outputs y and additionally the values of the masked outputs (if any). The function ExecObs can be easily made of the execution of the gadget G and AssignWires with the same random coins.

Theorem 2 ([BBE⁺24]). *Let d be a masking order. Let us consider a signature scheme $\Sigma = (\text{Ver}, \text{Sign}, \text{KeyGen})$ where Sign and KeyGen are d -NIO for public outputs Y_{Sign} and Y_{KeyGen} . Let $\Sigma' = (\text{Ver}, \text{Sign}', \text{KeyGen}')$ be the same signature scheme where Sign' and KeyGen' include the public outputs as part of their returned values. Let $\mathcal{A}$ be an adversary for the d -probing-sUF-CMA game as defined in Definition 10. Hence, there exists an adversary $\mathcal{B}$ for the plain sUF-CMA game such that*

$$\text{Adv}_{\Sigma}^{d\text{-probing-sUF-CMA}}(\mathcal{A}) = \text{Adv}_{\Sigma'}^{\text{sUF-CMA}}(\mathcal{B}).$$

Proof. This theorem is a straightforward generalization of the proof of [BBE⁺24, Th. 4]. The proof consists in starting from the d -probing-sUF-CMA security game and using the d -NIO property to replace all the AssignWires (inside ExecObs) by the simulator Sim_2 of Definition 8. Next, since the simulated leakage does not provide any advantage to the attacker because it is independent from the unmasked value, the final advantage can be reduced to the advantage for plain sUF-CMA security game. $\square$

Remark 3. Note that the other way around is also straightforwardly true: for any adversary $\mathcal{A}$ for the sUF-CMA game on Σ' , there exists an adversary $\mathcal{B}$ for the d -probing-sUF-CMA game such that $\text{Adv}_{\Sigma}^{d\text{-probing-sUF-CMA}}(\mathcal{B}) = \text{Adv}_{\Sigma'}^{\text{sUF-CMA}}(\mathcal{A})$.

2.5 ML-DSA with Full Verification Key

ML-DSA (formerly known as Crystals-Dilithium [LDK⁺22]) is a lattice-based signature algorithm recently standardized by the NIST in [NIS24a]. It is essentially based on the MLWE problem and is built on the Fiat-Shamir with Aborts [Lyu12] paradigm.

We refer to the official specification [NIS24a] for a description of algorithms `ExpandA`, `ExpandS` and `ExpandMask` which are not needed for the description of our countermeasure. We recall other supporting algorithms for ML-DSA in Figure 2. These algorithms are extended on vectors by applying them coefficient-wise. Their main property is recalled in Lemma 1.

Decompose_q(r, α): 1: $r \leftarrow r \bmod^+ q$ 2: $r_0 \leftarrow r \bmod^\pm \alpha$ 3: if $r - r_0 = q - 1$ then 4: $(r_1, r_0) \leftarrow (0, r_0 - 1)$ 5: else 6: $r_1 \leftarrow (r - r_0)/\alpha$ 7: return (r_1, r_0)	HighBits_q(r, α): 1: $(r_1, r_0) \leftarrow \text{Decompose}_q(r, \alpha)$ 2: return r_1 LowBits_q(r, α): 1: $(r_1, r_0) \leftarrow \text{Decompose}_q(r, \alpha)$ 2: return r_0
--	---

Figure 2: Supporting Algorithms for ML-DSA.

Lemma 1. *Let r, α and q be integers such that $q > 2\alpha, q = 1 \bmod \alpha$ and α is even. Then $\text{HighBits}_q(r, \alpha) \cdot \alpha + \text{LowBits}_q(r, \alpha) = r \bmod q$.*

Simplified pseudocode for ML-DSA is described in Figure 3 and relevant parameters are recalled in Table 2. For simplification, we present the version of ML-DSA without truncation of the verification key. Adding this truncation requires adding an additional check during signing. This check, however, is deterministic and computed only on public data: it only ensures that verification passes. Hence, masking and proving the security of this version of ML-DSA is equivalent to doing the same thing for ML-DSA.

KeyGen(ζ): 1: $(\psi, \psi') \in \leftarrow H(\zeta)$ 2: $\mathbf{A} \in \mathcal{R}_q^{k \times \ell} \leftarrow \text{ExpandA}(\psi')$ 3: $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k \leftarrow \text{ExpandS}(\psi)$ 4: $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 \bmod q$ 5: return $(\text{vk} = (\mathbf{A}, \mathbf{t}), \text{sk} = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$ Verify(vk, M, σ = (z, c)): 1: $\mathbf{w}'_1 = \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t} \bmod q, 2\gamma_2)$ 2: return $(\ \mathbf{z}\ _\infty < \gamma_1 - \beta) \wedge (c = H(M \parallel \mathbf{w}'_1))$	Sign(sk, M, ψ): 1: $\mathbf{z} = \perp$ 2: while $\mathbf{z} = \perp$ do 3: $\mathbf{y} \in \tilde{S}_{\gamma_1}^\ell \leftarrow \text{ExpandMask}(\psi)$ 4: $\mathbf{w} = \mathbf{A}\mathbf{y}$ 5: $\mathbf{w}_1 = \text{HighBits}(\mathbf{w}, 2\gamma_2)$ 6: $c = H(M \parallel \mathbf{w}_1)$ 7: $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ 8: $\mathbf{r}_0 = \text{LowBits}(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$ 9: if $\ \mathbf{z}\ _\infty \geq \gamma_1 - \beta$ then 10: if $\ \mathbf{r}_0\ _\infty \geq \gamma_2 - \beta$ then 11: $\mathbf{z} = \perp$ 12: return $\sigma = (\mathbf{z}, c)$
---	---

Figure 3: Truncation-free ML-DSA pseudocode.

The key generation algorithm consists in generating a random public matrix $\mathbf{A}$, two small secret vectors $(\mathbf{s}_1, \mathbf{s}_2)$, and computing $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ the second part of the public key. On a very high level, signing a message is equivalent to proving the knowledge of a short vector in the lattice. First, a vector $\mathbf{y}$ is sampled to mask the secret vectors that will be used to produce the signature. Then, a challenge c is computed as the hash of the message and the nonce $\mathbf{y}$. Finally, the signature $\mathbf{z}$ is computed as $\mathbf{y} + c\mathbf{s}_1$. However, this signature may not be valid yet. Too large coefficients in $\mathbf{z}$ may reveal information about the secret vector $\mathbf{s}_1$, and thus a final check called rejection sampling is performed. The signing procedure is restarted if $\mathbf{z}$ does not pass the check. As those invalid signatures are

sensitive, the rejection sampling operation must be performed in a most secure way. In the context of SCA, this means that the generated signature must be kept protected until this final check is passed. Signatures are quite often rejected, with an average number of repetitions of 4.25 for ML-DSA-44 (see Table 2), which emphasizes the need for efficient operations in the rejection sampling procedure. It is worth noting that we have omitted, for clarity and simplicity reasons, the details of the deterministic generation of $\mathbf{A}$, $(\mathbf{s}_1, \mathbf{s}_2)$ and $\mathbf{y}$ from seeds as this does not change the description and analysis of our countermeasure.

Table 2: ML-DSA parameters and resulting quantities of interest. All are polynomial in λ . The number of repetitions takes into account aborts due to the truncation, while the aborting probability does not.

Parameters	Constraints	ML-DSA-44	ML-DSA-65	ML-DSA-87
n	Power of two	256	256	256
q	$= 512 \cdot q_0 + 1$, prime	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$	$2^{23} - 2^{13} + 1$
(k, ℓ)		$(4, 4)$	$(6, 5)$	$(8, 7)$
γ_1		2^{17}	2^{19}	2^{19}
γ_2	Even, divides $q - 1$	$(q - 1)/88$	$(q - 1)/32$	$(q - 1)/32$
η	(see below)	2	4	2
τ	(see below)	39	49	60
$\beta = \tau \cdot \eta$	$\leq \min(\gamma_1, \gamma_2)$	78	196	120
Name	Quantity			
/	Repetitions	≈ 4.25	≈ 5.1	≈ 3.85
ρ	Aborting probability	≈ 0.76	≈ 0.80	≈ 0.74
ω	Commitment H_∞	$\gtrsim nk$	$\gtrsim nk$	$\gtrsim nk$

3 Shuffle-and-Unmask Countermeasure

The rejection sampling step of ML-DSA (Step 9 and 10 of Figure 3) involves the manipulation of sensitive data. Specifically, the comparison of the infinity norms of $\mathbf{z}$ and $\mathbf{r}_0$ against a fixed public threshold can be interpreted as a series of independent tests on each coefficient. Coefficients whose absolute values remain below the threshold are uniformly distributed over a centered interval, and thus do not reveal any information. However, when a coefficient exceeds the threshold, it may leak partial information about the underlying secret. A sound countermeasure must therefore ensure that these coefficients are not clearly identified.

In this section, we first recall the most efficient techniques from the literature for implementing the rejection sampling step in a manner that resists d -probing attacks, without compromising security. We then introduce our novel countermeasure, SUCRE, based on a new conjectured hard problem, and present a more efficient variant, SUCRE-no-SP2S, which can be applied to $\mathbf{r}_0$ as is, and could be applied to $\mathbf{z}$ at the cost of relying on a less conservative assumption.

3.1 Previous Approaches

In the following, we present the state-of-the-art approaches to implement a d -probing secure rejection sampling step, and we explain why the naive strategy of shuffling followed by unmasking is not enough in our leakage model.

State-of-the-art. The methods proposed in [BBE⁺18, BBE⁺24] and [ABC⁺23] implement the two norm comparisons (for $\mathbf{z}$ and $\mathbf{r}_0$) through an arithmetic-to-Boolean (A2B) conversion, followed by secure additions and sign bit extraction. The method in [ABC⁺23] improves upon [BBE⁺18, BBE⁺24] by reducing the number of operations and using gadgets designed for bitsliced implementations. The overall complexity of the more efficient approach is that of an A2B conversion combined with a secure addition. In contrast, [CGL⁺24] introduces a new technique for high-order rejection sampling that targets non-bitsliced implementations and achieves a complexity independent of the modulus size q . Their approach only depends on the size β of the rejection interval. It consists of an arithmetic shift of the polynomial coefficients, followed by a zero-test.

A naive initial approach. A natural idea might be to shuffle the masked vector $\llbracket \mathbf{z} \rrbracket$ and then securely unmask it before performing the rejection check in its unmasked form. Such a method could work under a very weak security model in which the attacker cannot probe either the permutation indices or the shares before and after the permutation.

However, our work considers a stronger and more realistic adversarial model in which permutation indices, as well as shares before and after the permutations, are considered as intermediate variables accessible to an attacker. Under this model, a naive shuffle of $\llbracket \mathbf{z} \rrbracket$ prior to unmasking would leak information about the permutation to the attacker, since the permutation indices or the input and output shares could be probed. The knowledge of the original position of the unmasked coefficients of $\mathbf{z}$ would effectively allow the attacker to bypass the rejection sampling for certain coefficients, potentially leading to key recovery attacks.

Our countermeasure, detailed in the next section, addresses this issue by ensuring that the permutation remain masked and cannot be recovered by the attacker throughout the process.

Remark 4. Note that the shuffling traditionally employed in secure implementations serves as an additional layer of practical security (by adding a linear factor to the noise) but is, strictly speaking, not part of the d -probing countermeasure. Indeed, while shuffling complicates side-channel attacks, its presence or absence have no impact on the security parameter d . Probing the permutation indices would merely cancel this beneficial side-effect but would not compromise the claimed d -probing security guarantees. Conversely, in our work, the secrecy of the permutation is crucial: recovering it would directly break the security of the countermeasure.

3.2 Our New Rejection Sampling Gadget

The core idea of our countermeasure is to shuffle the coefficients of the polynomials $\mathbf{z}$ and $\mathbf{r}_0$ prior to unmasking. This shuffling breaks the correlation between a coefficient's position and its value, ensuring that location and content are not leaked simultaneously.

While revealing the shuffled version of $\mathbf{z}$ to the attacker might raise security concerns, we do not operate directly on $\mathbf{z}$. Instead, we apply a modular reduction to $\mathbf{z}$ (more precisely on its sharing $\llbracket \mathbf{z} \rrbracket$), which—surprisingly and beneficially—produces the same output under rejection sampling, as established in Lemma 6. SUCRE is described in Figure 4 and is proven secure in our main Theorem 3. Its two main blocks, the modular reduction SP2Switch, and the secure shuffling-unmasking step, NormCheck, are described in the following subsections.

In contrast to $\mathbf{z}$, the polynomial $\mathbf{r}_0$ does not require modular reduction, as its coefficients are, by construction, already within the rejection threshold. In this case, we rely on a simplified version of our countermeasure, which consists solely of a call to NormCheck for the norm verification with the bound $\gamma_2 - \beta$.

```

SUCRE( $\llbracket \mathbf{z} \rrbracket, \gamma_1 - \beta$ ) :
1: for  $i = 0$  to  $n \cdot \ell$  do
2:    $\llbracket \tilde{\mathbf{z}}[i] \rrbracket \leftarrow \text{SP2Switch}(\log_2(2\gamma_1), \llbracket \mathbf{z}[i] \rrbracket)$ 
3: return NormCheck( $\llbracket \tilde{\mathbf{z}} \rrbracket, \gamma_1 - \beta$ )

```

Figure 4: Our new countermeasure.

We also propose a variant of the countermeasure, that we call **SUCRE-no-SP2S**, which skips the modular reduction of $\mathbf{z}$ (Steps 1 and 2 in Figure 4) and directly checks its infinite norm, thereby improving efficiency. However, it relies on a more unconventional hardness assumption, which, although less standard, we argue remains computationally challenging in practice. In the case of $\mathbf{r}_0$, **SUCRE-no-SP2S** coincides with the original (more conservative) countermeasure, as no modular reduction is required.

The main advantage of our new proposal (and its variant) lies in its significantly improved complexity compared to previous approaches, thanks to lighter operations. A detailed complexity analysis is presented in Section 3.2.3, and concrete performance benchmarks are provided in Section 6. However, proving the security of our scheme is non-trivial. This applies both to the **sUF-CMA** security—where we must ensure that the introduction of the shuffling does not inadvertently strengthen the adversary—and to the d -NIo (non-interference with public outputs) security, given the tight control over the number of shares at each step and the impact of the uniformly random permutation. These aspects are rigorously analyzed in Section 5.

3.2.1 The SP2Switch Gadget

In this section, we introduce our **SP2Switch** gadget that turns an arithmetic sharing of $x \bmod q$ with $|x| < q/4$ into an arithmetic sharing of $x \bmod 2^\theta$, with $2^\theta < q$. This gadget relies on a simple modification of the **LMSwitch** gadget from [CGL⁺24].

Let $\llbracket x \rrbracket = (x_i)_{i=0}^d$ be an arithmetic sharing of x modulo $q = 2^\theta \cdot p + k$. We show how to turn it into an arithmetic sharing of $x \bmod 2^\theta$ by adapting the **LMSwitch** gadget from [CGL⁺24]. First, we note that over $\mathbb{Z}$, there exists δ such that

$$\sum_{i=0}^d x_i = x + \delta \cdot q = x + k\delta + (\delta \cdot p)2^\theta.$$

Following the same strategy as in the **LMSwitch** gadget, we compute an arithmetic sharing $\llbracket \delta \rrbracket = (\delta_i)_{i=0}^d$ of δ modulo 2^θ , which (by definition) satisfies:

$$\sum_{i=0}^d \delta_i = \delta \bmod 2^\theta.$$

All that remains is to note that

$$x = \sum_{i=0}^d (x_i - k\delta_i) \bmod 2^\theta.$$

We present in Figure 5 the resulting gadget **SP2Switch** (Smaller Power of 2 Switch), where **RefreshQuasiLin** is the quasi-linear refresh gadget from [BCPZ16, MM21] and **ShiftMod _{α}** is taken from [CGT23]. This construction is proven to be d -NI secure as stated in Lemma 3 in Section on the masking security (Sec. 5.1).

```

SP2Switch( $\theta, \llbracket x \rrbracket$ ) :
1:  $\alpha \leftarrow \lceil \log_2(d+1) \rceil + 1$ 
2: for  $i = 0$  to  $d$  do
3:    $y_i \leftarrow \lfloor x_i \cdot 2^\alpha / q \rfloor$ 
4:    $y_1 \leftarrow y_1 + d + 2^{\alpha-2} \bmod 2^{\theta+\alpha}$ 
5:    $\llbracket \delta \rrbracket \leftarrow \text{ShiftMod}_\alpha(2^{\theta+\alpha}, \llbracket y \rrbracket)$ 
6:    $\llbracket \delta \rrbracket \leftarrow \text{RefreshQuasiLin}(\llbracket \delta \rrbracket)$ 
7: for  $i = 0$  to  $d$  do
8:    $v_i \leftarrow x_i - (q \bmod 2^\theta) \delta_i \bmod 2^\theta$ 
9: return  $\llbracket v \rrbracket$ 

```

Figure 5: The SP2Switch gadget.

3.2.2 The NormCheck Gadget

For both countermeasures, a key technical challenge lies in computing the shuffling and the unmasking securely, in a manner that aligns with the masking scheme employed throughout the algorithm. To this end, algorithm **NormCheck** in Figure 6 introduces an iterative permutation mechanism, combined with a mask refreshing at each step. For clarity, the algorithm is described for $\mathbf{z}$, but it applies identically to $\mathbf{r}_0$ after adjusting for its size and the correct integer bound. This construction is proven to be d -NLo secure, even when the permuted version of $\mathbf{z}$ (or $\mathbf{r}_0$) is publicly revealed, as shown in Lemma 4 in Section on the masking security (Sec. 5.1).

NormCheck($\llbracket \mathbf{z} \rrbracket, B$): Require: $\llbracket \mathbf{z} \rrbracket = (\mathbf{z}_0, \mathbf{z}_1, \dots, \mathbf{z}_d) \in (\mathcal{R}_q^\ell)^{d+1}$, $B \in \mathbb{N}$ Ensure: true if $\ \mathbf{z}\ _\infty < B$, false otherwise <pre> 1: for $i = 0$ to $d - 1$ do 2: $\text{Perm}((\mathbf{z}_0, \mathbf{z}_1, \dots, \mathbf{z}_{d-i}), d - i)$ 3: $\text{RefreshLin}((\mathbf{z}_0, \mathbf{z}_1, \dots, \mathbf{z}_{d-i}), d - i)$ 4: $\mathbf{z}_{d-i-1} \leftarrow \mathbf{z}_{d-i-1} + \mathbf{z}_{d-i}$ $\triangleright$ <i>Permute and check</i> 5: for $j = 0$ to $n \cdot \ell - 1$ do 6: $r \leftarrow U([j, n \cdot \ell - 1])$ 7: if $\mathbf{z}_0[r] \geq B$ then 8: $\mathbf{z}_0[r] \leftarrow \mathbf{z}_0[j]$ 9: $\mathbf{z}_0[r] \leftarrow \mathbf{z}_0[j]$ 10: return true </pre>	Perm($\llbracket \mathbf{z} \rrbracket, d$): <pre> 1: for $j = 0$ to $n \cdot \ell - 1$ do 2: $r \leftarrow U([j, n \cdot \ell - 1])$ 3: for $k = 0$ to d do 4: $\text{tmp} \leftarrow \mathbf{z}_k[r]$ 5: $\mathbf{z}_k[r] \leftarrow \mathbf{z}_k[j]$ 6: $\mathbf{z}_k[j] \leftarrow \text{tmp}$ </pre> $\triangleleft$ RefreshLin($\llbracket \mathbf{z} \rrbracket, d$): <pre> 1: for $k = 0$ to $d - 1$ do 2: $\mathbf{r} \leftarrow U(\mathbb{Z}_q^{n \times \ell})$ 3: $\mathbf{z}_k \leftarrow \mathbf{z}_k + \mathbf{r}$ 4: $\mathbf{z}_d \leftarrow \mathbf{z}_d - \mathbf{r}$ </pre>
--	---

Figure 6: Shuffle-and-unmask countermeasure for rejection sampling (SUCRE). The algorithm is depicted for an input $\llbracket \mathbf{z} \rrbracket \in (\mathcal{R}_q^\ell)^{d+1}$ without modulus switching (*i.e.* without a previous call to SP2Switch). It can be trivially adapted to a input sharing $\llbracket \tilde{\mathbf{z}} \rrbracket$ modulo 2^θ , by considering $\mathbb{Z}_{2^\theta}^{n \times \ell}$ instead of $\mathbb{Z}_q^{n \times \ell}$ as vector space in the RefreshLin gadget as well as for the addition in line 4 of NormCheck.

3.2.3 Complexity

To evaluate the complexity of the countermeasure and enable a fair comparison with the state-of-the-art, Table 3 and Table 4 present the complexity—both in terms of randomness and computational operations—of the various components of algorithms in Figures 5 and 6. For simplicity, we assign the same cost to all randomness samplings, although generating permutation indices could be more efficient than sampling elements from $\mathbb{Z}_q$ since $n \cdot \ell \ll q$.

Table 3: Complexity analysis of the main components in SP2Switch (Figure 5), expressed in terms of randomness and additions.

Operation (Line)	Total Cost
ShiftMod $_{\alpha}$ (Line 5) [CGL ⁺ 24]	$d(d+1)$ randomness $\alpha \cdot (2(d+1)^2 + 10(d+1) - 9)$ additions with $\alpha = \lceil \log_2(d+1) \rceil + 1$
RefreshQuasiLin (Line 6) [MM21]	$\frac{(d+1)}{2} \log_2(d+1)$ randomness $(d+1) \log_2(d+1)$ additions
Total (SP2Switch)	$d(d+1) + \frac{(d+1)}{2} \log_2(d+1)$ randomness $\alpha \cdot (2d^2 + 14d + 3) + (d+1) \log_2(d+1) + 4d + 6$ additions with $\alpha = \lceil \log_2(d+1) \rceil + 1$

Table 4: Complexity analysis of the main components of NormCheck (Figure 6), expressed in terms of randomness, additions, memory accesses (read/write), and comparisons.

Operation (Line)	Cost per Iteration	Cost
Permutation (Line 2)	$(n\ell - 1)$ randomness $2n\ell(d - i + 1)$ read/write	$(d+1)(n\ell - 1)$ randomness $\frac{3}{2}n\ell(d+1)(d+2)$ read/write
Linear Refresh (Line 3)	$(d - i)n\ell$ randomness $2(d - i)n\ell$ additions	$\frac{1}{2}d(d+1)n\ell$ randomness $d(d+1)n\ell$ additions
Vector Addition (Line 4)	$n\ell$ additions	$dn\ell$ additions
Rejection Sampling (Line 7)	$n\ell$ comparisons	$n\ell$ comparisons
Total (NormCheck) (= SUCRE-no-SP2S)	–	$(d+1) \left(\frac{(d+2)n\ell}{2} - 1 \right)$ randomness $\frac{3}{2}(d+1)(d+2)n\ell$ read/write $d(d+2)n\ell$ additions $n\ell$ comparisons

We first observe that SP2Switch has almost the same complexity as LMSwitch gadget from [CGL⁺24] from which it is inspired. The complexity of NormCheck (Figure 6) is of $O(d^2)$ operations, with d the order of masking, implying a global $O(d^2)$ complexity for SUCRE-no-SP2S. To compute the complexity of SUCRE, we must add the SP2Switch complexity, which is $O(d^2 \log_2 d)$. The resulting complexity of SUCRE may appear at first sight very similar to the complexity of RejectSampling algorithm from [CGL⁺24], as the latter is also $O(d^2 \log_2 d)$.³ However, we offer significant improvement in the constant factors, as it can be seen in Table 5. As both algorithms SP2Switch and LMSwitch have similar (and non-trivial) complexity, we also present in Table 4 the complexity of RejectSampling from [CGL⁺24] without LMSwitch for fair (and easier) comparison with SUCRE-no-SP2S. Note that we took [CGL⁺24] complexity for $\rho = 8$, using the notation from the article, corresponding to the rejection sampling of $\mathbf{z}$. We do not provide in Table 5 a detailed comparison with [ABC⁺23] as their complexity is $O(d^2 \log_2 q)$, but we discuss their practical results in Section 6.

³Although this does not drastically change the overall complexity comparison, we have slightly optimized the original LMSwitch and RejectSampling by using a quasi-linear RefreshQuasiLin gadget in place of the quadratic RefreshMasks from [CGL⁺24], holding the same SNI property.

Sample_u($q, p, \alpha, \eta, c, \mathbf{r}$): 1: if $\mathbf{r} = \perp$ then 2: $\mathbf{u}_0 \leftarrow U(\mathcal{R}_q^k)$ 3: $\mathbf{u} \leftarrow \lfloor \mathbf{u}_0 \rfloor_p$ 4: else 5: $\mathbf{s}'_2 \leftarrow U(S_\eta^k)$ 6: for $i = 1$ to nk do 7: if $r_i = -\alpha/2$ then 8: $u_i \leftarrow \lfloor q - (\alpha/2) + (c\mathbf{s}'_2)_i \rfloor_p$ 9: else 10: $u_i \leftarrow U(\{0, \dots, p-1\})$ 11: $\mathbf{u} \leftarrow (\sum_{j=0}^{n-1} u_{j+ni} x^j)_{i=0}^k$ 12: return $\mathbf{u}$	Sample_r($\mathbf{y}, c, \mathbf{s}_1, \mathbf{s}_2, q, \beta, \gamma_1$): 1: if $\ \mathbf{y} + c\mathbf{s}_1\ _\infty \geq \gamma_1 - \beta$ then 2: $\mathbf{r} \leftarrow \perp$ 3: else 4: $\mathbf{r} \leftarrow e(\mathbf{A}\mathbf{y} - c\mathbf{s}_2 \bmod q)$ 5: return $\mathbf{r}$
--	---

Figure 7: Sampling algorithm for $\mathbf{u}$ and $\mathbf{r}$. The hint $\mathbf{r}$ is set to $\perp$ when $\mathbf{z}$ is rejected, as it is not leaked. When $\mathbf{r} \neq \perp$, we sample a dummy $\mathbf{s}'_2$ so that when $v_i = -\alpha/2$, u_i is the high bits of $v_i + c\mathbf{s}'_2 \bmod q$.

Table 5: Comparison of SUCRE and SUCRE-no-SP2S with [CGL⁺24].

Algorithm	Cost (operations)
SUCRE	$(\alpha \cdot (2d^2 + 14d + 3) + \frac{5}{2}d^2 + (d+1)\log_2(d+1) + \frac{21}{2}d + 10) \cdot n\ell$ with $\alpha = \lceil \log_2(d+1) \rceil + 1$
[CGL ⁺ 24]	$[\alpha \cdot (2d^2 + 14d + 3) + \frac{79}{2}d^2 + 2(d+1)\log_2(d+1) + \frac{479}{2}d + 57] \cdot n\ell$ with $\alpha = \lceil \log_2(d+1) \rceil + 1$
SUCRE-no-SP2S	$(\frac{5}{2}d^2 + \frac{13}{2}d + 4) \cdot n\ell$
[CGL ⁺ 24]-no-LMSwitch	$[\frac{79}{2}d^2 + (d+1)\log_2(d+1) + \frac{471}{2}d + 52] \cdot n\ell$

4 Permuted Hint Module Learning with Rounding

We defer the security analysis of our countermeasure in Section 5, where we show that in the d -probing model, there is no other leak on rejected signatures than $\mathbf{w}_1$, the permuted $\mathbf{z}$ and the permuted $\mathbf{r}_0$ (Theorem 3). We also adapt the security proof of ML-DSA, to show not only that $\mathbf{w}_1$ is a MLWR instance, but also that the leaked permutations on $\mathbf{z}$ and $\mathbf{r}_0$ are *hints* on this instance.

But first, in this section, we prepare ourselves by introducing the MLWR problem with leaked permutations, which we dub $\pi\text{nt-MLWR}$ and we underline how cryptanalysis fails to exploit the hints, due to the hardness of either unraveling the permutation or finding statistical correlation between the hints and $\mathbf{w}_1$.

4.1 Definition

We define a variant of MLWR with a leak on the secret and the “noise”, due to the leak of the permuted rejected transcripts in our approach of masking. Note that in this context, the secret is $\mathbf{y}$.

Definition 11 ($\pi\text{nt-MLWR}$). Let $n, q, k, \ell, \tau, \eta, \gamma$ be seven non-negative integers with n a power of two. Let α be an integer satisfying $q > 2\alpha, q = 1 \bmod \alpha$ and $\alpha = 0 \bmod 2$. Let $p = (q-1)/\alpha$.

For any adversary $\mathcal{A}$, we define the advantage of $\mathcal{A}$ in distinguishing the $\pi\text{nt-MLWR}$ distribution with parameters $(n, q, \alpha, k, \ell, \gamma, \eta, \tau)$ with the uniform distribution as:

$$\text{Adv}_{n,q,\alpha,k,\ell,\gamma,\eta,\tau}^{\pi\text{nt-MLWR}}(\mathcal{A}) = \left| \Pr_{\substack{\mathbf{A} \leftarrow U(\mathcal{R}_q^{k \times \ell}) \\ \mathbf{y} \leftarrow U(\tilde{S}_\gamma^\ell) \\ (\mathbf{s}_1, \mathbf{s}_2) \leftarrow U(S_\eta^\ell \times S_\eta^k) \\ c \leftarrow B_\tau}} \left(1 \leftarrow \mathcal{A} \left(\begin{array}{c} \mathbf{A}, \lfloor \mathbf{A}\mathbf{y} \rfloor_p, \\ \pi(\mathbf{y} + c\mathbf{s}_1 \bmod 2\gamma), \\ \pi(\text{Sample}_r(\mathbf{y}, c, \mathbf{s}_1, \mathbf{s}_2, q, \eta\tau, \gamma)), \\ (\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2), c \end{array} \right) \right) \right. \\ \left. - \Pr_{\substack{\mathbf{A} \leftarrow U(\mathcal{R}_q^{k \times \ell}) \\ \mathbf{y} \leftarrow U(\tilde{S}_\gamma^\ell) \\ c \leftarrow B_\tau \\ (\mathbf{s}_1, \mathbf{s}_2) \leftarrow U(S_\eta^\ell \times S_\eta^k) \\ \mathbf{r} \leftarrow \text{Sample}_r(\mathbf{y}, c, \mathbf{s}_1, \mathbf{s}_2, q, \eta, \tau, \gamma)}} \left(1 \leftarrow \mathcal{A} \left(\begin{array}{c} \mathbf{A}, \text{Sample}_u(q, p, \alpha, \eta, c, \mathbf{r}), \\ \pi(\mathbf{y} + c\mathbf{s}_1 \bmod 2\gamma), \pi(\mathbf{r}), \\ (\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2), c \end{array} \right) \right) \right|,$$

where $\lfloor \cdot \rfloor_p$ is the rounding operator with respect to p (modulo q), $\mathbf{e}(\cdot)$ is the rounding error with respect to p (modulo q) and π is the frequency mapping (see Section 2.1). Finally, the sampling algorithm Sample_u is defined in Figure 7.

When $\eta = 0$, we drop η and τ from the subscript, and do not give $\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ and c to the adversary $\mathcal{A}$. We call it the *special case*. In this case, note that Sample_r always returns $\mathbf{e}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2)$.

We expect the generic case to be at least as hard as the special case. Indeed, the permuted hints in the generic case follow the same distribution as the permuted hints in the special case, while being less correlated with $\mathbf{y}$.

Remark 5. The problem is not conditioned on $\mathbf{z}$ being rejected. Indeed, this data is actually computable from the permuted hints: $\mathbf{z}$ is rejected if and only if their infinity norm is larger than a threshold. This gives an immediate reduction, which, due the rejection probability being $\approx 3/4$ in ML-DSA, loses at most one bit of security. Moreover, in the security reduction, we actually reduce the security of SUCRE to this exact problem, without the rejection on $\mathbf{z}$.

4.2 Exploratory Cryptanalysis of $\pi\text{nt-MLWR}$

We focus on the special case of Definition 11. By definition of the error function $\mathbf{e}$, we can rewrite the equation as $\mathbf{A}\mathbf{y} - \mathbf{e}(\mathbf{A}\mathbf{y}) = \lfloor \mathbf{A}\mathbf{y} \rfloor_p \cdot 2\gamma_2 \bmod q$, which is equivalent to finding a particular solution of:

$$(\mathbf{A} \mid \mathbf{I}) \cdot \begin{pmatrix} \mathbf{u} \\ \mathbf{e} \end{pmatrix} = \lfloor \mathbf{A}\mathbf{y} \rfloor_p \cdot 2\gamma_2 \bmod q.$$

where the vector $\mathbf{u}$ is sampled uniformly in $[-\gamma_1, \gamma_1]^{\ell \cdot n}$ and $\mathbf{e}$ lies in $[-\gamma_2, \gamma_2]^{k \cdot n}$. Since $q^{k \cdot n} \ll (2\gamma_1)^{\ell \cdot n} \cdot (2\gamma_2)^{k \cdot n}$, such a solution is non-unique with overwhelming probability. Therefore, this can be viewed as an instance of the MSIS problem in the ℓ_∞ norm. Notice that security estimates for similar MSIS parameters are already considered from ML-DSA⁴.

A natural question then arises:

Does knowledge of the permutation reduce the hardness of this problem?

We consider two approaches for exploiting this additional information to recover $\mathbf{y}$:

- **Case 1:** Directly recover the permutation $\pi = (\pi_1, \pi_2)$.

⁴<https://github.com/pq-crystals/security-estimates>

- **Case 2:** Leverage auxiliary information from $\pi_1(\mathbf{y})$ and $\pi_2(\mathbf{e}(\mathbf{A}\mathbf{y}))$, such as the coordinate sums of $\mathbf{y}$ and $\mathbf{e}(\mathbf{A}\mathbf{y})$, to recover a valid MSIS solution using lattice reduction techniques.

In the first case, the naive exhaustive search for the permutation $\pi = (\pi_1, \pi_2)$ has a prohibitive cost of $(\ell \cdot n)!(k \cdot n)!$, making it impractical in all reasonable settings. However, this can be improved by leveraging algorithms that solve the *(Inhomogeneous) Permuted Kernel Problem (PKP)*:

Definition 12 (Inhomogeneous Permuted Kernel Problem (iPKP)). Let (q, m, n) be positive integers such that $m < n$, $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, and $(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}_q^m \times \mathbb{Z}_q^n$. Let π be a permutation such that $\mathbf{A} \cdot \pi(\mathbf{x}) = \mathbf{y}$. Given $(\mathbf{A}, \mathbf{x}, \mathbf{y})$ and assuming $\mathbf{x} \neq \mathbf{0}$, the goal of iPKP is to recover π or $\mathbf{x}$.

While algorithmic approaches (see, for example, [KMP19]) offer improvements over brute-force search, they remain inefficient compared to directly solving an (M)SIS instance. For small parameters such as $(n = 128, m = 64, q = 8380417)$, the best known attacks already have a complexity of approximately 2^{256} , and this cost increases exponentially with the dimension. The complexity of such attacks can be estimated using the `PKEstimator` tool.⁵

Interestingly, we can estimate the number of solutions to the π nt-MLWR problem using the same approach as for PKP: $(\ell \cdot n)! \cdot (k \cdot n)! / q^{k \cdot n} \ll 1$. This implies that, under these parameters, the permuted vectors $\pi_1(\mathbf{y})$ and $\pi_2(\mathbf{e}(\mathbf{A}\mathbf{y}))$ uniquely determine $\mathbf{y}$ with overwhelming probability.

In the second case, we leverage knowledge of the permuted vectors by intersecting the SIS lattice embedding with a subspace that captures this partial information—specifically, the coordinate sums. Following the approach of [DDGR20, DGHK23], we consider the classical lattice embedding used to solve SIS and LWE instances, defined by the lattice $\Lambda_{\mathbf{t}}^\perp(\mathbf{A} \mid \mathbf{I})$, with target $\mathbf{t} = \lfloor \mathbf{A}\mathbf{y} \rfloor_p \cdot 2\gamma_2 \in \mathbb{Z}_q^{k \cdot n}$:

$$\Lambda_{\mathbf{t}}^\perp(\mathbf{A} \mid \mathbf{I}) = \begin{pmatrix} -\mathbf{I}_{\ell n} & 0 & 0 \\ \mathbf{A} & q \cdot \mathbf{I}_{kn} & \mathbf{t} \\ 0 & 0 & 1 \end{pmatrix} \cdot \mathbb{Z}^{(k+\ell)n+1}.$$

To incorporate partial knowledge from the permuted vectors $\pi_1(\mathbf{y})$ and $\pi_2(\mathbf{e}(\mathbf{A}\mathbf{y})) = \pi_2(\mathbf{e})$, we intersect this lattice with the hyperplane defined by the known inner products involving $\mathbf{y}$ and $\mathbf{e}$. This yields a subspace $F \subset \mathbb{Z}^{(k+\ell)n+1}$ of dimension $(k + \ell)n - 1$, reflecting the additional linear constraints imposed by the known coordinate sums of $\mathbf{y}$ and $\mathbf{e}$:

$$F = \left\{ \mathbf{u} \in \mathbb{Z}^{(k+\ell)n+1} \left| \sum_{i=0}^{\ell n-1} u_i = \sum_{i=0}^{\ell n-1} \pi_1(y_i) \wedge \sum_{i=\ell n}^{(k+\ell)n} u_i = \sum_{i=\ell n}^{(k+\ell)n} \pi_2(e_i) \right. \right\}.$$

Lemma 2 ([Mar03, (Proposition 1.1.3)]). Let Λ be a lattice in a Euclidean space E , and let $F \subset E$ be a subspace of dimension m . Then $\Lambda' = \Lambda \cap F$ is a relative lattice in F , and is a full-rank lattice in F if and only if F contains m linearly independent vectors of Λ .

As a result, the problem of finding a short vector in $\Lambda_{\mathbf{t}}^\perp(\mathbf{A} \mid \mathbf{I})$ reduces to finding a short vector in the sublattice $\Lambda_{\mathbf{t}}^\perp(\mathbf{A} \mid \mathbf{I}) \cap F$, while accounting for the altered distribution of $(\mathbf{y}, \mathbf{e}(\mathbf{A}\mathbf{y}))$. This reduction in dimension yields only a marginal improvement in the effectiveness of lattice reduction algorithms such as BKZ. Nevertheless, the resulting reduced basis can then be used to recover candidate solutions for $\mathbf{y}$.

In the context of SUCRE, this translates to a marginal reduction in security relative to ML-DSA. However, this does not fully offset the additional complexity involved in

⁵<https://github.com/Crypto-TII/CryptographicEstimators>

identifying the correct solution $\mathbf{y}$ among the many SIS solutions. Thus, in practice, we expect that this attack will offer no gain.

5 Security Reduction for SUCRE

In this section, we discuss the security of SUCRE first in terms of resistance against a d -probing attacker, then by reducing its security to the π nt-MLWR problem.

We aim to prove the security of our countermeasure against the d -probing-sUF-CMA attacker. This model relevantly combines the black-box security with a signature oracle with a grey-box model where the adversary can query intermediate variables during the execution. The main theorem is now stated below. It combines the results that will be presented in the sequel (Sections 5.1 and 5.2). Our work only focuses on a subpart of ML-DSA. Thus, for the rest of the algorithm, we focus on the following hypothesis.

Hypothesis 1. *Let d be a masking order. Let us assume the existence of a masked-ML-DSA, an instance of ML-DSA with the parameters of Table 2 which is made of the secure composition of d -NI and d -NIO gadgets in order to achieve d -probing-sUF-CMA-security. More precisely, masked-ML-DSA is such that for any adversary $\mathcal{A}$ there is an adversary $\mathcal{A}'$ such that*

$$\text{Adv}_{\text{masked-ML-DSA}}^{d\text{-probing-sUF-CMA}}(\mathcal{A}) \leq \text{Adv}_{\text{ML-DSA}}^{\text{sUF-CMA}}(\mathcal{A}').$$

For example, a masked ML-DSA implementation with a masked $\mathbf{w}_1$, a masked computation of the challenge and a masked rejection sampling may validate Hypothesis 1. We will use it as a framework where we can include our countermeasure.

Theorem 3 (Main theorem). *Let d be a masking order. Let Q_s (resp. Q_h) be a maximum amount of signature (random oracle) queries. Let $n, q, k, \ell, \gamma_1, \gamma_2, \eta, \tau, \beta = \tau\eta$ be nine integers satisfying the constraints from Table 2 and ρ and ω be the corresponding aborting probability and commitment min-entropy. We assume the existence of masked-ML-DSA validating Hypothesis 1. We now modify masked-ML-DSA into an instance denoted SUCRE-ML-DSA for which the Sign implementation (in Figure 3)*

1. *unmasks $\mathbf{w}_1$ in order to compute the challenge c without masking,*
2. *and uses our new gadget SUCRE on the $(d+1)$ -sharing of $\mathbf{z}$ and SUCRE-no-SP2S on the $(d+1)$ -sharing of $\mathbf{r}_0$ for the rejection sampling step.*

Let $\mathcal{A}$ be a ppt (resp. quantum) adversary for the d -probing-sUF-CMA game (Def. 10) for SUCRE-ML-DSA in the ROM (resp. QROM), there exists three adversaries $\mathcal{B}_0, \mathcal{B}_1$ and $\mathcal{B}_2$ such that

$$\begin{aligned} \text{Adv}_{\text{SUCRE-ML-DSA}}^{d\text{-probing-sUF-CMA}}(\mathcal{A}) &\leq \frac{\lambda \cdot Q_s}{\log_2(1/\rho)} \overbrace{\text{Adv}_{n,q,\frac{q-1}{2},k,\ell,\gamma_1,S_{\eta},\tau}^{\pi\text{nt-MLWR}}(\mathcal{B}_0)}^{\text{our new hypothesis (Def. 11)}} + \text{Adv}^{\text{CUR}}(\mathcal{B}_1) \\ &\quad + 2^{-\omega/C} \cdot B_{(\text{Q})\text{ROM}}(Q_s, Q_h) + \text{Adv}^{\text{UF-NMA}}(\mathcal{B}_2) + \text{negl}(\lambda), \end{aligned}$$

where $C = 2$ if $\mathcal{A}$ is quantum in the QROM, $C = 1$ otherwise.

Proof. We first note that masked-ML-DSA may have other internal unmasked variables (like bits of information on conditional branches) that should be considered as public outputs. Hence, we define ML-DSA* as an instance of ML-DSA that includes all the public outputs of masked-ML-DSA in its returned values. By using Hypothesis 1 and Remark 3, for any $\mathcal{A}_0$ there exists $\mathcal{A}_1$ and $\mathcal{A}_2$ such that

$$\text{Adv}_{\text{ML-DSA}^*}^{\text{sUF-CMA}}(\mathcal{A}_0) \stackrel{\text{R. 3}}{=} \text{Adv}_{\text{masked-ML-DSA}}^{d\text{-probing-sUF-CMA}}(\mathcal{A}_1) \stackrel{\text{Hyp.1}}{\leq} \text{Adv}_{\text{ML-DSA}}^{\text{sUF-CMA}}(\mathcal{A}_2). \quad (1)$$

We now define another intermediate unmasked signature algorithm, denoted π -ML-DSA, that performs exactly as ML-DSA in black-box except that it outputs the value of $\mathbf{w}_1$ and the permuted $\mathbf{z}_0$ for each call to NormCheck as part of its returned values (hence outputting both permuted reduced $\mathbf{z}$ and the permuted $\mathbf{r}_0$). By extension, we also define π -ML-DSA* that outputs both the public outputs of ML-DSA* and the ones of π -ML-DSA. Next, we use Corollary 1, whose statement and proof are the purpose of Section 5.1 and Theorem 2 above to get the existence of an adversary $\mathcal{C}$ such that:

$$\text{Adv}_{\text{SUCRE-ML-DSA}}^{d\text{-probing-sUF-CMA}}(\mathcal{A}) \stackrel{\text{Th. 2}}{=} \text{Adv}_{\pi\text{-ML-DSA}^*}^{\text{sUF-CMA}}(\mathcal{C}).$$

The inequality from Equation 1 states that the knowledge of the public outputs of ML-DSA* does not provide any extra advantage to the attacker. Hence, it implies that we can remove them from the security game. In other words, there exists an adversary $\mathcal{D}$ such that

$$\text{Adv}_{\pi\text{-ML-DSA}^*}^{\text{sUF-CMA}}(\mathcal{C}) \leq \text{Adv}_{\pi\text{-ML-DSA}}^{\text{sUF-CMA}}(\mathcal{D}).$$

First, we note that the CUR property and UF-NMA security of π -ML-DSA are identical to that of ML-DSA, as they do not give a signature oracle to the adversary. Let us use Theorem 1 to link $\text{Adv}_{\pi\text{-ML-DSA}}^{\text{sUF-CMA}}(\mathcal{D})$ to an adversary of the sc-HVZK game and next use Theorem 4 that bounds the sc-HVZK advantage of π -ML-DSA. The last theorem is the purpose of Section 5.2. Therefore, we obtain the required three adversaries $\mathcal{B}_0, \mathcal{B}_1$ and $\mathcal{B}_2$ and the coefficient C such that

$$\begin{aligned} \text{Adv}_{\text{SUCRE-ML-DSA}}^{d\text{-probing-sUF-CMA}}(\mathcal{A}) &\leq \text{Adv}_{\pi\text{-ML-DSA}}^{\text{sUF-CMA}}(\mathcal{D}) \stackrel{\text{Th. 1}}{\leq} \frac{\lambda \cdot Q_s}{\log_2(1/\rho)} \overbrace{\text{Adv}_{n,q,\frac{q-1}{2\gamma_2},k,\ell,\gamma_1,S_\eta,\tau}^{\pi\text{nt-MLWR}}(\mathcal{B}_0)}^{\text{Th. 4}} + \\ &\quad \text{Adv}^{\text{CUR}}(\mathcal{B}_1) + 2^{-\omega/C} \cdot B_{\text{ROM}}(Q_s, Q_h) + \\ &\quad \text{Adv}^{\text{UF-NMA}}(\mathcal{B}_2) + \text{negl}(\lambda). \end{aligned}$$

□

5.1 Proofs of Probing Security

In this section, we prove (or recall) the non-interference of our two main gadgets, namely SP2Switch and NormCheck.

Lemma 3 (Non interference property of SP2Switch). *The SP2Switch gadget, defined in Figure 5 is d -NI.*

Proof. As shown in [CGL⁺24, Th. 6], the security property follows from the composition of RefreshQuasiLin which is d -SNI (as proven in [MM21] and deduced from [BCRT23, Cor. 2, 3, and 5]) and ShiftMod _{α} which is d -NI (as proved in [CGTZ23, Th. 1]). □

Lemma 4 (Non interference property of NormCheck). *The NormCheck gadget, defined in Figure 6 is d -NIO, when considering the final $\mathbf{z}_0$ in Step 10 as a public output.*

Proof. First, to avoid any confusion with intermediate variables, the final $\mathbf{z}_0$ will be denoted $\mathbf{z}_{\text{out}}$ and the shared input of the algorithm will be denoted $\llbracket \mathbf{z}_{\text{in}} \rrbracket$. Let us fix the notations for each intermediate variable. For an iteration $i \in [0, d+1]$, note that each bold variable has $n\ell$ coefficients. The coefficients are identified with index $j \in [0, n\ell-1]$ and the shares are identified with index $k \in [0, d-i]$.

Possible probes for the i -th iteration:

- $\mathbf{z}_0, \mathbf{z}_1, \dots, \mathbf{z}_{d-i}$, input of Perm (each share has $n\ell$ coefficients on $\mathbb{Z}_q$),
- $r_0, r_1, \dots, r_{n\ell-1}$ during Perm,

- $\mathbf{z}_0^\sigma, \mathbf{z}_1^\sigma, \dots, \mathbf{z}_{d-i}^\sigma$, input of RefreshLin (each share has $n\ell$ coefficients on $\mathbb{Z}_q$),
- $\mathbf{r}'_0, \mathbf{r}'_1, \dots, \mathbf{r}'_{d-i-1}$ during RefreshLin (each share has $n\ell$ coefficients on $\mathbb{Z}_q$),
- $\mathbf{z}_0^{\text{fresh}}, \mathbf{z}_1^{\text{fresh}}, \dots, \mathbf{z}_{d-i}^{\text{fresh}}$, the outputs of RefreshLin (each share has $n\ell$ coefficients on $\mathbb{Z}_q$),
- $\forall t \in [0, d-i] \quad \mathbf{z}_{d-i}^{t-\text{fresh}} = \mathbf{z}_{d-i}^\sigma - \sum_{u=0}^{t-1} \mathbf{r}'_u$ during RefreshLin (each value has $n\ell$ coefficients on $\mathbb{Z}_q$),

Let $\mathcal{P}$ be a set of at most d probes of the gadget. We consider that probes can be uniquely identified with a label and indexes. More precisely, a probe can be a variable $\mathbf{z}$, $\mathbf{r}'$, $\mathbf{z}^\sigma$, $\mathbf{z}^{\text{fresh}}$ associated with indexes i, j, k for the iteration, the coefficient and the share index. A probe can also be a random r associated with indexes i, j for the iteration and the coefficient. It can also be a partial sum $\mathbf{z}_{d-i}^{t-\text{fresh}}$ associated with indexes i, j for the iteration and coefficient and index $t \in [1, d-i]$.

We will show that $\mathcal{P}$ can be simulated with at most d shares of the input $\llbracket \mathbf{z}_{\text{in}} \rrbracket$ and with the public output $\mathbf{z}_{\text{out}}$.

Simulation of the full permutation. By design, the distribution of the permutation values (corresponding to intermediate variables r) is independent from the distribution of the other intermediate values and from the inputs/outputs of the algorithm. Therefore, to ease the simulation process, we assume that the simulation will first draw the full permutation (instead of drawing only the probed elements). More precisely, all the r are drawn following the permutation distribution for all possible indexes i, j and they are stored as local variables. Thus, any probe of $\mathcal{P}$ that concerns the permutation can be taken from this step and each relative permutation is known by the simulation.

Next, we divide $\mathcal{P}$ into $d+1$ disjoint sets $\mathcal{P}_i$ corresponding to the probes of each iteration (with iteration d corresponding to the instructions from Steps 5 to 9), *i.e.* $\mathcal{P} = \cup_{i=0}^d \mathcal{P}_i$. As there are $d+1$ iterations and $|\mathcal{P}| \leq d$, there exists at least one iteration i such that $\mathcal{P}_i = \emptyset$. We denote by $i_\emptyset$ the smallest index $i \in [0, d]$ such that $\mathcal{P}_{i_\emptyset} = \emptyset$.

Fact 1. *The distribution of $\mathcal{P}^{(0)} := \cup_{i=0}^{i_\emptyset-1} \mathcal{P}_i$ can be perfectly simulated with at most d shares of the input of the algorithm $\llbracket \mathbf{z}_{\text{in}} \rrbracket$ and without the public output $\mathbf{z}_{\text{out}}$.*

Fact 2. *The distribution of $\mathcal{P}^{(1)} := \cup_{i=i_\emptyset+1}^d \mathcal{P}_i$ can be perfectly simulated with the public output $\mathbf{z}_{\text{out}}$ and without any share of the input of the algorithm $\llbracket \mathbf{z}_{\text{in}} \rrbracket$.*

Both facts allow to conclude the proof of Lemma 4 because $\mathcal{P} = \mathcal{P}^{(0)} \cup \mathcal{P}^{(1)}$.

Proof of Fact 1. Let $\mathcal{I}$ be the set of indexes of the shares of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$ that are necessary to simulate $\mathcal{P}^{(0)}$. We aim at proving $|\mathcal{I}| \leq d$. Let $0 \leq i < i_\emptyset$ and let us consider the probes in $\mathcal{P}_i$. By definition, $|\mathcal{P}_i| \geq 1$ for all $i \in [0, i_\emptyset - 1]$. Therefore one can deduce $|\mathcal{P}^{(0)}| \geq i_\emptyset$. Hence, since $d \geq |\mathcal{P}^{(0)}|$,

$$\forall i \in [0, i_\emptyset - 1] \quad 1 \leq |\mathcal{P}_i| \leq d - i_\emptyset + 1 \leq d - i. \quad (2)$$

We will now show that each probe in $\mathcal{P}_i$ that depend on the first $d-i$ shares can be simulated from at most one share of each input and each probe that depends on the last $(d-i)^{\text{th}}$ share may depend on $i+1$ shares, but only if $\mathcal{P}$ contains i probes from the previous iterations that also depends on these same $i+1$ shares. With the above upper bound, it will be enough to conclude that $\mathcal{P}^{(0)}$ can be simulated only from at most d shares of each input.

Any probe in $\mathcal{P}_i$ on a random variable r' of the refresh gadget is drawn uniformly at random by the simulation. Any probe on a random variable r used for the permutation

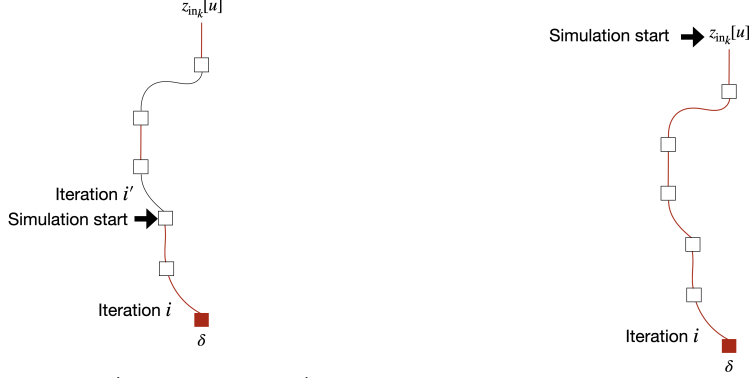
is simulated from the permutation that was drawn in the first step and stored as local variable. For any probe $\delta \in \mathcal{P}_i$ on $\mathbf{z}_{d-i}^{t-\text{fresh}}$, with j be the corresponding coefficient, k the corresponding share index, and t number of terms in the partial sum, the simulation will either require uniform random or it will require to simulate the corresponding $\mathbf{z}^\sigma$. So let us focus on a probe $\delta \in \mathcal{P}_i$ on $\mathbf{z}$, $\mathbf{z}^\sigma$ or $\mathbf{z}^{\text{fresh}}$, where j is the corresponding coefficient and k the corresponding share index.

1. Let us first consider the simple case where $k < d - i$ (excluding the last share that has a special dependency). We will show that the probe δ can be perfectly simulated with at most one share of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$. Indeed, the simulation first recovers the relative permutation by computing $u \in [0, n\ell - 1]$ such that $j = r^i[\dots r^1[r^0[u]] \dots]$ with the previously drawn permutation values. We will distinguish two subcases⁶:
 - (a) there exists $i' \leq i$, such that the distribution of $r'_k[j]^{i'}$ conditioned to the values of other probes in $\mathcal{P}^{(0)}$ is uniform. In other words, at iteration i' , the refresh value for share index k and coefficient j is left unobserved and cannot be deduced from other probes. In that case, δ can be simulated without any input share. The simulation will simulate the value of the corresponding share at iteration i' with uniform random and consistently drawing the simulation in between iteration i' and iteration i . If the refresh values were previously drawn in the simulation (for example for simulating other probes in $\mathcal{P}_{i''}$ for $i'' < i$), they are simply reused.
 - (b) $\forall i' \leq i$ either $r'_k[j]^{i'} \in \mathcal{P}_{i'}$ or $r'_k[j]^{i'}$ can be deduced from other probes. In that case, the index (k, u) (meaning the share index k and coefficient index u) is added in $\mathcal{I}$. The simulation can then simulate the probe's distribution by drawing all the refresh values in the path from $\mathbf{z}_{\text{in}_k}[u]$ to δ . Similarly, if the refresh values were previously drawn in the simulation, they are simply reused.
2. Now assume that the probe δ is on the last share $k = d - i$. This case is more complex as the last share is the sum of shares $d - i$ and $d + 1 - i$ of iteration $i - 1$. Going recursively to the input, the last share is a sum of $i + 1$ terms where each term depends on at most one share of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$. Defining a dependency path as the set of all the refreshes from a share of the input to δ , there are at least $i + 1$ dependency paths from $\llbracket \mathbf{z}_{\text{in}} \rrbracket$ to δ . Each path contributes to the sum defining δ . For each path, the two previous cases apply:
 - (a) either one refresh in the path is not in $\mathcal{P}^{(0)}$ (and cannot be deduced). In that case, the contribution of this path in the sum defining δ can be simulated without any input share exactly as for case (a) above with $k < d - i$.
 - (b) or all the refresh values can be deduced from $\mathcal{P}^{(0)}$ and the simulation of the contribution of this path to δ will require the corresponding share and coefficient of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$. In that case, the index (k, u) (meaning the share index k of coefficient u) is added in $\mathcal{I}$. The contribution of this path in the sum defining δ can be simulated with the corresponding input share exactly as for case (b) above with $k < d - i$.

We now introduce two macro-subcases adapted to the context of the sum of shares.

- (A) All the dependency paths are in case (a). In that case, δ can be simulated without any input share of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$.

⁶In fact, we could restrict ourselves to the second case and always add (k, u) to $\mathcal{I}$, which would already suffice for our proof. However, we chose to present both cases explicitly to make the simulation strategy clearer and to highlight how each probe can be more tightly simulated, by minimizing the number of required input shares.



(a) The refresh value r' of one iteration i' cannot be deduced from $\mathcal{P}$. The simulation will start the simulation by drawing the corresponding share and coefficient of $\mathbf{z}$ at iteration i' . The simulation does not require any share of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$ for simulating δ . (b) All the refresh values can be deduced from $\mathcal{P}$. The simulation will start the simulation by using the corresponding share and coefficient of the input $\llbracket \mathbf{z}_{\text{in}} \rrbracket$.

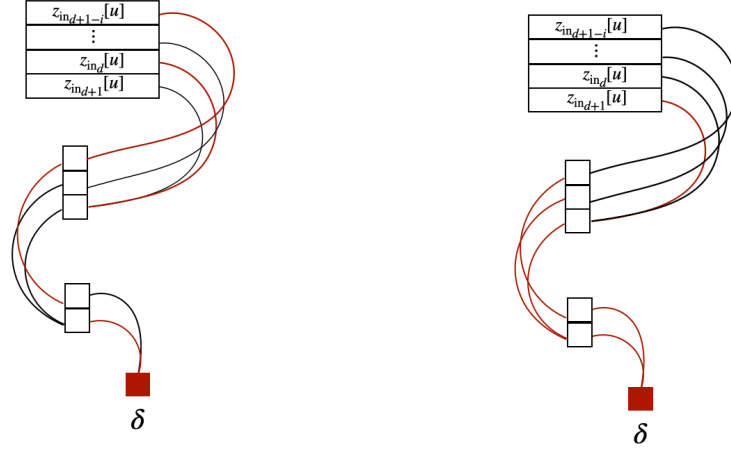
Figure 8: Illustration of the simulation depending on the subcases for $k < d - i$. The squared nodes correspond to values of $\mathbf{z}$ or $\mathbf{z}^{\text{fresh}}$, the edges correspond to the permutation and addition of r' . The red parts correspond to elements in $\mathcal{P}$. When an edge is red, we assume that at least $r' \in \mathcal{P}$ (the permutation index r can also possibly be in $\mathcal{P}$).

- (B) At least one path is in case (b). We will show that there exists a subset $\mathcal{S} \subset \mathcal{P}^{(0)}$ containing δ , such that $|\mathcal{S}| \geq i + 1$ that can be perfectly simulated with at most $i + 1$ shares of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$. For the path in (b), all r' in the path can be deduced from $\mathcal{P}^{(0)}$. As the distribution of all r' is uniform, it implies that there are at least i probes in $\mathcal{P}^{(0)}$ that can be simulated from uniform random. We define $\mathcal{S}$ as the set of all the probes that can be simulated with uniform random for each path in case (b). Hence, $|\mathcal{S}| \geq i$. As δ may be simulated with at most $i + 1$ shares of $\mathbf{z}_{\text{in}}$, adding δ in $\mathcal{S}$, we have $|\mathcal{S}| \geq i + 1$ and $\mathcal{S}$ can be simulated with at most $i + 1$ shares of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$ and uniform random.

We have shown that for each $i < i_0$, each probe in $\mathcal{P}_i$ can be simulated from at most one share of each input for the ones that depend on the first $d - i$ shares, and any probe that depends on the last $(d - i)^{\text{th}}$ share may depend on $i + 1$ shares, but only if $\mathcal{P}$ contains i probes from the previous iterations that also depends on these same $i + 1$ shares, which concludes the proof of Fact 1.

Proof of Fact 2. The set of probes in $\mathcal{P}^{(1)}$ depends on the input of iteration $i_0 + 1$. As the linear refresh of iteration i_0 is left fully unobserved (by definition of i_0), any strict subset of shares of the inputs of iteration $i_0 + 1$ is independent from the input shares of $\llbracket \mathbf{z}_{\text{in}} \rrbracket$. As the permutation of i_0 is left fully unobserved (by definition of i_0), the location of the coefficients is uniformly random at iteration $i_0 + 1$. All the probes in $\mathcal{P}^{(1)}$ can be simulated as follows. For all coefficient indexes $j \in [0, n\ell - 1]$,

- the shares $\mathbf{z}_k^{i_0+1}[j]$ for $k \in [0, d - i_0 - 2]$ are drawn uniformly at random.
- the simulation computes $u \in [0, n\ell - 1]$ such that $u = r^{d+1}[\dots r^{i_0+2}[r^{i_0+1}[j]] \dots]$ with the previously drawn permutation values. Next, the $(d - i_0 - 1)$ -th share of



(A) All paths from input shares to δ contain a black edge. For each path, the simulation will draw the contribution to δ without any input share as in subcase (a) above.

(B) There is a path from input share $\mathbf{z}_{\text{in},d+1}$ to δ that contain only red edges. The simulation will need the value of $\mathbf{z}_{\text{in},d+1}$ for the simulation.

Figure 9: Illustration of the simulation depending on the subcases for $k = d - i$. The squared nodes correspond to values of $\mathbf{z}$ or $\mathbf{z}^{\text{fresh}}$, the edges correspond to the permutation and addition of r' . The red parts correspond to elements in $\mathcal{P}$. When an edge is red, we assume that at least $r' \in \mathcal{P}$ (the permutation index r can also possibly be in $\mathcal{P}$).

$\mathbf{z}_k^{i_\emptyset+1}[j]$ is drawn as

$$\mathbf{z}_{d-i_\emptyset-1}^{i_\emptyset+1}[j] = \mathbf{z}_{\text{out}}[u] - \sum_{k=0}^{d-i_\emptyset-2} \mathbf{z}_k^{i_\emptyset+1}[j].$$

Now the complete input of iteration $i_\emptyset + 1$ is simulated, allowing to execute the algorithm in order to get simulations for any probe in $\mathcal{P}^{(1)}$.

□

Corollary 1. *Let us consider a d -NIO $(d+1)$ -shared gadget which performs steps 3 to 8 of Sign from Figure 3, with public output c . Then, the Sign implementation made of the composition of this gadget with SUCRE applied on the $(d+1)$ -sharing of $\mathbf{z}$ and SUCRE-no-SP2S applied on the $(d+1)$ -sharing of $\mathbf{r}_0$ is d -NIO with public outputs c and a uniform and random permutation of $\mathbf{z}$.*

Proof. The proof directly builds on the d -NIO properties SUCRE and SUCRE-no-SP2S, established in Lemmas 3 and 4. The properties of [BBD⁺16] then are enough to conclude with the sequential compositions of d -NIO gadgets. □

Of course, Sign with our more efficient variant, when we omit the modular reduction on $\mathbf{z}$ (applying directly SUCRE-no-SP2S) is also d -NIO with public outputs c and a uniform and random permutation of $\mathbf{z}$.

5.2 A sc-HVZK Simulator for SUCRE

In this section, for simplicity, we assume that the leakage corresponds to the frequency mapping π defined in Section 2.1 instead of a random permutation. This is strictly equivalent: for any permutation matrix $\mathbf{P}$, we have $\pi(\mathbf{P}\mathbf{z}) = \pi(\mathbf{z})$, and choosing a random

preimage of $\pi(\mathbf{z})$ to apply a random permutation to is equivalent to applying a random permutation to $\mathbf{z}$.

The following lemma will be useful in showing that the assumption our simulator relies on is the hardness of MLWR.

Lemma 5. *Let q, α be integers satisfying $q > 2\alpha$, $q - 1 = 0 \bmod \alpha$ and $\alpha = 0 \bmod 2$. We have:*

$$\text{HighBits}_q(r, \alpha) = \left\lfloor \frac{r-1}{\alpha} \right\rfloor \bmod^+ \frac{q-1}{\alpha} = \lfloor r \rfloor_{\frac{q-1}{\alpha}}, \forall r \in [0, q-1].$$

Moreover, $\mathbf{e}(r) = \text{LowBits}_q(r, \alpha)$.

We postpone proving this lemma to Appendix A. Our fundamental trick to simulating rejected transcripts lies in the following lemma.

Lemma 6 (Distribution of $\mathbf{z} \bmod \pm 2\gamma_1$). *Let $n, q, k, \ell, \gamma_1, \gamma_2, \eta, \tau, \beta = \tau\eta$ be nine integers satisfying the constraints from Table 2. Let $\mathbf{s}_1 \in S_\eta^\ell$ and $c \in B_\tau$. Let $\mathbf{y} \leftarrow U(\tilde{S}_{\gamma_1}^\ell)$ and set $\mathbf{z} \leftarrow \mathbf{y} + \mathbf{s}_1 c$ and $\tilde{\mathbf{z}} = \mathbf{z} \bmod \pm 2\gamma_1$. Then:*

- $\tilde{\mathbf{z}}$ follows the uniform distribution $U(\tilde{S}_{\gamma_1}^\ell)$ over $\tilde{S}_{\gamma_1}^\ell$,
- $\|\tilde{\mathbf{z}}\|_\infty < \gamma_1 - \beta \Leftrightarrow \|\mathbf{z}\|_\infty < \gamma_1 - \beta$.

Proof. First, note that $\mathbf{y} \leftarrow U(\tilde{S}_{\gamma_1}^\ell)$ and $\tilde{\mathbf{z}} = \mathbf{y} + \mathbf{s}_1 c \bmod \pm 2\gamma_1$, hence it is also uniformly distributed over $\tilde{S}_{\gamma_1}^\ell$.

The equivalence holds by noting that:

- If $\|\mathbf{z}\|_\infty < \gamma_1 - \beta < \gamma_1$, then $\mathbf{z} = \tilde{\mathbf{z}}$, and then we also have $\|\tilde{\mathbf{z}}\|_\infty < \gamma_1 - \beta$.
- If $\|\tilde{\mathbf{z}}\|_\infty \leq \gamma_1 - \beta$, then recall that $\mathbf{z} = \mathbf{y} + \mathbf{s}_1 c$ and we show that actually $\mathbf{z} = \tilde{\mathbf{z}}$. For any coordinate z_i of $\mathbf{z}$, we have $|z_i| \leq \gamma_1 + \beta$. It implies that $z_i \leq \gamma_1 + \beta$. Assuming that $|z_i| > \gamma_1$ for some i , we assume without loss of generality that $z_i > \gamma_1$ and we have that the corresponding coordinate of $\tilde{\mathbf{z}}$ is $\tilde{z}_i = z_i - 2\gamma_1$, which is such that $\tilde{z}_i \leq \gamma_1 + \beta - 2\gamma_1 \leq -\gamma_1 + \beta$. Hence we end up with $|\tilde{z}_i| = -\tilde{z}_i \geq \gamma_1 - \beta$ as $\gamma_1 > \beta$. This contradicts $\|\tilde{\mathbf{z}}\|_\infty < \gamma_1 - \beta$ and shows that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$.

This concludes the proof. $\square$

We are now ready to assess the sUF-CMA security of π -ML-DSA. As previously introduced in the proof of Theorem 3, π -ML-DSA is a modified version of ML-DSA that performs exactly as ML-DSA in black-box except that it outputs the value of $\mathbf{w}_1$ and the permuted $\mathbf{z}_0$ for each call to NormCheck as part of its returned values (hence outputting both permuted reduced $\mathbf{z}$ and the permuted $\mathbf{r}_0$).

Let us note that its CUR property and UF-NMA security are identical to that of ML-DSA, as they do not give a signature oracle to the adversary. All that remains is to provide a sc-HVZK simulator that could be included in the general security bound with Theorem 1. The sc-HVZK simulator should satisfy the following.

- It decides between generating an accepting transcript or a rejected one with the same probability as ML-DSA.
- In case of rejected transcript, it must provide $(\mathbf{w}_1, c, \pi(\tilde{\mathbf{z}}), \pi(\mathbf{r}_0))$ (the last one may be omitted if $\tilde{\mathbf{z}}$ is directly rejected).
- In case of accepted transcript, it must provide $(c, \mathbf{z})$, as $\mathbf{w}_1$ can be recomputed from these two elements.

$\text{Sim}(\text{vk}, c) :$	$\text{Real}(\text{vk}, \text{s}_1, c) :$
1: $\tilde{\mathbf{z}} \leftarrow U(\tilde{S}_{\gamma_1})$	1: $\mathbf{y} \leftarrow U(\tilde{S}_{\gamma_1})$
2: if $\ \tilde{\mathbf{z}}\ _\infty < \gamma_1 - \beta$ then	2: $\mathbf{w} \leftarrow \mathbf{A}\mathbf{y} \bmod q$
3: $\mathbf{z} \leftarrow \tilde{\mathbf{z}}$	3: $\mathbf{w}_1 \leftarrow \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$
4: $\mathbf{r}_0 \leftarrow \text{LowBits}_q(\mathbf{A}\mathbf{z} - \text{ct}, 2\gamma_2)$	4: $\mathbf{z} \leftarrow \mathbf{y} + \text{s}_1 c$
5: if $\ \mathbf{r}_0\ _\infty < \gamma_2 - \beta$ then	5: $\tilde{\mathbf{z}} \leftarrow \mathbf{z} \bmod \pm 2\gamma_1$
6: return $(c, \mathbf{z})$	6: if $\ \mathbf{z}\ _\infty < \gamma_1 - \beta$ then
7: $\mathbf{w}_1 \leftarrow \text{Sample}_u(q, \frac{q-1}{2\gamma_2}, 2\gamma_2, \eta, \mathbf{r}_0)$	7: $\mathbf{r}_0 \leftarrow \text{LowBits}_q(\mathbf{A}\mathbf{z} - \text{ct}, 2\gamma_2)$
8: return $(\mathbf{w}_1, c, \pi(\tilde{\mathbf{z}}), \pi(\mathbf{r}_0))$	8: if $\ \mathbf{r}_0\ _\infty < \gamma_2 - \beta$ then
9: $\mathbf{w} \leftarrow U(\mathcal{R}_q^k)$	9: return $(c, \mathbf{z})$
10: $\mathbf{w}_1 \leftarrow \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$	10: else
11: return $(\mathbf{w}_1, c, \pi(\tilde{\mathbf{z}}))$	11: return $(\mathbf{w}_1, c, \pi(\tilde{\mathbf{z}}), \pi(\mathbf{r}_0))$
	12: return $(\mathbf{w}_1, c, \pi(\tilde{\mathbf{z}}))$

Figure 10: Transcript generator for π -ML-DSA. Simulated on the left, real transcripts on the right. Sample_u is defined in Figure 7.

- The distribution of simulated transcripts must be computationally indistinguishable from the distribution of real transcripts (i.e. leakage and signatures).

Theorem 4 (schVZK). *Let $n, q, k, \ell, \gamma_1, \gamma_2, \eta, \tau, \beta = \tau\eta$ be nine integers satisfying the constraints from Table 2. For any adversary $\mathcal{A}$ distinguishing between the output of either of the algorithms of Figure 10, i.e. $\text{Sim}(\text{vk}, c)$ or $\text{Real}(\text{vk}, \text{s}_1, c)$, there exists an adversary $\mathcal{B}$ such that the following holds:*

$$\left| \Pr_{\text{vk}, c, \text{s}_1} (\mathcal{A}(\text{Real}(\text{vk}, \text{s}_1, c)) \rightarrow 1) - \Pr_{\text{vk}, c, \text{s}_1} (\mathcal{A}(\text{Sim}(\text{vk}, c)) \rightarrow 1) \right| \leq \text{Adv}_{n, q, \frac{q-1}{2\gamma_2}, k, \ell, \gamma_1, S_\eta, \tau}^{\pi\text{nt-MLWR}}(\mathcal{B}).$$

Proof. We first introduce an hybrid simulator Hyb in Figure 11.

$\text{Hyb}(\text{vk}, \text{s}_1, c) :$
1: $\mathbf{y} \leftarrow U(\tilde{S}_{\gamma_1})$
2: $\mathbf{z} \leftarrow \mathbf{y} + \text{s}_1 c$
3: $\tilde{\mathbf{z}} \leftarrow \mathbf{z} \bmod \pm 2\gamma_1$
4: if $\ \mathbf{z}\ _\infty < \gamma_1 - \beta$ then
5: $\mathbf{r}_0 \leftarrow \text{LowBits}_q(\mathbf{A}\mathbf{z} - \text{ct}, 2\gamma_2)$
6: if $\ \mathbf{r}_0\ _\infty < \gamma_2 - \beta$ then
7: return $(c, \mathbf{z})$
8: $\mathbf{w}_1 \leftarrow \text{Sample}_u(q, \frac{q-1}{2\gamma_2}, 2\gamma_2, \eta, \mathbf{r}_0)$
9: return $(\mathbf{w}_1, c, \pi(\tilde{\mathbf{z}}), \pi(\mathbf{r}_0))$
10: $\mathbf{w} \leftarrow U(\mathcal{R}_q^k)$
11: $\mathbf{w}_1 \leftarrow \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$
12: return $(\mathbf{w}_1, c, \pi(\tilde{\mathbf{z}}))$

Figure 11: Hybrid transcript generator. The only change with respect to Real is the distribution of $\mathbf{w}_1$ in case of rejection.

Real versus hybrid. We build the following reduction $\mathcal{B}$ that calls $\mathcal{A}$ to solve the $\pi\text{nt-MLWR}$ problem. Let $\mathbf{A}, \mathbf{w}_1, \mathbf{z}', \mathbf{r}'_0, \mathbf{t}, c$ be a $\pi\text{nt-MLWR}$ instance.

The reduction $\mathcal{B}$ does the following.

Accepted. If $\|\mathbf{z}'\|_\infty < \gamma_1 - \beta$ and $\|\mathbf{r}'_0\|_\infty < \gamma_2 - \beta$, $\mathcal{B}$ returns **unif** with probability $1/2$, else it returns **mlwr**.

Rejected on $\mathbf{r}_0$. Otherwise if $\|\mathbf{z}'\|_\infty < \gamma_1 - \beta$ set $\mathbf{trans} = (\mathbf{w}_1, c, \mathbf{z}')$.

Rejected on $\mathbf{z}'$. Else set $\mathbf{trans} = (\mathbf{w}_1, c, \mathbf{z}', \mathbf{r}'_0)$.

Call $\mathcal{A}$ on $\mathbf{trans}$. In the last two cases, when $\mathcal{A}$ outputs **Hyb**, $\mathcal{B}$ returns **unif**. Otherwise, when $\mathcal{A}$ outputs **Real**, $\mathcal{B}$ returns **mlwr**. We note that

- As shown in Lemma 5, this algorithm outputs something that either follows the output distribution of **Hyb** if $\mathbf{w}_1$ is sampled according to Sample_u , or the output distribution of **Real** if $\mathbf{w}_1$ is a rounding of some $\mathbf{A}\mathbf{y}$.
- In the case of an accepting transcript (*i.e.* of the form $(c, \mathbf{z})$), the output distribution of the above algorithm is independent of $\mathbf{w}_1$, hence conditioned on this event, $\mathcal{A}$ has distinguishing advantage 0, which is why $\mathcal{B}$ aborts by returning a random answer.
- In the case of a rejected transcript, $\mathcal{B}$ is right if and only if $\mathcal{A}$ is right. Their distinguishing advantages, conditioned on this event, are identical.

Hence, we have:

$$\left| \Pr_{\mathbf{vk}, c, \mathbf{s}_1} (\mathcal{A}(\text{Real}(\mathbf{vk}, \mathbf{s}_1, c)) \rightarrow 1) - \Pr_{\mathbf{vk}, c, \mathbf{s}_1} (\mathcal{A}(\text{Hyb}(\mathbf{vk}, c)) \rightarrow 1) \right| = \text{Adv}_{n, q, \frac{q-1}{2\gamma_2}, k, \ell, \gamma_1, S_\eta, \tau}^{\pi\text{nt-MLWR}}(\mathcal{B}).$$

Hybrid versus simulated. To conclude, we show that

$$\left| \Pr_{\mathbf{vk}, c, \mathbf{s}_1} (\mathcal{A}(\text{Hyb}(\mathbf{vk}, \mathbf{s}_1, c)) \rightarrow 1) - \Pr_{\mathbf{vk}, c, \mathbf{s}_1} (\mathcal{A}(\text{Sim}(\mathbf{vk}, c)) \rightarrow 1) \right| = 0.$$

To do so, we recall Lemma 6. The only changes in computations between **Sim** and **Hybrid** are the generation of $\tilde{\mathbf{z}}$ and the check of $\|\tilde{\mathbf{z}}\|_\infty \leq \gamma_1 - \beta$ instead of $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$. However, Lemma 6 ensures that the distributions are identical, and the checks are also identical. The output of both algorithms hence follows the same distribution. $\square$

Remark 6. The approach above can be adapted to the case of “standard” ML-DSA and a variant of $\text{MLWR}_{n, q, \frac{q-1}{2\gamma_2}, k, \ell, U(\tilde{S}_{\gamma_1})}$, where the adversary is not only given some independently sampled $(\mathbf{s}_1, \mathbf{s}_2, c) \leftarrow U(S_\eta^\ell \times S_\eta^k \times \mathcal{C})$ but also a bit b stating whether $\|\mathbf{y} + c\mathbf{s}_1\|_\infty < \gamma_1 - \beta$ and $\|\text{LowBits}_q(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta$. While this only represents one extra bit of information, reducing it to $\text{MLWR}_{n, q, \frac{q-1}{2\gamma_2}, k, \ell, U(\tilde{S}_{\gamma_1})}$ is tricky as guessing the bit would not be sufficient to give a meaningful reduction, while this works for their search variants.

Remark 7. As already mentioned in Remark 5, adversary $\mathcal{B}$ is essentially forfeiting when the sample is accepted. As such, its advantage can never be more than ρ , the rejecting probability, which corresponds to the gap between $\pi\text{nt-MLWR}$ and its variant which would be conditioned on rejected samples.

6 Performance of SUCRE on Cortex-M4, Comparison to other Countermeasures ([CGL⁺24] and [ABC⁺23])

In this section, we compare the performance of the most efficient implementations for the rejection sampling step of ML-DSA applied to a single polynomial vector of ℓ elements, either $\mathbf{z}$ or $\mathbf{r}_0$: **SecBoundCheck** from [ABC⁺23], **RejectSampling** from [CGL⁺24]⁷, and our

⁷The C code of [CGL⁺24] has been made publicly available at: https://github.com/fragerar/tches24_masked_Dilithium. It is worth noting that, for the sake of fairness and completeness in our comparison, we had to correct the **LMSwitch** and **RejectSampling** implementations to use the quasilinear **RefreshQuasiLin** gadget instead of the linear **RefreshLin** gadget, as the authors appeared to have mistakenly used the latter. This correction, however, does not significantly affect either the execution speed or the memory consumption.

new SUCRE(-no-SP2S), for a number of shares between 2 and 11 (*i.e.* between $d = 1$ and $d = 10$ respectively). For the particular case of `SecBoundCheck` from [ABC⁺23], the comparison scope is limited due to unavailable public implementation as discussed below.

The comparisons and measurements are performed on the sole rejection sampling routine (as opposed to the whole ML-DSA algorithm) since we only focus on this primitive. This has also the advantage of overcoming tight memory constraints issues on the measurement platform. Following the other benchmarks methodology regarding ML-DSA, the cycles measurement concerns a unique rejection sampling loop iteration to remain independent from the rejection probability. For fairness of the benchmark, we also avoid early aborts of the tested rejection sampling primitives by averaging the results over conforming-only values that will always process the complete ℓ components of the polynomial vector to be checked (*i.e.* values such that $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $\|\mathbf{r}_0\|_\infty < \gamma_2 - \beta$). For memory consumption, we expect l size polynomial vectors available as inputs and only account for the rejection sampling internal memory usage.

The performance measurements are performed on Cortex-M4, namely a STM32 Nucleo L4R5ZI board with a STM32L4R5ZI SoC embedding 640 KiB of SRAM and 2 MiB of flash, configured with zero-wait state clocks at 20 MHz for accurate benchmarking. The measurements capture both speed (number of cycles) as well as memory consumption⁸ used for the algorithm. Also, since masking techniques make heavy usage of randomness, the impact of RNG generation is shown by using two different sources of randomness:

- STM32L4R5ZI's hardware based TRNG (True Random Number Generator). It is a dedicated peripheral in the SoC producing one 32-bit random value every 53 CPU cycles.
- A lightweight non-cryptographic PRNG based on a very simple LFSR. The idea is to measure the randomness bottleneck with a source providing 32 bits in much fewer cycles than the STM32 TRNG (32 bits in < 10 cycles).

The compilation toolchain used for these measurements is ARM's `arm-none-eabi` in version 13.3.re11, with the `-O3` option for speed. We provide on Figure 12 (respectively on Figure 13 and Figure 14) the performance measurements for speed in kilo-cycles and memory in bytes for ML-DSA-44 (respectively for ML-DSA-65, ML-DSA-87). The detailed data for these measurements are provided in Table 6 (respectively in Table 7 and Table 8). Both SUCRE and SUCRE-no-SP2S (*i.e.* without the modulus switching, see Section 3) are benchmarked against the rejection sampling implementation from [CGL⁺24]. Specifically for ML-DSA-65, we provide the comparison with the masked rejection sampling primitive with TRNG `SecBoundCheck` from [ABC⁺23] that optimizes its Boolean gadgets with bit-slicing (and that uses a transfer from arithmetic to Boolean shares for the bound checking)⁹. The authors of [ABC⁺23] do not provide much details on how the rejection sampling is benchmarked, we only know that one loop of a complete Dilithium-3 (*i.e.* ML-DSA-65) is measured. More precisely, it is not clear if the figures provided for `SecBoundCheck` encompass the two rejections for $\mathbf{z}$ and $\mathbf{r}_0$ without early aborts. In order to keep things comparable, we additionally provide on Figure 13 the cumulative performance for SUCRE applied on $\mathbf{z}$ and SUCRE-no-SP2S applied on $\mathbf{r}_0$ (denoted SUCRE+SUCRE-no-SP2S, the total memory consumption remains the same).

The cycles performance of both variants of SUCRE are competitive as the complexity analysis in Table 4 suggests, at the expense of a steep linear increase of the memory

⁸Only static stack buffers are used with no dynamic allocation using `malloc`. This allows to easily measure memory consumption with a stack spraying technique as used in the [KPR⁺] framework.

⁹Although the figures are directly extracted from the article's Table 2 for 2 to 8 shares (no source code has been provided by the authors), we stress that the comparison is still fair as the exact same Cortex-M4 platform has been used. Only the number of clock cycles is provided for comparison as the memory usage has not been reported by the authors.

usage with the number of shares (which is expected since all the ℓ elements of the vector polynomial to be checked must be available in memory for applying the full permutation, and the original value to be checked cannot be altered in the case of $\mathbf{z}$ as it is returned in the signature). The usage of SP2Switch for SUCRE has impacts on the cycles count but not on the total memory consumption (the gadget’s local memory usage is shadowed by the other parts of the countermeasure). As a matter of fact, SUCRE is tailored for situations where speed matters while keeping the implementation secure using high-order masking, achieving four to six times speedups on average when compared to [CGL⁺24] (and an order of magnitude speedup when using SUCRE-no-SP2S). For ML-DSA-65, SUCRE speed seems on par with [ABC⁺23] when both $\mathbf{z}$ and $\mathbf{r}_0$ are checked thanks to SUCRE-no-SP2S very competitive results. This latter performance comparison is however inherently limited due to the lack of information (about memory usage, the other ML-DSA variants and large number of shares), as well as the usage of assembly optimised parts for the arithmetic according to the authors as opposed to pure C for the current SUCRE benchmarks.

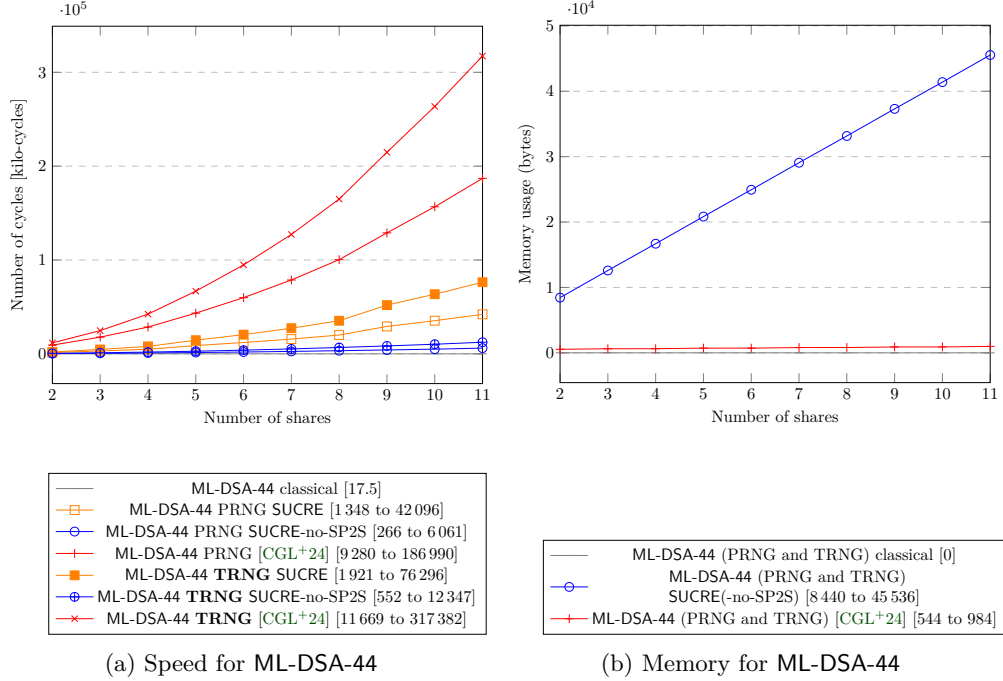


Figure 12: Classical, SUCRE, SUCRE-no-SP2S and [CGL⁺24] speed (in kilo-cycles) and memory usage (in bytes) performance comparison for ML-DSA-44, for 2 to 11 shares.

Timing and memory optimizations for SUCRE. It is worth noticing that one can push the advantage of SUCRE-no-SP2S very competitive speeds even further in ML-DSA by checking $\mathbf{r}_0$ before $\mathbf{z}$, while exploiting the relatively high rejection probability of $\mathbf{r}_0$. More optimizations are also expected for speed when replacing performance-critical parts with dedicated assembly.

The memory usage can also be drastically improved for SUCRE(no-SP2S) when applied to inputs that can be modified, *e.g.* in the case of $\mathbf{r}_0$: since it is not used after bound checking, all the operations can be performed in place and memory consumption boils down to the permutation and internal gadgets, hence saving $256 \times k \times (d+1) \times 4$ bytes for this operation. In the case where inputs must be preserved, *e.g.* the returned $\mathbf{z}$ when no rejection applies, it is also possible to save $256 \times \ell \times (d+1) \times 4$ bytes by using a PRNG with

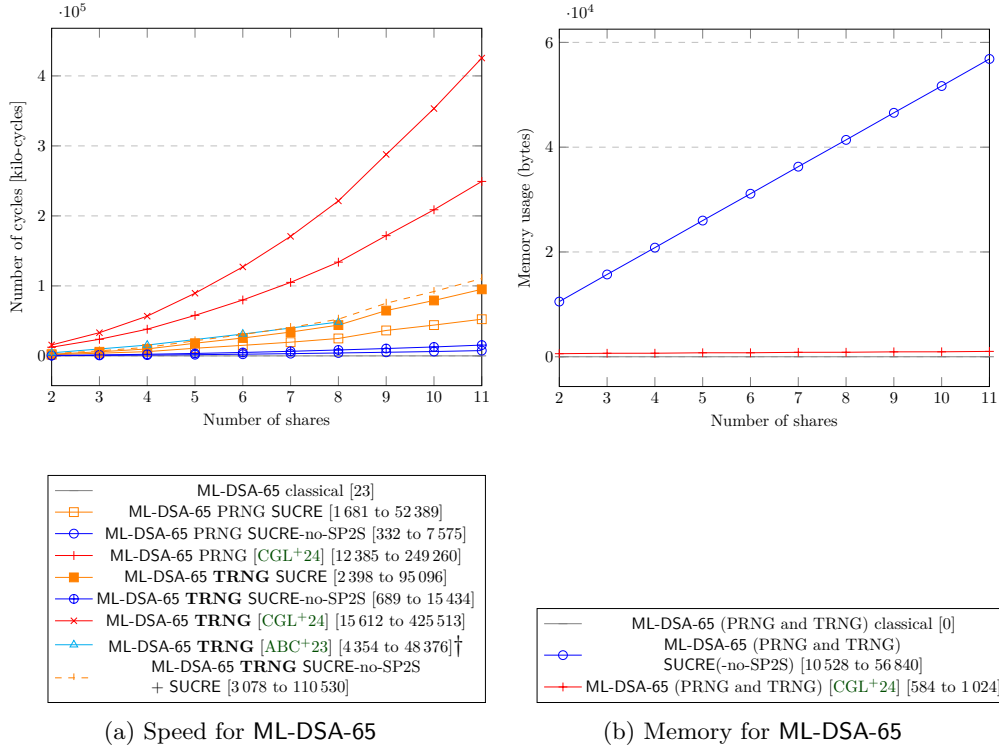


Figure 13: Classical, SUCRE, SUCRE-no-SP2S and [CGL⁺24] speed (in kilo-cycles) and memory usage (in bytes) performance comparison for ML-DSA-65, for 2 to 11 shares.
[†]Only speed performance for [ABC⁺23] with TRNG (4, 6 and 8 shares) as extracted from the article.

$d + 1$ seeds¹⁰, each one generating a permutation of **NormCheck**. In this case, it is possible to apply the permutations in-place in the memory slot of $\mathbf{z}$. The seeds are stored in memory so that the inverse permutations can be applied to the unmasked vector $\mathbf{z}$ when it is not rejected. In order to make reversing the random swaps easier, the PRNG could be used in counter mode so that random permutation indices can be easily re-generated in reverse order. The extra cost in cycles of this inversion would be contained and only applied in case of accepted $\mathbf{z}$. Moreover, the PRNG usage for the permutations in **NormCheck** could also improve the performance in hardware contexts where TRNG generation is very slow. Security-wise, using a PRNG with $d + 1$ seeds instead of a TRNG for the permutation indices also preserves the probing security. Indeed, the probing security proof remains valid even if a single probe reveals a full permutation to the adversary.

7 Conclusion and future Works

We have introduced a new lattice assumption that leads to a new comparison gadget implying a significant performance gain for masked ML-DSA. This work has opened several interesting future works.

$\pi\text{nt-MLWR}$ security. A natural question is whether the permutation information in $\pi\text{nt-MLWR}$ provides any real advantage in solving the underlying decisional MLWR problem—especially given the vast number of solutions even when candidate solutions are constrained

¹⁰This only concerns SUCRE-no-SP2S and not SUCRE because of the modulus switching.

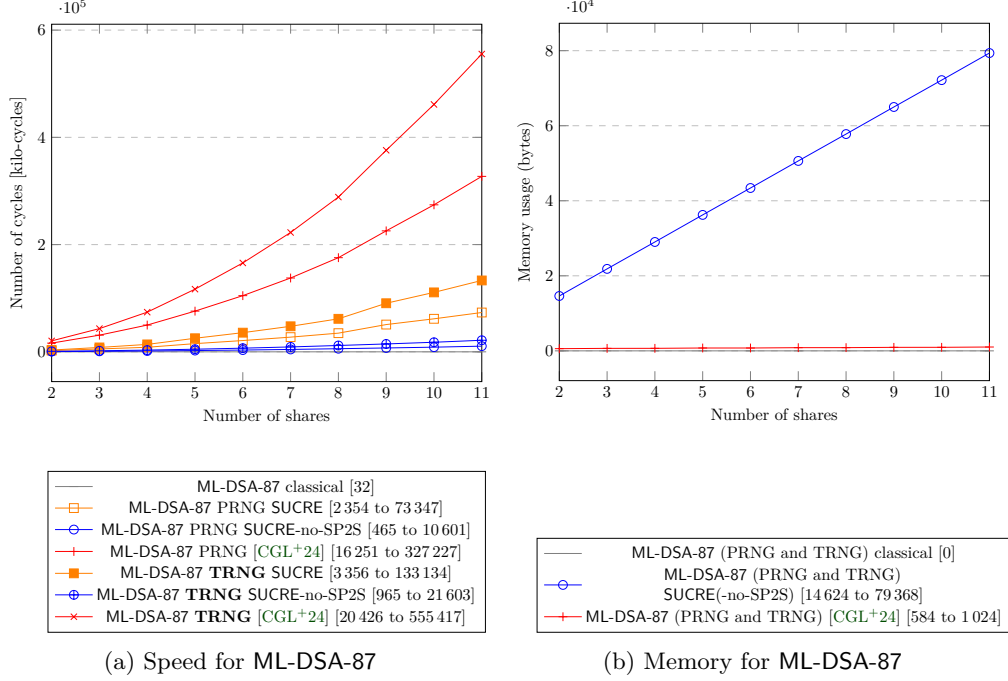


Figure 14: Classical, SUCRE, SUCRE-no-SP2S and [CGL⁺24] speed (in kilo-cycles) and memory usage (in bytes) performance comparison for ML-Dsa-87, for 2 to 11 shares.

to match the observed coordinate sums. Another promising direction is to enhance existing attacks on the Permuted Kernel Problem by exploiting the shortness of the vector $\mathbf{y}$ and the structure in both the matrix and the permutation. Exploring this may help clarify the cryptanalytic value of such partial leakage.

Leakage Model. While SUCRE is formally proven secure in the probing model—which remains the standard framework for all existing countermeasures of ML-Dsa—it is important to recognize the limitations of this abstraction. In particular, the probing model does not capture certain classes of practical attacks, such as horizontal attacks, which can exploit statistical dependencies across multiple executions and are known to be especially effective in real-world embedded systems. A more realistic framework for modeling physical leakage is the noisy leakage model, formalized by Prouff and Rivain [PR13], where adversaries observe noisy functions of intermediate values rather than idealized probes. Unfortunately, directly proving the security of cryptographic schemes in this model remains extremely challenging. First introduced as an intermediate step, the random probing model [DDF14] has emerged as an attractive alternative. It combines practical relevance with theoretical simplicity, and closely relates to the noisy leakage model through a tight reduction. That said, establishing the security of our construction in the random probing model—although a highly desirable milestone—is non-trivial. In particular, current proof techniques do not allow for tight estimation of the random probing leakage introduced by the permutation layers used in our design. Therefore, reaching this goal would probably require adapting the current design of SUCRE to better align with the constraints of existing proof techniques. In the meantime, the use of fresh refresh gadgets between each permutation step—which is a feature of our countermeasure—significantly mitigates the risks of cumulative leakage and strengthens our confidence in its practical resistance against advanced side-channel attacks.

Parallel implementations. The introduced lattice assumption could also be applied in a context where parallel instructions are possible so that the comparisons for all the coefficients could be performed at the same clock cycle and combined together. Indeed, in that case, the leakage would correspond to the leakage of all the coefficients at the same time, which comes down to removing the knowledge of the order of the coefficients. Hence, with such parallelization, assuming π nt-MLWR or variants, the countermeasure could consist in only refreshing, unmasking, and comparing. Such parallel implementations obviously include dedicated hardware circuits, but also software implementation taking advantage of bitslicing and vectorized SIMD instructions on general purpose CPUs or GPUs. An interesting future avenue would be to investigate such parallel implementations.

Memory consumption reduction. A way to significantly reduce the memory consumption of our countermeasure would be to permute the coefficients in subsets of n elements, rather than permuting the entire vector, *i.e.* applying one permutation per polynomial. Our security reduction can be easily adapted to this approach, with the caveat that the resulting variant of MLWR would provide a much stronger hint, due to this more limited form of shuffling. Investigating the memory savings of this variant, as well as the cryptanalysis of the resulting assumption, would be of great interest to understand the impact of this relaxation of SUCRE.

Acknowledgements. This work was funded by the European Union (ERC, AMaskZONE, 101077506, and Next Generation EU through the France Relance program) and by the French government through the France 2030 program (RESQUE project). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

- [ABC⁺23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting Dilithium against leakage revisited sensitivity analysis and improved implementations. *IACR TCHES*, 2023(4):58–79, 2023.
- [AOVCG25] Paco Azevedo-Oliveira, Andersson Calle Viera, Benoît Cogliati, and Louis Goubin. Finding a polytope: A practical fault attack against dilithium. Cryptology ePrint Archive, Report 2025/195, 2025.
- [ASY22] Shweta Agrawal, Damien Stehlé, and Anshu Yadav. Round-optimal lattice-based threshold signatures, revisited. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *ICALP 2022*, volume 229 of *LIPICs*, pages 8:1–8:20. Schloss Dagstuhl, July 2022.
- [BAE⁺24] Olivier Bronchain, Melissa Azouaoui, Mohamed ElGhamrawy, Joost Renes, and Tobias Schneider. Exploiting small-norm polynomial multiplication with physical attacks application to CRYSTALS-Dilithium. *IACR TCHES*, 2024(2):359–383, 2024.
- [BBD⁺16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016.

- [BBD⁺23] Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. Fixing and mechanizing the security proof of Fiat-Shamir with aborts and Dilithium. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 358–389. Springer, Cham, August 2023.
- [BBE⁺18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 354–384. Springer, Cham, April / May 2018.
- [BBE⁺24] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. *Journal of Cryptology*, 37(1):5, January 2024.
- [BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *IACR TCHES*, 2022(4):553–588, 2022.
- [BCM21] Rouzbeh Behnia, Yilei Chen, and Daniel Masny. On removing rejection conditions in practical lattice-based signatures. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*, pages 380–398. Springer, Cham, 2021.
- [BCP⁺20] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. Random probing security: Verification, composition, expansion and new constructions. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 339–368. Springer, Cham, August 2020.
- [BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 23–39. Springer, Berlin, Heidelberg, August 2016.
- [BCRT23] Sonia Belaïd, Gaëtan Cassiers, Matthieu Rivain, and Abdul Rahman Taleb. Unifying freedom and separation for tight probing-secure composition. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part III*, volume 14083 of *LNCS*, pages 440–472. Springer, Cham, August 2023.
- [BDE⁺18] Jonathan Bootle, Claire Delaplace, Thomas Espitau, Pierre-Alain Fouque, and Mehdi Tibouchi. LWE without modular reduction and improved side-channel attacks against BLISS. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 494–524. Springer, Cham, December 2018.
- [BDH11] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 117–129. Springer, Berlin, Heidelberg, November / December 2011.

- [CGL⁺24] Jean-Sébastien Coron, François Gérard, Tancrede Lepoint, Matthias Trannoy, and Rina Zeitoun. Improved high-order masked generation of masking vector and rejection sampling in dilithium. *IACR TCHES*, 2024(4):335–354, 2024.
- [CGMZ22] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion algorithms and masking lattice-based encryption. *IACR TCHES*, 2022(2):1–40, 2022.
- [CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of Dilithium. *IACR TCHES*, 2023(4):110–145, 2023.
- [CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between Boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 188–205. Springer, Berlin, Heidelberg, September 2014.
- [CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Berlin, Heidelberg, August 1999.
- [DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, Berlin, Heidelberg, May 2014.
- [DDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with side information: Attacks and concrete security estimation. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 329–358. Springer, Cham, August 2020.
- [DFPS23] Julien Devevey, Pouria Fallahpour, Alain Passelègue, and Damien Stehlé. A detailed analysis of Fiat-Shamir with aborts. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 327–357. Springer, Cham, August 2023.
- [DGHK23] Dana Dachman-Soled, Huijing Gong, Tom Hanson, and Hunter Kippen. Re-visiting security estimation for LWE with hints from a geometric perspective. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 748–781. Springer, Cham, August 2023.
- [dPKPR24] Rafaël del Pino, Shuichi Katsumata, Thomas Prest, and Mélissa Rossi. Raccoon: A masking-friendly signature proven in the probing model. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 409–444. Springer, Cham, August 2024.
- [dPN25] Rafael del Pino and Guilhem Niot. Finally! a compact lattice-based threshold signature. In *PKC*, 2025.
- [dPPRS23] Rafaël del Pino, Thomas Prest, Mélissa Rossi, and Markku-Juhani O. Saari-nen. High-order masking of lattice signatures in quasilinear time. In *2023 IEEE Symposium on Security and Privacy*, pages 1168–1185. IEEE Computer Society Press, May 2023.

- [GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 530–547. Springer, Berlin, Heidelberg, September 2012.
- [GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR TCHES*, 2022(3):141–164, 2022.
- [GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the “duplication” method). In Çetin Kaya Koç and Christof Paar, editors, *CHES’99*, volume 1717 of *LNCS*, pages 158–172. Springer, Berlin, Heidelberg, August 1999.
- [GR20] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qtesla. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications*, pages 74–91, Cham, 2020. Springer International Publishing.
- [GR24] Morgane Guerreau and Mélissa Rossi. A not so discrete sampler: Power analysis attacks on HAWK signature scheme. *IACR TCHES*, 2024(4):156–178, 2024.
- [HBD⁺22] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS⁺. Technical report, National Institute of Standards and Technology, 2022. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Berlin, Heidelberg, August 2003.
- [KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO’99*, volume 1666 of *LNCS*, pages 388–397. Springer, Berlin, Heidelberg, August 1999.
- [KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 552–586. Springer, Cham, April / May 2018.
- [KMP19] Eliane Koussa, Gilles Macario-Rat, and Jacques Patarin. On the complexity of the permuted kernel problem. Cryptology ePrint Archive, Report 2019/412, 2019.
- [KPR⁺] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. <https://github.com/mupq/pqm4>.
- [LDK⁺22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and

- Technology, 2022. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [LW20] Feng-Hao Liu and Zhedong Wang. Rounding in the rings. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 296–326. Springer, Cham, August 2020.
- [Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Berlin, Heidelberg, December 2009.
- [Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, Berlin, Heidelberg, April 2012.
- [Mar03] Jacques Martinet. *Perfect Lattices in Euclidean Spaces*. Springer, 2003.
- [MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19 International Conference on Applied Cryptography and Network Security*, volume 11464 of *LNCS*, pages 344–362. Springer, Cham, June 2019.
- [MM21] Axel Mathieu-Mahias. *Securisation of implementations of cryptographic algorithms in the context of embedded systems*. PhD thesis, Université Paris-Saclay, 2021.
- [NIS24a] NIST. Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standards Publication, NIST FIPS 204, 2024. <https://doi.org/10.6028/NIST.FIPS.204>.
- [NIS24b] NIST. Stateless Hash-Based Digital Signature Standard. Federal Information Processing Standards Publication, NIST FIPS 205, 2024. <https://doi.org/10.6028/NIST.FIPS.205>.
- [PFH⁺22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Berlin, Heidelberg, May 2013.
- [SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 534–564. Springer, Cham, April 2019.
- [UMTS24] Vincent Quentin Ulitzsch, Soundes Marzougui, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling side-channel attacks on dilithium - A small bit-fiddling leak breaks it all. In Benjamin Smith and Huapeng Wu, editors, *SAC 2022*, volume 13742 of *LNCS*, pages 3–32. Springer, Cham, August 2024.

- [ZLYW23] Shiduo Zhang, Xiuhua Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on falcon. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part IV*, volume 14007 of *LNCS*, pages 565–595. Springer, Cham, April 2023.
- [ZWSY25] Yuanyuan Zhou, Weijia Wang, Yiteng Sun, and Yu Yu. Rejected challenges pose new challenges: Key recovery of CRYSTALS-dilithium via side-channel attacks. Cryptology ePrint Archive, Report 2025/214, 2025.

A Delayed Proof of Lemma 5

Proof of Lemma 5. We start with the first identity. Let $r_0 = r \bmod \pm\alpha$.

Note that

$$\text{HighBits}_q(r, \alpha) = \begin{cases} \frac{r-r_0}{\alpha} & \text{if } r-r_0 \neq q-1 \\ 0 = \frac{r-r_0}{\alpha} \bmod +\frac{q-1}{\alpha} & \text{if } r-r_0 = q-1. \end{cases}$$

We show that $0 \leq r-r_0 \leq q-1$, which will show that the first case is already reduced $\bmod +\frac{q-1}{\alpha}$. By definition, we have both $(r-r_0) = 0 \bmod \alpha$ and $|r_0| \leq \alpha/2$. As such, $r-r_0 \in \alpha\mathbb{Z} \cap [-\alpha/2, q-1+\alpha/2] \subset [0, q-1]$, as $q-1 = 0 \bmod \alpha$. We thus have shown that

$$\text{HighBits}_q(r, \alpha) = \frac{r-r_0}{\alpha} \bmod +\frac{q-1}{\alpha}.$$

It remains to show that $(r-r_0)/\alpha = \lfloor \frac{r-1}{\alpha} \rfloor$. By definition of $r_0 \in (-\alpha/2, \alpha/2]$, we have:

$$r-r_0 - \frac{\alpha}{2} < r \leq r-r_0 + \frac{\alpha}{2}.$$

As this is an inequality between integers, we note that this is equivalent to the following statement, which reverses the strictness of the inequalities by subtracting 1 to the middle r :

$$r-r_0 - \frac{\alpha}{2} \leq r-1 < r-r_0 + \frac{\alpha}{2}.$$

Dividing by α , we find that $(r-r_0)/\alpha \in \mathbb{Z}$ satisfies the definition of the rounding of $(r-1)/\alpha$, which shows the first identity by uniqueness of the rounding.

We move on to the second identity, which we show holds over $\mathbb{Z}$. Let us assume that there exists $r \in [0, q-1]$ such that $\lfloor r \rfloor_{\frac{q-1}{\alpha}} \neq \lfloor \frac{r-1}{\alpha} \rfloor$. Note moreover that

$$\frac{r-1}{\alpha} < \frac{(q-1)r}{q\alpha} = \frac{r}{\alpha} - \frac{r}{\alpha \cdot q} \quad \text{and} \quad \left| \frac{r-1}{\alpha} - \frac{r}{\alpha} + \frac{r}{\alpha \cdot q} \right| < 1,$$

due to the fact that $r < q$. Hence we have, as they are not equal by assumption:

$$\left\lfloor \frac{r-1}{\alpha} \right\rfloor = \left\lfloor \frac{r}{\alpha} - \frac{r}{\alpha \cdot q} \right\rfloor - 1 = \lfloor r \rfloor_{\frac{q-1}{\alpha}} - 1.$$

Then there exists an integer $k \in \mathbb{Z}$ such that, by definition of the rounding function:

$$\frac{r}{\alpha} - \frac{1}{\alpha} < k + \frac{1}{2} \leq \frac{r}{\alpha} - \frac{r}{\alpha \cdot q},$$

or, equivalently,

$$0 < \alpha \left(k + \frac{1}{2} \right) - r + 1 \leq 1 - \frac{r}{q} < 1.$$

However, $\alpha(k + 1/2) - r + 1$ is an integer as $\alpha = 0 \bmod 2$, which contradicts the fact that it belongs in $(0, 1)$. This shows the equality for all $r \in [0, q-1]$.

Finally, $\text{LowBits}_q(r, \alpha) = \mathbf{e}(r)$ since $\lfloor \alpha q / (q-1) \rfloor = \alpha$. □

B Detailed Performance for SUCRE and SUCRE-no-SP2S against [CGL⁺24] and [ABC⁺23] on Cortex-M4

Table 6: Performance of the various rejection sampling primitives for ML-DSA-44. Speed performance is in kilo-cycles, memory consumption is in bytes.

Performance for ML-DSA-44											
Algorithm	# shares	2	3	4	5	6	7	8	9	10	11
		Speed	Speed	Speed	Speed	Speed	Speed	Speed	Speed	Speed	Speed
Classical (no masking)	Memory	0	0	0	0	0	0	0	0	0	0
SUCRE-no-SP2S	Speed	266	561	903	1 371	1 918	2 566	3 335	4 132	4 981	6 061
PRNG	Memory	8 440	12 592	16 688	20 832	24 936	29 072	33 160	37 304	41 392	45 536
SUCRE	Speed	1 348	3 074	4 888	8 775	12 069	15 735	20 000	29 177	35 342	42 096
PRNG	Memory	8 440	12 592	16 688	20 832	24 936	29 072	33 160	37 304	41 392	45 536
[CGL ⁺ 24]	Speed	9 280	17 779	28 527	43 376	59 868	78 674	100 339	128 909	156 681	186 990
PRNG	Memory	544	624	632	720	720	808	816	896	904	984
SUCRE-no-SP2S	Speed	552	1 133	1 856	2 800	3 918	5 234	6 764	8 419	10 219	12 347
TRNG	Memory	8 440	12 592	16 688	20 832	24 936	29 072	33 160	37 304	41 392	45 536
SUCRE	Speed	1 921	4 698	7 944	14 504	20 470	27 290	35 275	51 919	63 517	76 296
TRNG	Memory	8 440	12 592	16 688	20 832	24 936	29 072	33 160	37 304	41 392	45 536
[CGL ⁺ 24]	Speed	11 669	24 751	42 297	66 790	94 744	127 210	164 925	214 795	263 662	317 382
TRNG	Memory	544	624	632	720	720	808	816	896	904	984

Table 7: Performance of the various rejection sampling primitives for ML-DSA-65. Speed performance is in kilo-cycles, memory consumption is in bytes. For [ABC⁺23], only speed for 2, 4, 6 and 8 shares is provided as this is the only data available for the non-public implementation (see section 6).

Performance for ML-DSA-65											
Algorithm	# shares	2	3	4	5	6	7	8	9	10	11
		Speed	Speed	Speed	Speed	Speed	Speed	Speed	Speed	Speed	Speed
Classical (no masking)	Memory	0	0	0	0	0	0	0	0	0	0
SUCRE-no-SP2S	Speed	332	702	1 129	1 713	2 396	3 207	4 168	5 164	6 225	7 575
PRNG	Memory	10 528	15 704	20 824	25 992	31 112	36 280	41 392	46 560	51 672	56 840
SUCRE	Speed	1 681	3 834	6 091	10 930	15 054	19 589	24 897	36 338	44 002	52 389
PRNG	Memory	10 528	15 704	20 824	25 992	31 112	36 280	41 392	46 560	51 672	56 840
[CGL ⁺ 24]	Speed	12 385	23 719	38 115	57 849	79 885	105 037	133 949	171 765	208 785	249 260
PRNG	Memory	584	664	672	760	760	848	856	936	944	1 024
SUCRE-no-SP2S	Speed	689	1 416	2 320	3 500	4 897	6 542	8 455	10 523	12 774	15 434
TRNG	Memory	10 528	15 704	20 824	25 992	31 112	36 280	41 392	46 560	51 672	56 840
SUCRE	Speed	2 398	5 861	9 907	18 089	25 549	34 025	43 982	64 722	79 188	95 096
TRNG	Memory	10 528	15 704	20 824	25 992	31 112	36 280	41 392	46 560	51 672	56 840
[CGL ⁺ 24]	Speed	15 612	33 162	56 761	89 517	127 094	170 766	221 437	287 801	353 375	425 513
TRNG	Memory	584	664	672	760	760	848	856	936	944	1 024
[ABC ⁺ 23]	Speed	4 354	—	15 472	—	31 234	—	48 376	—	—	—
TRNG	Memory	—	—	—	—	—	—	—	—	—	—
SUCRE-no-SP2S	Speed	3 078	7 277	12 227	21 589	30 446	40 567	52 437	75 245	91 962	110 530
+SUCRE TRNG	Memory	10 528	15 704	20 824	25 992	31 112	36 280	41 392	46 560	51 672	56 840

Table 8: Performance of the various rejection sampling primitives for ML-DSA-87. Speed performance is in kilo-cycles, memory consumption is in bytes.

Performance for ML-DSA-87											
Algorithm	# shares	2	3	4	5	6	7	8	9	10	11
Classical (no masking)	Speed	32	32	32	32	32	32	32	32	32	32
	Memory	0	0	0	0	0	0	0	0	0	0
SUCRE-no-SP2S	Speed	465	982	1 580	2 398	3 354	4 490	5 835	7 229	8 711	10 601
PRNG	Memory	14 624	21 848	29 016	36 232	43 400	50 616	57 776	64 992	72 152	79 368
SUCRE	Speed	2 354	5 375	8 525	15 310	21 076	27 421	34 855	50 860	61 602	73 347
PRNG	Memory	14 624	21 848	29 016	36 232	43 400	50 616	57 776	64 992	72 152	79 368
[CGL ⁺ 24]	Speed	16 251	31 120	49 929	75 924	104 788	137 693	175 606	225 605	274 171	327 227
PRNG	Memory	584	664	672	760	760	848	856	936	944	1 024
SUCRE-no-SP2S	Speed	965	1 983	3 248	4 899	6 856	9 158	11 837	14 732	17 880	21 603
TRNG	Memory	14 624	21 848	29 016	36 232	43 400	50 616	57 776	64 992	72 152	79 368
SUCRE	Speed	3 356	8 207	13 869	25 325	35 768	47 629	61 574	90 607	110 867	133 134
TRNG	Memory	14 624	21 848	29 016	36 232	43 400	50 616	57 776	64 992	72 152	79 368
[CGL ⁺ 24]	Speed	20 426	43 323	74 003	116 875	165 794	222 614	288 611	375 882	461 390	555 417
TRNG	Memory	584	664	672	760	760	848	856	936	944	1 024