An efficient and provable masked implementation of qTESLA

François Gérard[1], Mélissa Rossi[2,3,4]

[1] Université libre de Bruxelles, Brussels, Belgium
fragerar@ulb.ac.be
[2] École normale supérieure, CNRS, PSL University, Paris, France
[3] Thales, Gennevilliers, France
[4] Inria, Paris, France
melissa.rossi@ens.fr

Abstract. Now that the NIST's post-quantum cryptography competition has entered in its second phase, the time has come to focus more closely on practical aspects of the candidates. While efficient implementations of the proposed schemes are somewhat included in the submission packages, certain issues like the threat of side-channel attacks are often lightly touched upon by the authors. Hence, the community is encouraged by the NIST to join the war effort to treat those peripheral, but nonetheless crucial, topics. In this paper, we study the lattice-based signature scheme qTESLA in the context of the masking countermeasure. Continuing a line of research opened by Barthe et al. at Eurocrypt 2018 with the masking of the GLP signature scheme, we extend and modify their work to mask qTESLA. Based on the work of Migliore et al. in ACNS 2019, we slightly modify the parameters to improve the masked performance while keeping the same security. The masking can be done at any order and specialized gadgets are used to get maximal efficiency at order 1. We implemented our countermeasure in the original code of the submission and performed tests at different orders to assess the feasibility of our technique.

Keywords: Lattice based signatures, Side-channels, Masking

1 Introduction

Following NIST's call for proposals a few years ago, the practical aspects of post-quantum cryptography have lately been studied more closely in the scientific literature. Many researchers tried to optimize parameters of cryptosystems to achieve reasonable practicality while still resisting state-of-the-art cryptanalysis. Once the design phase was over, a lot of implementations flourished on various platforms, proving that those cryptosystems can hope to achieve something useful outside of academia. Nevertheless, everyone is now well aware that having a fast and correct implementation of some functionality is seldom sufficient to get a secure system. In practice, side-channel attacks should not be overlooked and the capability of a cryptosystem to be easily protected against this kind of threats may be a strong argument to decide what will be the reigning algorithm in a post-quantum world.

In this work, we focus on applying the masking countermeasure to qTESLA [1], a Fiat-Shamir lattice-based signature derived from the original work of Lyubashevsky [22]. This signature is, with Dilithium [15], one of the most recent iteration of this line of research and a candidate for the NIST's competition. In 2018, Barthe et al. [3] described and implemented a proof of concept for a masked version of an ancestor of Dilithium/qTESLA called GLP [18]. Their goal was to prove that it is possible to mask the signature procedure at any order. This work led to a concrete masked implementation of Dilithium with experimental leakage tests [25]. In the latter, Migliore *et al.* noticed that replacing the prime modulus by a power of two allows to obtain a considerably more efficient masked scheme, by a factor of 7.3 to 9 for the most timeconsuming masking operations. Our work is in the same spirit. Similarly, we slightly modify the signature and parameters to ease the addition of the countermeasure while keeping the original security.

In addition, we provide a detailed proof of masking for the whole signature process taking public outputs into account. Indeed, similarly to the masking of GLP in [3], several elements of qTESLA may be securely unmasked, like, for example, the number of rejections. Besides, we propose an implementation for which we have focused on *performance and reusability*. Our masked signature implementation still keeps the property of being compatible with the original verifying procedure of qTESLA and has been directly implemented within the code of the submission. Even if we target high order masking, we also implemented specialized gadgets for order 1 masking to provide a lightweight version of the masking scheme with reasonable performance fitting nicely on embedded systems. We finally provide extensive performance data and show that the cost of provable masking can be reasonable at least for small orders. Our code is publicly available at https://github.com/fragerar/Masked_qTESLA

Parameter sets removal. While this paper was under peer review, the heuristic parameter sets on which our experiments are based were removed by the qTESLA team. We emphasis that the parameters we use were *not* broken but are not part of the standardization process anymore. Furthermore, our theoretical work is somewhat oblivious to the underlying parameter set used to instanciate the signature and the code can be adapted to implement the provably-secure sets as well.

2 Preliminaries

2.1 Notations

Rings. For any integers q, n and $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, we denote by $\mathcal{R}_q$ the ring $\mathbb{Z}_q[X]/(X^n + 1)$. Polynomials are written with bold lower case, e.g. $\mathbf{y} \in \mathcal{R}_q$. Let B be an integer, we write $\mathcal{R}_{q,[B]}$ to denote the subset of polynomials in $\mathcal{R}_q$ with coefficients in $[-B, B]$.

Norms. The usual norm operators are extended to polynomials by interpreting them as a vector of their coefficients. For a polynomial $\mathbf{v} = \sum_{i=0}^{n-1} v_i \cdot \mathbf{x}^i$, $||\mathbf{v}||_1 = \sum_{i=0}^{n-1} |v_i|$ and $||\mathbf{v}||_\infty = \mathsf{max}_i|v_i|$.

Representative. For a modulus q and an integer x, we write $x \bmod q$ to denote the unique integer $x_{cn} \in [0, \ldots, q-1]$ such that $x_{cn} \equiv x \pmod q$. We call this integer the *canonical representative* of x modulo q. We also write $x \bmod^{\pm} q$ to denote the unique integer $x_{ct} \in (-q/2, \ldots, q/2]$ (where the lower bound is included if q is odd) such that $x_{ct} \equiv x \pmod q$. We call this integer the *centered representative* of x modulo q.

Rounding. For integers w, d, the function $[\cdot]_L : \mathbb{Z} \to \mathbb{Z}, w \mapsto w \bmod^{\pm} 2^d$ denotes the signed extraction of the d last bits of w. We use this function to define $[\cdot]_M : \mathbb{Z} \to \mathbb{Z}, w \mapsto (w \bmod^{\pm} q - [w]_L)/2^d$. Those two functions are extended to polynomials by applying them separately on each coefficient.

2.2 Masking

Side channel attacks are a family of cryptanalytic attacks where the adversary is able access several physical parameters of the device running the algorithm. These physical attacks include, for instance, cache attacks, simple and correlation electromagnetic analysis or fault injections.

Modelling and protecting the information leaked though physical parameters has been an important research challenge since the original attack warning in [20].

The *probing model* or *ISW model* from its inventors [19] is the most studied leakage model. It has been introduced in order to theoretically define the vulnerability of implementations exposed to side-channel attacks. In a nutshell, a cryptographic implementation is N-probing secure iff any set of at most N intermediate variables is statisctically independent of the secrets. This model can be applied to practical leakages with the reduction established in [14] and tightened in [17]. The *masking* countermeasure performs computations on secret-shared data. It is the most deployed countermeasure in this landscape. Basically, each input secret x is split into $N+1$ variables $(x_i)_{0\leq i\leq N}$ referred to as shares. N of them are generated uniformly at random whereas the last one is computed such that their combination reveals the secret value x. The integer N is called *masking order* and represents the security level of an implementation with respect to side channels. Let us introduce two types of additive combination in the following definition.

Definition 1 (Arithmetic and Boolean masking). *A sensitive value x is shared with mod q arithmetic masking if it is split into $N+1$ shares $(x_i)_{0\leq i\leq N}$ such that*

$$x = x_0 + \cdots + x_N \pmod{q}. \qquad \text{(Arithmetic masking mod } q\text{)}$$

It is shared with Boolean masking if it is split into $N+1$ shares $(x_i)_{0\leq i\leq N}$ such that

$$x = x_0 \oplus \cdots \oplus x_N. \qquad \text{(Boolean masking)}$$

For lattice-based cryptography where most operations are linear for mod q addition, arithmetic masking seems the best choice. However, for certain operations like the randomness generation and comparisons, Boolean masking is better fit. Fortunatly, some conversions exist [11, 8, 3] and allow to switch from one masking to another.

Proofs by composition. While the conceptual idea behind the masking countermeasure is pretty simple, implementing it efficiently to achieve N-probing security has been shown to be a complex task. On one hand, it is straightforward on linear operations on which masking is equivalent to applying the original operation on each share of the sensitive data. On the other hand, the procedure is much more complicated on non-linear functions. In the latter, the mix of shares to compute the result makes it mandatory to introduce random variables and the bigger the program is, the more dependencies to be considered. This is why Barthe et al. formally defined in [4] two security properties, namely *non-interference* and *strong non-interference*, which (1) ease the security proofs for small gadgets (see Definition 2), and (2) allows to securely combine secure gadgets by inserting refreshing gadgets (which refresh sharings using fresh randomness) at carefully chosen locations[5].

Definition 2. *A (u, v)-gadget is a probabilistic algorithm that takes as inputs u shared values, and returns distributions over v-tuples of shared values.*

We first introduce the affine property for gadgets as introduced in [4].

Definition 3. *A gadget is affine iff it manipulates its input share by share.*

[5]Notice that non-interference was already used in practice [28, 13] to prove probing security of implementations.

In other words, one observation in an affine gadget can be simulated with only one share of its input. This property will be used for compositions. We now formally introduce the NI and SNI properties (as defined in [4]).

Definition 4. *A gadget is N-non-interfering (N-NI) iff any set of at most N observations can be perfectly simulated from at most N shares of each input.*

Definition 5. *A gadget is N-strong non-interfering (N-SNI) iff any set of at most N observations whose N_{int} observations on the internal data and N_{out} observations on the outputs can be perfectly simulated from at most N_{int} shares of each input.*

It is easy to check that N-SNI implies N-NI which implies N-probing security. The strong non-interference only appears in the proofs for subgadgets inside the signature and key generation algorithm. An additional notion was introduced in [3] to reason on the security of lattice-based schemes in which some intermediate variables may be revealed to the adversary.

Definition 6. *A gadget with public outputs X is N-non-interfering with public outputs (N-NIo) iff every set of at most N intermediate variables can be perfectly simulated with the public outputs and at most N shares of each input.*

2.3 Ring learning with errors

While not necessary to understand our work, we briefly recall, for completeness, the security assumption on which qTESLA is based: the hardness of *Ring Learning With Errors* (RLWE) [23]. The RLWE problem is believed to be hard for a quantum adversary and comes in two versions : Search-RLWE and Decisional-RLWE. Let χ be a narrow zero mean distribution over $\mathbb{Z}$.

Definition 7. *(*Search-RLWE*) for a secret $\mathbf{s} \in \mathcal{R}_q$ and a (polynomially bounded) number of samples $\mathbf{a}_i \cdot \mathbf{s} + \mathbf{e}_i \in \mathcal{R}_q$ with $\mathbf{a}_i \xleftarrow{r} \mathcal{R}_q$ and $\mathbf{e}_i \in \mathcal{R}$ with coefficients sampled from χ, find $\mathbf{s}$.*

Definition 8. *(*Decisional-RLWE*) for a secret $\mathbf{s} \in \mathcal{R}_q$ and a (polynomially bounded) number of samples $\mathbf{t}_i = \mathbf{a}_i \cdot \mathbf{s} + \mathbf{e}_i \in \mathcal{R}_q$ with $\mathbf{a}_i$ and $\mathbf{e}_i$ sampled as above, distinguish, with non-negligible probability, the distribution of the $\mathbf{t}_i$ from the uniform distribution over $\mathcal{R}_q$.*

In qTESLA, the distribution used is a centered gaussian of standard deviation σ.

2.4 The qTESLA signature

Let us now describe qTESLA [1], a (family of) lattice-based signature based on the RLWE problem and round 2 candidate for the NIST's post-quantum competition. The signature stems from several iterations of improvements over the original scheme of Lyubashevsky [22]. It is in fact

Table 1. Parameters for qTESLA-I and qTESLA-III

Parameters	qTESLA-I	qTESLA-III	Description
n	512	1024	Dimension of the ring
q	$4\,205\,569 \approx 2^{22}$	$8\,404\,993 \approx 2^{23}$	Modulus
σ	22.93	10.2	Standard deviation
h	30	48	Nonzero entries of $\mathbf{c}$
E	1586	1147	Rejection parameter
S	1586	1233	Rejection parameter
B	$2^{20} - 1$	$2^{21} - 1$	Bound for $\mathbf{y}$
d	21	22	Bits dropped in $[\cdot]_M$

a concrete instantiation of the scheme of Bai and Galbraith [2] over ideal lattices. Its direct contender in the competition is Dilithium [15] which is also based on this same idea of having a lattice variant of Schnorr signature. The security of Dilithium rely on problems over module lattices instead of ideal lattices, in the hope of increasing security by reducing algebraic structure, at the cost of a slight performance penalty.

To avoid overloading the paper, we will not describe in details all the subroutines and subtleties of `qTESLA` and sometimes simplify some aspects of the signature not required to understand our work.

Parameters

Here is a set of selected parameters that are relevant for the rest of the paper:

- n: Dimension of the ring
- q: Modulus
- σ: Standard deviation of the discrete gaussian
- h: Number of nonzero entries of the polynomial $\mathbf{c}$
- E and S: Rejection parameters
- B: Bounds for the coefficients of the hiding polynomial $\mathbf{y}$
- d: Number of bits dropped in rounding (used in the computation of $[\cdot]_M$)

For the sake of practicability, we focus on the heuristic version of `qTESLA` in this work. More specifically, we implement our countermeasure in `qTESLA-I` and `qTESLA-III` even though the techniques we used are not specific to any parameter set.

Scheme

Hereunder will be explicitly described the main algorithms, namely key generation, sign and verify. Beforehand, let us briefly recall the functionality of each of the subroutines for completeness. We redirect the interested reader to [1] or the NIST submission for a detailed description.

- PRF: Pseudorandom function, used to expand a seed into arbitrary size randomness.
- GenA: Generate a uniformly random polynomial $\mathbf{a} \in \mathcal{R}_q$.
- GaussSampler: Sample a polynomial according to a Gaussian distribution, parameters of the distribution are fixed in the sampler.
- CheckS: Verify that the secret polynomial $\mathbf{s}$ does not have too large coefficients.
- CheckE: Verify that the secret polynomial $\mathbf{e}$ does not have too large coefficients.
- ySampler: Sample a uniformly random polynomial $\mathbf{y} \in \mathcal{R}_{q,[B]}$.
- H: Collision resistant hash function.
- Enc: Encode a bitstring into a sparse polynomial $\mathbf{c} \in \mathcal{R}_{q,[1]}$ with $||\mathbf{c}||_1 = h$

Key generation (Alg 1). The key generation will output a RLWE sample together with some seeds used to generate public parameters and to add a deterministic component to the signing procedure. The algorithm starts by expanding some randomness into a collection of seeds and generates the public polynomial $\mathbf{a}$ before moving on to the two secret values $\mathbf{s}$ and $\mathbf{e}$. Those two values are sampled from a gaussian distribution and have to pass some checks to ensure that the products $\mathbf{s} \cdot \mathbf{c}$ and $\mathbf{e} \cdot \mathbf{c}$ do not have too large coefficients. After that, the main component $\mathbf{t}$ of the public key is computed as $\mathbf{t} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e}$. The output consists of the secret key $sk = (\mathbf{s}, \mathbf{e}, \text{seed}_a, \text{seed}_y)$ and the public key $pk = (seed_a, \mathbf{t})$.

Algorithm 1 qTESLA key generation

Result: $sk = (\mathbf{s}, \mathbf{e}, \text{seed}_a, \text{seed}_y)$,
$pk = (seed_a, \mathbf{t})$

counter $\leftarrow 1$
pre-seed $\xleftarrow{r} \{0,1\}^\kappa$
$\text{seed}_{s,e,a,y} \leftarrow \mathsf{PRF}(\text{pre-seed})$
$\mathbf{a} \leftarrow \mathsf{GenA}(\text{seed}_a)$
do
    $\mathbf{s} \leftarrow \mathsf{GaussSampler}(\text{seed}_s, \text{counter})$
    counter $\leftarrow$ counter $+ 1$
while $(\mathsf{CheckS}(\mathbf{s}) \neq 0)$
do
    $\mathbf{e} \leftarrow \mathsf{GaussSampler}(\text{seed}_e, \text{counter})$
    counter $\leftarrow$ counter $+ 1$
while $(\mathsf{CheckE}(\mathbf{e}) \neq 0)$
$\mathbf{t} \leftarrow \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \bmod q$
$sk \leftarrow (\mathbf{s}, \mathbf{e}, \text{seed}_a, \text{seed}_y)$
$pk \leftarrow (seed_a, \mathbf{t})$
return sk, pk

Algorithm 2 qTESLA sign

Data: $sk = (\mathbf{s}, \mathbf{e}, \text{seed}_a, \text{seed}_y)$
Result: $\Sigma = (\mathbf{z}, \mathbf{c})$

counter $\leftarrow 1$
$r \xleftarrow{r} \{0,1\}^\kappa$
rand $\leftarrow \mathsf{PRF}(\text{seed}_y, r, \mathsf{H}(m))$
$\mathbf{y} \leftarrow \mathsf{ySampler}(\text{rand, counter})$
$\mathbf{a} \leftarrow \mathsf{GenA}(\text{seed}_a)$
$\mathbf{v} \leftarrow \mathbf{a} \cdot \mathbf{y} \bmod^{\pm} q$
$\mathbf{c} \leftarrow \mathsf{Enc}(\mathsf{H}([\mathbf{v}]_M, m))$
$\mathbf{z} \leftarrow \mathbf{y} + \mathbf{s} \cdot \mathbf{c}$
if $\mathbf{z} \notin \mathcal{R}_{q,[B-S]}$ **then**
    counter $\leftarrow$ counter $+ 1$
    goto 4
end if
$\mathbf{w} \leftarrow \mathbf{v} - \mathbf{e} \cdot \mathbf{c} \bmod^{\pm} q$
if $||[\mathbf{w}]_L||_\infty \geq 2^{d-1} - E$
or $||\mathbf{w}||_\infty \geq \lfloor q/2 \rfloor - E$ **then**
    counter $\leftarrow$ counter $+ 1$
    goto 4
end if
return $(\mathbf{z}, \mathbf{c})$

Sign(Alg 2). The sign procedures takes as input a message m and the secret key sk and outputs a signature $\Sigma = (\mathbf{z}, \mathbf{c})$. First, in order to generate the randomness needed in the algorithm, a seed is derived from a fresh random value r, seed_y and m. Next, a polynomial $\mathbf{y} \in \mathcal{R}_{q,[B]}$ is sampled to compute the value $\mathbf{v} = \mathbf{a} \cdot \mathbf{y} \bmod^{\pm} q$. The algorithm will now hash the rounded version of $\mathbf{v}$ together with the message and encode the result in a sparse polynomial $\mathbf{c}$ with only h entries in $\{-1, 1\}$. The candidate signature is computed as $\mathbf{z} = \mathbf{y} + \mathbf{s} \cdot \mathbf{c}$. Before outputting the result, two additional checks must be performed: we must ensure that $\mathbf{z}$ is in $\mathcal{R}_{q,[B-S]}$ and that $\mathbf{w} = \mathbf{v} - \mathbf{e} \cdot \mathbf{c} \bmod^{\pm} q$ is well rounded, meaning that $||[\mathbf{w}]_L||_\infty < 2^{d-1} - E$ and $||\mathbf{w}||_\infty < \lfloor q/2 \rfloor - E$ should hold. When one of the check fails, the signing procedure is restarted by sampling a new $\mathbf{y}$. When eventually both checks pass, the signature $\Sigma = (\mathbf{z}, \mathbf{c})$ is output.

Verify. (Alg 3) Signature verification is pretty lightweight and straightforward for this type of signature. Taking as input the message m, signature $\Sigma = (\mathbf{z}, \mathbf{c})$ and public key $pk = (seed_a, \mathbf{t})$, it works as follow: First, it generates the public parameter $\mathbf{a}$, then computes $\mathbf{w} = \mathbf{a} \cdot \mathbf{z} - \mathbf{t} \cdot \mathbf{c}$ and accepts the signature if $\mathbf{z} \in \mathcal{R}_{q,[B-S]}$ and $\mathbf{c} \neq \mathsf{Enc}(\mathsf{H}([\mathbf{w}]_M, m))$

Algorithm 3 qTESLA verify

Data: message m, signature $\Sigma = (\mathbf{z}, \mathbf{c})$ and public key $pk = (seed_a, \mathbf{t})$
Result: 0 if the signature is accepted else -1

$\mathbf{a} \leftarrow \mathsf{GenA}(\text{seed}_a)$
$\mathbf{w} \leftarrow \mathbf{a} \cdot \mathbf{z} - \mathbf{t} \cdot \mathbf{c} \bmod^{\pm} q$
if $\mathbf{z} \notin \mathcal{R}_{q,[B-S]}$ **or** $\mathbf{c} \neq \mathsf{Enc}(\mathsf{H}([\mathbf{w}]_M, m))$ **then**
    return -1
end if
return 0

3 Masked qTESLA

3.1 Masking-friendly design

In the process of masking qTESLA, we decided to make slight modifications in the signing procedure in order to facilitate masking. The idea is that some design elements providing small efficiency gains may be really hard to carry on to the masked version and actually do even more harm than good. Our two main modifications are the modulus which is chosen as the closest power of two of the original parameter set and the removal of the PRF to generate the polynomial $\mathbf{y}$.

Power of two modulus. Modular arithmetic is one of the core component of plenty of cryptographic schemes. While, in general, it is reasonably fast for any modulus (but not necessarily straightforward to do in constant time), modular arithmetic in masked form is very inefficient and it is often one of the bottlenecks in terms of running time. In [3], a gadget SecAddModp is defined to add two integers in boolean masked form modulo p. The idea is to naively perform the addition over the integers and to subtract p if the value is larger than p. While this works completely fine, the computational overhead is large in practice and avoiding those reductions would drastically enhance execution time. The ideal case is to work over $\mathbb{Z}_{2^n}$. In this case, almost no reductions are needed throughout the execution of the algorithm and, when needed, can be simply performed by applying a mask on boolean shares. The reason why working with a power of two modulus is not the standard way to instanciate lattice-based cryptography is that it removes the possibility to use the number theoretic transform (NTT) to perform efficient polynomial multiplication in $\mathcal{O}(n \log n)$. Instead, multiplication of polynomial has to be computed using the Karatsuba/Toom-Cook algorithm which is slower for parameters used in state-of-the-art algorithms. Nevertheless, in our case, not having to use the heavy SecAddModp gadget largely overshadows the penalty of switching from NTT to Karatsuba. Since modulus for both parameter sets were already close to a power of two, we rounded to the closest one, i.e. 2^{22} for qTESLA-I and 2^{23} for qTESLA-III. This modification does not change the security of the scheme. Indeed, security-wise, for the heuristic version of the scheme that we study, we need a q such that $q > 4B$ [6] and the corresponding decisional LWE instance is still hard. Yet, the form of q does not impact the hardness of the problem as shown in [21] and, since q was already extremely close to a power of two for both parameters sets, the practical bit hardness of the corresponding instance is not sensibly changed.

Removal of the PRF. It is well known that in Schnorr-like signatures, a devastating attack is possible if the adversary gets two different signatures using the same $\mathbf{y}$. Indeed, they can simply compute the secret $\mathbf{s} = \frac{\mathbf{z}-\mathbf{z}'}{\mathbf{c}-\mathbf{c}'}$. While such a situation is very unlikely due to the large size of $\mathbf{y}$, a technique to create a deterministic version of the signature was introduced in [26]. The idea is to compute $\mathbf{y}$ as $\mathsf{PRF}(\mathit{secret\_seed}, m)$ such that each message will have a different value for $\mathbf{y}$ unless a collision is found in PRF. This modification acts as a protection against very weak entropy sources but is not necessary to the security of the signature and was not present in ancestors of qTESLA. Unfortunately, adding this determinism also enabled some side-channel attacks [27, 7]. Hence, the authors of qTESLA decided to take the middle ground by keeping the deterministic design but also seeding the oracle with a fresh random value r[7].
While those small safety measures certainly make sense if they do not incur a significant performance penalty, we decided to drop it and simply sample $\mathbf{y}$ at random at the beginning of the

[6] The other condition on q in the parameters table of the submission is to enable the NTT
[7] Note that the fault attacks is still possible in case of failure of the RNG picking r

signing procedure. The reason is twofold. First, keeping deterministic generation of $\mathbf{y}$ implied masking the hash function evaluation itself which is really inefficient if not needed and would unnecessarily complicate the masking scheme. Second, implementing a masking countermeasure is, in general, making the hypothesis that a reasonable source of randomness (or at least not weak to the point of having a nonce reuse on something as large as $\mathbf{y}$) is available to generate shares and thus can be also used for the signature itself.

3.2 Existing gadgets

First, let us describe gadgets already existing in the literature. Since they are not part of our contribution, we decided to only recall their functionalities without formally describing them.

- SecAnd: Computes the logical and between two values given in boolean masked form, output also in boolean masked form. Order 1 algorithm: [12]. Order n algorithm [3].
- SecAdd: Computes the arithmetic add between two values given in boolean masked form, output also in boolean masked form. Order 1 algorithm: [12]. Order n algorithm [3].
- SecArithBoolModq: Converts a value in arithmetic masked form to a value in boolean masked form. Order 1 algorithm: [16]. Order n: [11]. We slightly modify it to an algorithm denoted GenSecArithBoolModq taking into account non power of two number of shares. It can be found in Algorithm 4. When a masked value composed of an odd number of shares t is presented to the algorithm, it first splits them in two uneven parts of size $\lfloor t/2 \rfloor + 1$ and $\lfloor t/2 \rfloor$ before proceeding to the recursive call. The subroutine Expand takes as input an arbitrary number of shares t' and expand them in $2t'$ shares. Applying Expand to both parts, we end up with a part p_1 of size $t+1$ and a part p_2 of size $t-1$. We merge the two last shares of p_1 and append a zero to p_2 to get two size t masking that are finally added together to yield the final boolean masking. Note that in practice, the top level call is done from another (non recursive) function that reduces the result in order to have a conversion modulo q. We recall that thanks to our power of two modulus, this can be done by simply keeping $\log_2 q$ bits of each shares.
- SecBoolArith: Converts a value in boolean masked form to a value in arithmetic masked form. Order 1 algorithm: [16]. Order n algorithm: [8]. This gadget does not explicitly appear in the following but is used inside DataGen.
- DataGen: Takes as input an integer B and outputs a polynomial $\mathbf{y} \in \mathcal{R}_{q,[B]}$ in arithmetic masked form. Uses the boolean to arithmetic conversion.
- FullXor: Merges shares of a value in boolean masked form and output the unmasked value.
- FullAdd: Merges shares of a value in arithmetic masked form and output the unmasked value.
- Refresh: Refreshes a boolean sharing using fresh randomness [19]. We use its N-SNI version, sometimes denoted FullRefresh ([10] Algorithm 4), which is made of a succession of $N+1$ linear refresh operations.

3.3 New gadgets

To comply with the specifications of qTESLA, our signature scheme includes new components to be masked that were not covered or different than in [3, 25]. In all the following, RADIX refers to the size of the integer datatype used to store the shares.

Absolute value (Alg. 5): The three checks during the signing procedure are : $\mathbf{z} \notin \mathcal{R}_{q,[B-S]}$, $||[\mathbf{w}]_L||_\infty \geq 2^{d-1} - E$ and $||\mathbf{w}||_\infty \geq \lfloor q/2 \rfloor - E$. They all involve going through individual coefficients (or their low bits) of a polynomial and checking a bound on their absolute value. In the first version of our work, we were actually making two comparisons on each signed coefficients

Algorithm 4 GenSecArithBoolModq

Data: An arithmetic masking $(a_i)_{0\leq i\leq N}$ of some integer x
Result: A boolean masking $(b_i)_{0\leq i\leq N}$ of the same integer x

if $N = 0$ **then**
  $b_0 \leftarrow a_0$
  return $(b_i)_{0\leq i\leq N}$
end if
$\texttt{HALF} \leftarrow \lfloor N/2 \rfloor$
$(x_i)_{0\leq i\leq \texttt{HALF}} \leftarrow \mathsf{GenSecArithBoolModq}((a_i)_{0\leq i\leq \texttt{HALF}})$
$(x'_i)_{0\leq i\leq 2*\texttt{HALF}} \leftarrow \mathsf{Expand}((x_i)_{0\leq i\leq \texttt{HALF}})$
$(y_i)_{0\leq i\leq \lfloor (N-1)/2 \rfloor} \leftarrow \mathsf{GenSecArithBoolModq}((a_i)_{\texttt{HALF}+1\leq i\leq N})$
$(y'_i)_{0\leq i\leq 2*\lfloor (N-1)/2 \rfloor} \leftarrow \mathsf{Expand}((y_i)_{0\leq i\leq \lfloor (N+1)/2 \rfloor})$
if N is even **then**
  $y'_{2*\lfloor (N-1)/2 \rfloor} \leftarrow 0$
  $x'_{2*\texttt{HALF}-1} \leftarrow x'_{2*\texttt{HALF}-1} \oplus x'_{2*\texttt{HALF}}$
end if
$(b_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((x'_i)_{0\leq i\leq N}, (y'_i)_{0\leq i\leq N})$

Algorithm 5 Absolute Value - AbsVal

Data: A boolean masking $(x_i)_{0\leq i\leq N}$ of some integer x and an integer k
Result: A boolean masking $(|x|_i)_{0\leq i\leq N}$ corresponding to the absolute value of $x \bmod^{\pm} 2^k$

1: $(mask_i)_{0\leq i\leq N} \leftarrow ((x_i)_{0\leq i\leq N} << (\text{RADIX} - k)) >> (\text{RADIX} - 1))$
2: $(x'_i)_{0\leq i\leq N} \leftarrow \mathsf{Refresh}((x_i)_{0\leq i\leq N})$
3: $(x_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((x'_i)_{0\leq i\leq N}, (mask_i)_{0\leq i\leq N}))$
4: $(|x|_i)_{0\leq i\leq N} \leftarrow ((x_i)_{0\leq i\leq N} \oplus (mask_i)_{0\leq i\leq N}) \wedge (2^k - 1)$

before realizing that it was actually less intensive to explicitly compute the absolute value and do only one comparison. The gadget takes as input any integer x masked in boolean form and outputs $|x \bmod^{\pm} 2^k|$. Since computers are performing two's complement arithmetic, the absolute value of x can be computed as follows:

1. $m \leftarrow x >> RADIX - 1$
2. $|x| \leftarrow (x + m) \oplus m$

As we work on signed integers, one can note that the $>>$ in the first step is an arithmetic shift and actually writes the sign bit in the whole register. If x is negative then $m = -1$ (all ones in the register) and if x is positive then $m = 0$. The gadget AbsVal is using the same technique to compute $|x \bmod^{\pm} 2^k|$. The small difference is that the sign bit is in position k instead of position RADIX. This is why line 1 is moving the sign bit (modulo 2^k) in first position before extending it to the whole register to compute the mask.

Masked rounding (Alg. 6): In [2], a compression technique was introduced to reduce the size of the signature. It implies rounding coefficients of a polynomial. Revealing the polynomial before rounding would allow an adversary to get extra information on secret values and thus, this operation has to be done on the masked polynomial. Recall that the operation to compute is $[v]_M = (v \bmod^{\pm} q - [v]_L)/2^d$.
The first step is to compute the centered representative of v, i.e. subtract q from v if $v > q/2$. Taking advantage of our power of two modulus, this operation would be really easy to do if the centered representative was defined as the integer congruent to v in the range $[-q/2, q/2)$ since it would be equivalent to copying the q^{th} bit of v in the most significant part, which can be performed with simple shift operations on shares. Unfortunately, the rounding function of `qTESLA` works with representatives in $(-q/2, q/2]$. As we wanted compatibility with the original scheme, we decided to stick with their design. Nevertheless, we were still able to exploit our power of

Algorithm 6 Masked rounding - MaskedRound

Data: An arithmetic masking $(a_i)_{0\leq i\leq N}$ of some integer a
Result: An integer r corresponding to the modular rounding of a

1: $(\text{MINUS\_Q\_HALF}_i)_{0\leq i\leq N} \leftarrow (-q/2-1, 0, ..., 0)$
2: $(\text{CONST}_i)_{0\leq i\leq N} \leftarrow (2^{d-1}-1, 0, ..., 0)$
3: $(a'_i)_{0\leq i\leq N} \leftarrow \mathsf{GenSecArithBoolModq}(a_i)_{0\leq i\leq N}$
4: $(b_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((a'_i)_{0\leq i\leq N}, (\text{MINUS\_Q\_HALF}_i)_{0\leq i\leq N})$
5: $b_0 = \neg b_0$
6: $(b_i)_{0\leq i\leq N} \leftarrow ((b_i)_{0\leq i\leq N} >> \text{RADIX}-1) << \log_2 q$
7: $(a'_i)_{0\leq i\leq N} \leftarrow (a'_i)_{0\leq i\leq N} \oplus (b_i)_{0\leq i\leq N}$
8: $(a'_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((a'_i)_{0\leq i\leq N}, (\text{CONST}_i)_{0\leq i\leq N})$
9: $(a'_i)_{0\leq i\leq N} \leftarrow (a'_i)_{0\leq i\leq N} >> d$
10: **return** $t \coloneqq \mathsf{FullXor}((a'_i)_{0\leq i\leq N})$

Algorithm 7 Masked well-rounded - MaskedWR

Data: Integer $a \in \mathbb{Z}_q$ in arithmetic masked form $(a_i)_{0\leq i\leq N}$
Result: A boolean masking r of $(\|a\| \leq q/2-E) \wedge (\|[a]_L\| \leq 2^{d-1}-E)$

1: $(\text{SUP\_Q}_i)_{0\leq i\leq N} \leftarrow (-q/2+E, 0, ..., 0)$
2: $(\text{SUP\_D}_i)_{0\leq i\leq N} \leftarrow (-2^{d-1}+E, 0, ..., 0)$
3: $(a'_i)_{0\leq i\leq N} \leftarrow \mathsf{GenSecArithBoolModq}(a_i)_{0\leq i\leq N}$
4: $(x_i)_{0\leq i\leq N} \leftarrow \mathsf{AbsVal}((a'_i)_{0\leq i\leq N}, \log_2 q)$
5: $(x_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((x_i)_{0\leq i\leq N}, (\text{SUP\_Q}_i)_{0\leq i\leq N}))$
6: $(b_i)_{0\leq i\leq N} \leftarrow (x_i)_{0\leq i\leq N} >> (\text{RADIX}-1)$
7: $(a'_i)_{0\leq i\leq N} \leftarrow \mathsf{Refresh}((a'_i)_{0\leq i\leq N})$
8: $(a'_i)_{0\leq i\leq N} \leftarrow (a'_i)_{0\leq i\leq N} \wedge 2^d-1$
9: $(y_i)_{0\leq i\leq N} \leftarrow \mathsf{AbsVal}((a'_i)_{0\leq i\leq N}, d)$
10: $(y_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((y_i)_{0\leq i\leq N}, (\text{SUP\_D}_i)_{0\leq i\leq N}))$
11: $(b'_i)_{0\leq i\leq N} \leftarrow (y_i)_{0\leq i\leq N} >> (\text{RADIX}-1)$
12: $(b_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAnd}((b_i)_{0\leq i\leq N}, (b'_i)_{0\leq i\leq N})$
13: **return** $r \coloneqq \mathsf{FullXor}((b_i)_{0\leq i\leq N})$

two modulus. Indeed, in this context, switching from positive to negative representative modulo q is merely setting all the high bits to one. Hence, we subtract $q/2+1$ from v, extract the sign bit b and copy $\neg b$ to all the high bits of v.
The second step is the computation of $(v-[v]_L)/2^d$. We used a small trick here. Subtracting the centered representative modulo 2^d is actually equivalent to the application of a rounding to the closest multiple of 2^d with ties rounded down. Hence we first computed $v+2^{d-1}-1$ and dropped the d least significant bits. This is analogous to computing $\lfloor x \rceil = \lfloor x+0.499\ldots \rfloor$ to find the closest integer to a real value.

Masked well-rounded (Alg. 7): Unlike GLP, the signature scheme can fail to verify and may have to be restarted even if the rejection sampling test has been successful. This results from the fact that the signature acts fas a proof of knowledge only on the $\mathbf{s}$ part of the secret key and not on the error $\mathbf{e}$. Nonetheless, thanks to rounding, the verifier will be able to feed correct input to the hash function if the commitment is so called 'well-rounded'. Since not well-rounded signatures would leak information on the secret key, this verification has to be performed in masked form.
The MaskedWR gadget has to perform the two checks $||[\mathbf{w}]_L||_\infty < 2^{d-1}-E$ and $||\mathbf{w}||_\infty < \lfloor q/2 \rfloor - E$. While the cost of this rather simple operation is negligible compared to polynomial multiplication in the unprotected signature, this test is fairly expensive in masked form. Indeed, it requires four comparisons in addition to the extraction of the low bits of $\mathbf{w}$.
After trying the four comparisons method, we realized that the best strategy was actually to compute both absolute values with the AbsVal gadget. While comparisons only require one

Algorithm 8 Rejection Sampling - MaskedRS

Data: A value a to check, in arithmetic masked form $(a_i)_{0\leq i\leq N}$
Result: 1 if $|a| \leq B - S$ else 0

1: $(\text{SUP}_i)_{0\leq i\leq N} \leftarrow (-B + S - 1, 0, ..., 0)$
2: $(a'_i)_{0\leq i\leq N} \leftarrow \mathsf{GenSecArithBoolModq}((a_i)_{0\leq i\leq N})$
3: $(x_i)_{0\leq i\leq N} \leftarrow \mathsf{AbsVal}((a'_i)_{0\leq i\leq N}, \log_2 q)$
4: $(x_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((x_i)_{0\leq i\leq N}, (\text{SUP}_i)_{0\leq i\leq N})$
5: $(b_i)_{0\leq i\leq N} \leftarrow ((x_i)_{0\leq i\leq N} >> \text{RADIX} - 1)$
6: **return** $rs \coloneqq \mathsf{FullXor}((b_i)_{0\leq i\leq N})$

SecAdd and one shift, which is less than AbsVal, the cost of all SecAnd operations between the results of those comparisons makes our approach of computing the absolute value slightly better.

Rejection sampling (Alg. 8) The rejection sampling procedure consists in ensuring that the absolute value of all coefficients of a polynomial $\mathbf{z}$ are smaller than a bound B. In [3], a gadget verifying that the centered representative of a masked integer is greater than $-B$ was applied to both $\mathbf{z}$ and $-\mathbf{z}$. In [25], a less computationally intensive approach was taken: their rejection sampling gadget takes as input an arithmetic masking of a coefficient $a \in \mathbb{Z}_q$ identified by its canonical representative and check directly that either $a - B$ is negative or $a - q + B$ is positive. This can be easily done using precomputed constants $(-B - 1, 0, ..., 0)$ and $(-q + B, 0, ..., 0)$. Our approach is similar but we use instead the same technique as in the MaskedWR algorithm, that is to first compute the absolute value of a and perform the masked test $||a|| \leq B$. This saves the need for a masked operation to aggregate both tests.

Gaussian Generation (Alg. 9): This gadget is needed for the key generation. Following the round 2 specifications of `qTESLA`, we mask the technique of cumulative distribution table (CDT) used in the key generation. It consists in precomputing a table of the cumulative distribution function of a half Gaussian of standard deviation σ with a certain precision θ. This table contains say T elements p_j for $j \in [0, T]$ such that

$$p_j = 2 \cdot \sum_{i \leq j} \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-i^2}{2\sigma^2}} \text{ with } \theta \text{ bits of precision.}$$

The factor 2 is set to consider the half Gaussian. To produce a sample, we generate a random value in $(0, 1]$ with the same precision, and return the index of the last entry in the table that is smaller than that value. We present the masked version of table look up in Algorithm 9. The parameters T and θ are respectively the number of elements of the table and the bit precision of its entries. Note that this algorithm samples only half of the Gaussian, this means that to get a centered distribution, one should sample random signs for each coefficient afterward. Since, for correctness purpose, the key generation algorithm of `qTESLA` is checking the sum of the absolute value of the h largest coefficients of each polynomial after Gaussian sampling, we delay the sign choice to later in the key generation.

For the parameters, we take the exact same table as in the reference implementation. This table is kept unmasked because it contains public information. For `qTESLA`-I, $(T, \theta) = (208, 64)$ and for `qTESLA`-III, $(T, \theta) = (134, 128)$.

Due to the large size of the table, this process is quite heavy and impacts the efficiency of the key generation. To optimize the size of the table, a Renyi divergence technique is often used. It consists in using an upper bound on the relative error between the CDT distribution and an ideal

Algorithm 9 Gaussian Generation - GaussGen

Data: A table of T probability values p_j with θ bits of precision.
Result: An arithmetic masking of an element following a Gaussian of standard deviation σ

initialize $(r_i)_{0\leq i\leq N}$ as a θ-bit Boolean masking of a uniform random value $r \in (0, 1]$
$(x_i)_{0\leq i\leq N} \leftarrow (0, \ldots, 0)$
for $0 \leq j \leq T$ **do**
    initialize $(k_i)_{0\leq i\leq N}$ as a θ-bit Boolean masking of $-p_j$
    $(\delta_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}\big((r_i)_{0\leq i\leq N}, (k_i)_{0\leq i\leq N}\big)$
    $(b_i)_{0\leq i\leq N} \leftarrow (\delta_i)_{0\leq i\leq N} \gg (\theta - 1)$ ▷ 1 when $r < p_j$ or 0 otherwise
    $(b'_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAnd}\,((b_i)_{0\leq i\leq N}, (x_i)_{0\leq i\leq N})$
    initialize $(J_i)_{0\leq i\leq N}$ as a θ-bit Boolean masking of the index j
    $(b_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAnd}\,(\neg(b_i)_{0\leq i\leq N}, (J_i)_{0\leq i\leq N})$ ▷ j when $r \geq p_j$ or 0 otherwise
    $(x_i)_{0\leq i\leq N} \leftarrow (b'_i)_{0\leq i\leq N} \oplus (b_i)_{0\leq i\leq N}$
    $(x_i)_{0\leq i\leq N} \leftarrow \mathsf{Refresh}((x_i)_{0\leq i\leq N})$
end for
return $\mathsf{SecBoolArith}((x_i)_{0\leq i\leq N})$

Gaussian. This upper bound decreases with the maximum number of queries to the algorithm and it is often lower than a statistical distance estimation. Thus, with Renyi divergence techniques, smaller tables allow the same bit security. Interestingly, we could not manage to adapt this method because GaussGen is part of the key generation. First, the maximum amount of queries to the key generation is not clearly bounded in the specifications. Secondly, the security of the key generation is based on a decisional problem (Decisional-RLWE) and it is currently an open problem to apply the Renyi divergence techniques to decisional problems. Another possible optimization to reduce the size of the table would be to use the recursivity provided in [24] (proved with the statistical distance). However, the size of the table will not decrease enough to get an efficient and practical Gaussian sampling.

Masked Check (Alg. 10): This gadget is needed for the key generation. Its purpose is to check that the sum of the largest coefficients of the secret key is not too large. To recover the largest coefficients, this algorithm uses a straightforwardly masked bubble sort by doing h passes on the list of coefficients. The bubble sort uses a masked exchange subroutine where the bit 0 or 1, representing the need for an exchange or not, is also masked. It finishes with a masked comparison with a precomputed bound.

Algorithm 10 Masked Check - MaskedCheck

Data: An arithmetic masking of a polynomial $(\mathbf{s}_i)_{0\leq i\leq N}$, a bound S
Result: $ms \coloneqq 1$ if the sum of its h largest coefficients is larger than the bound S and 0 otherwise

$(\mathrm{BOUND}_i)_{0\leq i\leq N} = (-S, 0, \ldots, 0)$
Find the h largest coefficients $((c_i^{(0)})_{0\leq i\leq N}, \ldots, (c_i^{(h-1)})_{0\leq i\leq N})$ of $(\mathbf{s}_i)_{0\leq i\leq N}$ with a masked bubble sort.
$(sum_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((c_i^{(0)})_{0\leq i\leq N}, \ldots, (c_i^{(h-1)})_{0\leq i\leq N})$
$(\delta_i)_{0\leq i\leq N} \leftarrow \mathsf{SecAdd}((sum_i)_{0\leq i\leq N}, (\mathrm{BOUND}_i)_{0\leq i\leq N})$
$(\delta_i)_{0\leq i\leq N} \leftarrow ((\delta_i)_{0\leq i\leq N} >> RADIX - 1)$
return $ms \coloneqq \mathsf{FullXor}(\delta)$

3.4 Masked scheme

In all signature schemes, two algorithms can leak the secret key through side channels: the key generation algorithm and the signing algorithm.

Algorithm 11 Masked signature

Data: message m, secret key $sk = ((\mathbf{s}_i)_{0\leq i\leq N}, (\mathbf{e}_i)_{0\leq i\leq N})$, seed sd
Result: Signature $(\mathbf{z}_{unmasked}, \mathbf{c})$

$\mathbf{a} \leftarrow \mathsf{GenA}(sd)$
$(\mathbf{y}_i)_{0\leq i\leq N} \leftarrow \mathsf{DataGen}(B)$
for $i = 0, \ldots, N$ **do**
    $\mathbf{v}_i \leftarrow \mathbf{a} \cdot \mathbf{y}_i$
end for
$c \leftarrow \mathsf{MaskedHash}((\mathbf{v}_i)_{0\leq i\leq N}, m)$
$\mathbf{c} \leftarrow \mathsf{Encode}(c)$
for $i = 0, \ldots, N$ **do**
    $\mathbf{z}_i \leftarrow \mathbf{y}_i + \mathbf{s}_i \cdot \mathbf{c}$
end for
if $rs \coloneqq \mathsf{FullRS}((\mathbf{z}_i)_{0\leq i\leq N}) = 0$ **then**
    goto 2
end if
for $i = 0, \ldots, N$ **do**
    $\mathbf{w}_i \leftarrow \mathbf{v}_i - \mathbf{e}_i \cdot \mathbf{c}$
end for
if $r \coloneqq \mathsf{FullWR}((\mathbf{w}_i)_{0\leq i\leq N}) = 0$ **then**
    goto 2
end if
$\mathbf{z}_{unmasked} \leftarrow \mathsf{FullAdd}((\mathbf{z}_i)_{0\leq i\leq N})$
return $(\mathbf{z}_{unmasked}, \mathbf{c})$

Masked sign: The masked signature can be found in Algorithm 11. It uses the gadgets described in Section 3.3: the gadgets FullRS, FullWR and FullRound denote the extension of MaskedRS, MaskedWR and MaskedRound to all coefficients $j \in [0, n-1]$ of their input polynomial. Beside the removal of the PRF for $\mathbf{y}$, its structure follows closely the unmasked version of the signature. After generating the public parameter $\mathbf{a}$ with the original GenA procedure, the gadget DataGen is used to get polynomials $\mathbf{y}_i$ such that $\mathbf{y} = \sum_{i=0}^{N} \mathbf{y}_i$ belongs to $\mathcal{R}_{q,[B]}$. Then, thanks to the distributive property of the multiplication of ring elements, we can compute $\mathbf{v} = \mathbf{a} \cdot \mathbf{y} = \sum_{i=0}^{N} \mathbf{a} \cdot \mathbf{y}_i$ using regular polynomial multiplication, without relying on any complex gadget. The polynomial $\mathbf{c}$ is computed using the subroutine MaskedHash which is using the MaskedRounding gadget to compute qTESLA's rounding and hashing on a masked polynomial. In the sequel (see Section 4.2), we explain that the computation of the hash function does not have to be performed in masked form since the knowledge of its inputs does not impact the security. Once $\mathbf{c}$ has been computed, the candidate signature can be computed directly on shares with the masked secret key as $\mathbf{z} = \mathbf{y} + \mathbf{s} \cdot \mathbf{c} = \sum_{i=0}^{N} \mathbf{y}_i + \mathbf{s}_i \cdot \mathbf{c}$. Writing FullRS and FullWR to denote the extension of the MaskedRS and MaskedWR gadgets to all the coefficients of a polynomial, the security and correctness parts of the signature follow trivially. Once all checks have been passed, the signature can be safely unmasked using FullAdd and the signature output.

Algorithm 12 MaskedHash

Data: The n coefficients $a^{(j)}$ to hash, in arithmetic masked form $(a_i^{(j)})_{0\leq i\leq N}$ and the message to sign m
Result: Hash of the polynomial c

Let u be a byte array of size n
for $j = 1$ to n **do**
    $u_j \leftarrow \mathsf{MaskedRound}((a_i^{(j)})_{0\leq i\leq N})$
end for
$c \leftarrow \mathsf{H}(u, m)$
return

Masked key generation: As the number of signature queries per private key can be high (up to 2^{64} as required by the NIST competition), whereas the key generation algorithm is typically only executed once per private key, the vulnerability of the key generation to side channel attacks is therefore less critical. We nevertheless present an (inefficient) masked version of the key generation algorithm that can be found in Algorithm 13. One can remark that the bottlenck gadget, GaussGen, needs to make T comparisions for each coefficient of the polynomial which goes to a total of $T \cdot n$ comparisions for the whole generation. With a value of T around 200, sampling from the table is actually sensibly heavier than signing. Thus, our goal with this masked `qTESLA` key generation is to prove that masking without changing the design is coslty but still doable. However, for a practical implementation in which the key generation might be vulnerable to side channels, one could prefer changing the design of the scheme. For example, Dilithium generates the keys uniformly at random on a small interval and thus avoids this issue. One downside of this faster key generation is that the parameters of the scheme should be adapted in order to avoid having to much rejections in the signing algorithm.

In Algorithm 13, the element $\mathbf{a}$ is generated in unmasked form because it is also part of the public key. Then, $\mathbf{s}$ and $\mathbf{e}$ are drawn using the gadget GaussGen introduced in Section 3.3. Another gadget FullCheck is also introduced in the key generation. It checks that the sum of the h largest entries (in absolute value) is not above some bounds that can be found in Table 1. Then the public key $\mathbf{t}$ is computed in masked form and securely unmasked with the FullAdd gadget.

Algorithm 13 Masked key generation

Result: Secret key $sk = ((\mathbf{s}_i)_{0\le i\le N}, (\mathbf{e}_i)_{0\le i\le N}, sd)$, public key $pk = (sd, \mathbf{t})$

pre-seed $\xleftarrow{r} \{0,1\}^\kappa$
sd $\leftarrow$ PRF(pre-seed)
$\mathbf{a} \leftarrow \mathsf{GenA}(sd)$
do
    $(\mathbf{s}_i)_{0\le i\le N} \leftarrow$ GaussGen()
while (MaskedCheck$((\mathbf{s}_i)_{0\le i\le N}, S) \neq 0$)
do
    $(\mathbf{e}_i)_{0\le i\le N} \leftarrow$ GaussGen()
while (MaskedCheck$((\mathbf{e}_i)_{0\le i\le N}, E) \neq 0$)
initialize $(sign_i^{\mathbf{s}})_{0\le i\le N}$ and $(sign_i^{\mathbf{e}})_{0\le i\le N}$ as two 1-bit arithmetic masking of either -1 or 1
$(\mathbf{s}_i)_{0\le i\le N} \leftarrow$ SecAnd $((sign_i^{\mathbf{s}})_{0\le i\le N}, (\mathbf{s}_i)_{0\le i\le N})$
$(\mathbf{e}_i)_{0\le i\le N} \leftarrow$ SecAnd $((sign_i^{\mathbf{e}})_{0\le i\le N}, (\mathbf{e}_i)_{0\le i\le N})$
$(\mathbf{t}_i)_{0\le i\le N} \leftarrow \mathbf{a} \cdot (\mathbf{s}_i)_{0\le i\le N} + (\mathbf{e}_i)_{0\le i\le N} \bmod q$
$\mathbf{t} \leftarrow$ FullAdd$((\mathbf{t}_i)_{0\le i\le N})$
$sk \leftarrow ((\mathbf{s}_i)_{0\le i\le N}, (\mathbf{e}_i)_{0\le i\le N}, sd)$
$pk \leftarrow (sd, \mathbf{t})$
return sk, pk

4 Proof of masking

We first list all the known gadgets and new gadgets introduced together with their security properties. They can be found in Table 2.

Table 2. Security properties of the known and new gadgets.

Existing Gadgets			New Gadgets		
Name	Property	Reference	Name	Property	
SecAnd	N-NI	[12], [3]	GenSecArithBoolModq	N-NI	Lemma 13
SecAdd	N-NI	[12], [3]	AbsVal	N-NI	Lemma 14
SecArithBoolModq	N-SNI	[16], [11]	MaskedRound	N-NIo	Lemma 15
SecBoolArith	N-NI	[16], [11]	FullRound	N-NIo	Corollary 16
FullXor	N-NIo	[3]	MaskedWR	N-NIo	Lemma 17
FullAdd	N-NIo	[3]	FullWR	N-NIo	Corollary 18
DataGen	N-NIo	[3]	MaskedRS	N-NIo	Lemma 19
MultAdd	N-NI	[3], denoted H^1	FullRS	N-NIo	Corollary 20
Refresh	N-SNI	[19]	GaussGen	N-NI	Lemma 22
			MaskedCheck	N-NIo	Lemma 21

4.1 Main masking theorem

We introduce a theorem that proves the N-NI property of our masked key generation algorithm.

Theorem 9. *The key generation algorithm presented in Algorithm 13 is N-NIo secure with the public key as public output*[8].

Proof. The overall gadget decomposition of the key generation is in Figure 1. We ommit some of the non sensitive values (like *sd*) for clarity. The gadget GaussGen and MaskedCheck represents the computation of GaussGen iterated while MaskedCheck returns 1. The number of iterations is then a public output that is independant from the final secrets $\mathbf{e}$ or $\mathbf{s}$. The proof is very similar to the proof of the key generation of GLP in [3].

All the gadgets involved are either N-NI secure, N-NIo secure or they do not manipulate sensitive data (see Table 2 for the recap and Section 4.3 for the proofs). We prove that the final composition of all gadgets is N-NI. We assume that an attacker has made $\delta \leq N$ observations. We prove that all these δ observations can be perfectly simulated with the only knowledge of the public values $(\mathbf{a}, \mathbf{t})$.
In the following, we consider the following distribution of the attacker's δ observations: δ_1 observed during the computations of the lower GaussGen and MaskedCheck gadget that produces $(\mathbf{s}_i)_{0\leq i\leq N}$, δ_2 observed during the computations of the upper GaussGen and MaskedCheck gadget that produces $(\mathbf{e}_i)_{0\leq i\leq N}$, δ_3 observed during the computations of the lower SecAnd gadget, , δ_4 observed during the computations of the upper SecAnd, δ_5 observed during the computations of the gadget MultAdd and δ_6 observed during the computations of FullX that produces shares of $(\mathbf{y}_i)_{0\leq i\leq N}$. Some observations can be made on the public values and computations (GenA), their number will not matter in the proof. Finally, we have $\sum_{i=1}^{6} \delta_i \leq \delta$.

We build the proof from right to left, FullX is used at the very end. Since it is N-NIo, all the observations from its call can be simulated with at most δ_6 shares of $\mathbf{t}$. Then MultAdd is aslo N-NI, all the observations from its call can be simulated with at most $\delta_5 + \delta_6$ shares of $\mathbf{e}$ and the same amount for $\mathbf{s}$. Then, the upper (resp. lower) SecAnd is N-NI, so the observations

[8] To ease the reading, the number of restarts inside the key generation is ommited as a public output. Its knowledge is independant from the final output.

from its call can be simulated with at most $\delta_3 + \delta_5 + \delta_6$ shares of $\mathbf{e}$ (resp. $\delta_4 + \delta_5 + \delta_6$ shares of $\mathbf{s}$). With the additional δ_2 (resp. δ_1) observation performed on the upper (resp. the lower) GaussGen and MaskedCheck gadget, the number of observations on each block remains below δ. All the observations can thus be perfectly simulated with the only knowledge of the outputs which concludes the proof.

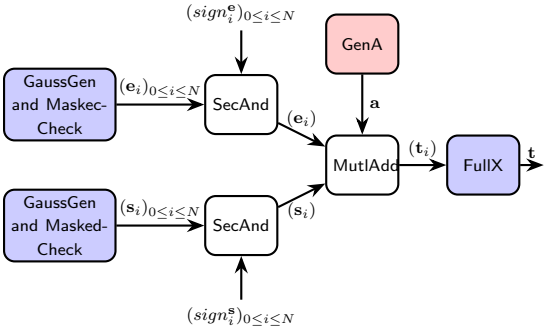

Fig. 1. Masked Key Generation structure (The white (resp. red, blue) gadgets are proved N-NI (resp. unmasked, N-NIo)). The non sensitive element sd is ommited for clarity.

□

In the following, we introduce a theorem that proves the N-NIo property of our masked signature algorithm. For simplicity and without losing generality, the theorem only considers one iteration for the signature: the signing algorithm outputs $\perp$ if one of the tests in Steps 12 or 18 in Algorithm 11 has failed. We also assume the security properties of Table 2are satisfied and refer to Section 4.3 for the proofs. We denote by $\left(r^{(j)}\right)_{0\le j<n}$,$\left(rs^{(j)}\right)_{0\le j<n}$ and $\left(u^{(j)}\right)_{0\le j<n}$ the outputs of FullRS, FullWR and FullRound (the values for each coefficient $j \in [0, n-1]$).

Theorem 10. *Each iteration of the masked signature in Algorithm 11 is N-NIo secure with public outputs*[9]

$$\left\{\left(r^{(j)}\right)_{0\le j<n}, \left(rs^{(j)}\right)_{0\le j<n}, \left(u^{(j)}\right)_{0\le j<n}\right\}$$

(and the signature if returned).

Proof. The overall gadget decomposition of the signature is in Figure 2.

Gadgets. The gadget $\times\mathbf{a}$ multiplies each share of the polynomial $\mathbf{y}$ by the public value $\mathbf{a}$. By linearity, it is N-NI. The gadget FullRound denotes the extension of the MaskedRound to all coefficients of $\mathbf{v}$ and is N-NIo, the proof is below in Corollary 16. The gadget MultAdd takes $(\mathbf{y}_i)_{0\le i\le N}$, $(\mathbf{s}_i)_{0\le i\le N}$ and $\mathbf{c}$ (resp. $(\mathbf{v}_i)_{0\le i\le N}$, $(\mathbf{e}_i)_{0\le i\le N}$ and $\mathbf{c}$) and computes $(\mathbf{z}_i)_{0\le i\le N} = (\mathbf{y}_i)_{0\le i\le N} - \mathbf{c}\cdot(\mathbf{s}_i)_{0\le i\le N}$ (resp. $(\mathbf{w}_i)_{0\le i\le N} = (\mathbf{v}_i)_{0\le i\le N} - \mathbf{c}(\mathbf{e}_i)_{0\le i\le N}$). The gadget End simply outputs $(\mathsf{FullAdd}((\mathbf{z}_i)_{0\le i\le N}), \mathbf{c})$ if rs and r are true; and $\perp$ otherwise. By the N-NIo security of FullAdd, this gadget is also N-NIo secure.

Thus, all the subgadgets involved are either N-NI secure, N-SNI secure, N-NIo secure or they do not manipulate sensitive data (see Table 2 for the recapand Section 4.3 for the proofs).

[9]Here too, the number of iterations of the gadget DG is ommited as a public output.

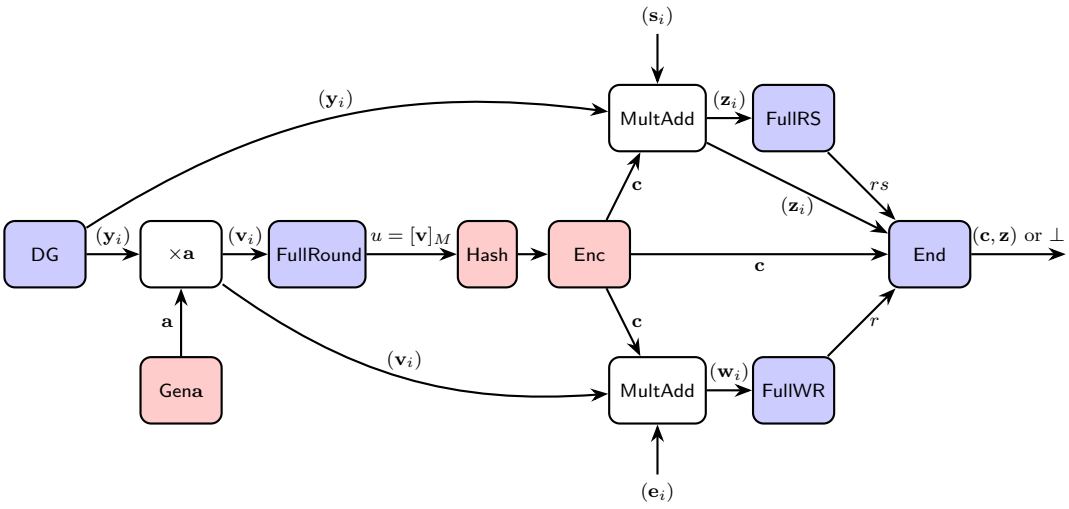

Fig. 2. Masked Signature structure (The white (resp. blue, red) gadgets are proved N-NI (resp. N-NIo, unmasked)). The non sensitive element sd is ommited for clarity.

We prove that the final composition of all gadgets is N-NIo. We assume that an attacker has access to $\delta \leq N$ observations. Our goal is to prove that all these δ observations can be perfectly simulated with at most δ shares of $(\mathbf{s}_i)_{0\leq i\leq N}$ and $(\mathbf{e}_i)_{0\leq i\leq N}$ and the knowledge of the outputs. In the following, we consider the following distribution of the attacker's δ observations:

- δ_1 observed during the computations of DG that produces shares of $(\mathbf{y}_i)_{0\leq i\leq N}$,
- δ_2 observed during the computations of the gadget $\times\mathbf{a}$ that produces the shares of $(\mathbf{v}_i)_{0\leq i\leq N}$,
- δ_3 observed during the computations of FullRound,
- δ_4 observed during the computations of the upper MultAdd gadget that produces $(\mathbf{z}_i)_{0\leq i\leq N}$,
- δ_5 observed during the computations of the lower MultAdd gadget that produces $(\mathbf{w}_i)_{0\leq i\leq N}$,
- δ_6 observed during the FullRS,
- δ_7 observed during the FullWR,
- δ_8 observed during the End.

Some observations may be done on the unmasked gadgets (GenA, Hash and Enc) but their amount will not matter during the proof. Finally, we have $\sum_{i=1}^{8} \delta_i \leq \delta$.

We build the proof from right to left. The gadgets End, FullRS, FullRound and FullWR are N-NIo secure with the output $(\mathbf{z}, \mathbf{c})$ or $\perp$ (resp. $\left(rs^{(j)}\right)_{0\leq j<n}$, $\left(u^{(j)}\right)_{0\leq j<n}$, $\left(r^{(j)}\right)_{0\leq j<n}$). As a consequence, all the observations from their call can be perfectly simulated with at most δ_8 (resp. δ_6, δ_7) shares of $\mathbf{z}$ (resp. $\mathbf{z}$, $\mathbf{w}$). For the upper MultAdd gadget, there are at most $\delta_8 + \delta_6$ observations on the outputs and δ_4 local observations. The total is still lower than δ and thus they can be simulated with at most $\delta_4 + \delta_6 + \delta_8 \leq \delta$ shares of $\mathbf{y}$ and $\mathbf{s}$.
Concerning the lower MultAdd gadget, there are at most δ_7 observations on $\mathbf{w}$ and δ_5 made locally. Thus they can be simulated with at most $\delta_5 + \delta_7 \leq \delta$ shares of $\mathbf{v}$ and $\mathbf{e}$.
The gadget FullRound is N-NIo so all the observations from its call can be simulated with at most δ_3 shares of $\mathbf{v}$. Thus, there are $\delta_3 + \delta_5 + \delta_7$ observations on the output of gadget $\times\mathbf{a}$. And then, they can be simulated with at most $\delta_3 + \delta_5 + \delta_7 + \delta_2$ shares of $\mathbf{y}$. Summing up all the observations of $\mathbf{y}$ gives $(\delta_3 + \delta_5 + \delta_7 + \delta_2) + (\delta_4 + \delta_6 + \delta_8) \leq \delta$. This allows to conlcude the proof by applying the N-NIo security of DG. All the observations on the algorithm can be perfectly simulated with at most $\delta_4 + \delta_6 + \delta_8 \leq \delta$ shares of $\mathbf{s}$, $\delta_5 + \delta_7 \leq \delta$ shares of $\mathbf{e}$ and the knowledge of the public ouputs.

□

4.2 EUF-CMA security in the N-probing model

We recall the EUF-CMA security in the N-probing model. For the complete game description, we refer to [3].

Definition 11. *A signature scheme is EUF-CMA-secure in the N-probing model if any PPT adversary has a negligible probability to forge a signature after a polynomial number of queries to a leaky signature oracle. By leaky signature oracle, we mean that the signature oracle will 1) update the shares of the secret key with a refresh algorithm 2) output a signature together with the leakage of the signature computation.*

Definition 12. *We denote by (r, rs, u)-qTESLA a variant of qTESLA where all the values*

$$\left\{ \left(r^{(j)}\right)_{0\leq j<n}, \left(rs^{(j)}\right)_{0\leq j<n}, \left(u^{(j)}\right)_{0\leq j<n} \right\}$$

are outputted for each iteration during the signing algorithm.

Theorem 10 allows to reduce the EUF-CMA security in the N-probing model of our masked qTESLA signature at order N to the EUF-CMA security of (r, rs, u)-qTESLA. The security of (r, rs, u)-qTESLA is actually not fully supported by the security proof of qTESLA because the adversary is not supposed to see these values for the failed attempts of signing. However, based on the work of [3], we can prove that, under some computational assumptions, outputting $\left(u^{(j)}\right)_{0\leq j<n}$for each iteration does not affect the security. We redirect the reader to [3] for further discussions on this issue. The values $\left\{ \left(r^{(j)}\right)_{0\leq j<n}, \left(rs^{(j)}\right)_{0\leq j<n} \right\}$ correspond to the conditions of rejection, and more precisely, the positions of the coefficients of the polynomials that do not pass the rejections. Such a knowledge do not impact the security of the scheme because the rejection probability does not depend on the position of the coefficients.

4.3 Security proof for the gadgets

Lemma 13. *The gadget* GenSecArithBoolModq *in Algorithm 4 is N-NI secure.*

Indeed, by construction, the security of this gadget is the same as for SecArithBoolModq (proved N-SNI in [9]). The only difference is that we generalize it for N being arbitrary (i.e. non power of two). This still keeps the N-SNI property. In the following, we only need the N-NI property which is automatically implied by N-SNI. We write ABMq to denote GenSecArithBoolModq for short.

Lemma 14. *The gadget* AbsVal *in Algorithm 5 is N-NI secure.*

Proof. A graphical representation of AbsVal is in Figure 3.

We consider that the attacker made $\delta \leq N$ observations. In the following, we prove that all these δ observations can be perfectly simulated with at most δ shares of $(x_i)_{0\leq i\leq N}$.
In the following, we consider the following distribution of the attacker's δ observations: δ_1 observed during the computations of the shift that produces shares of $(mask_i)_{0\leq i\leq N}$, δ_2 observed during the computations of the Resfresh that produces $(x'_i)_{0\leq i\leq N}$, δ_3 observed during the Secadd, and δ_4 observed during the final $\oplus$ and $\wedge$ step. Finally, we have $\sum_{i=1}^{4} \delta_i \leq \delta$.

We build the proof classically from right to left. By linearity for Boolean masking, the final $\oplus$ and $\wedge$ step is N-NI. It is also an affine gadget. In other words, each observation can be simulated

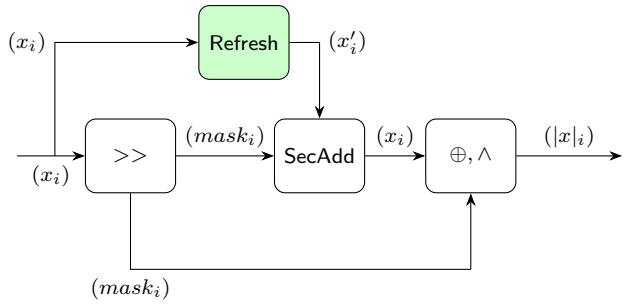

Fig. 3. Masked AbsVal structure (The green (resp. white, blue) gadgets are proved N-SNI (resp. N-NI, N-NIo))

with either one share of x or one share of $mask$. Thus, all the observations from its call can be simulated with at most δ_4 shares among all the shares of x and $mask$. Then it can be simulated with at most x_1 shares of x and x_2 shares of $mask$ with $x_1 + x_2 = \delta_4$.The gadget SecAdd is N-NI then all the observations from its call can be simulated with at most $x_1 + \delta_3$ shares of $mask$ and x'. Identically, the shift is N-NI (by linearity), so the observations from its call can be simulated with at most $\delta_1 + (x_1 + \delta_3) + x_2 = \delta_1 + \delta_3 + \delta_4$ shares of x. By N-SNI security of the lower Resfresh, all the observations from its call can be simulated with at most δ_2 shares of x. Finally, all the observations during the computations of AbsVal can be simulated with at most $\delta_1 + \delta_2 + \delta_3 + \delta_4 \leq \delta$ shares of x.

□

Lemma 15. *The gadget* MaskedRound *in Algorithm 6 is N-NIo with public output u.*

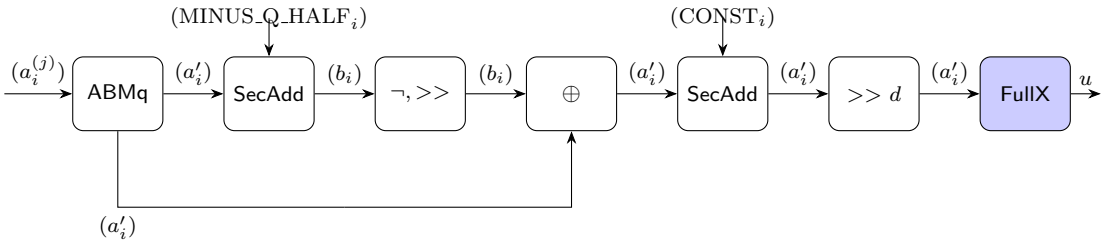

Fig. 4. Masked Rounding structure (The green (resp. white, blue) gadgets are proved N-SNI (resp. N-NI, N-NIo))

Proof. A graphical representation of Algorithm 6 is in Figure 4. Let $\delta \leq N$ be the number of observations made by the attacker. Our goal is to prove that all these δ observations can be perfectly simulated with at most δ shares of $(a_i)_{0 \leq i \leq N}$ and the knowledge of u.
In the following, we consider the following distribution of the attacker's δ observations: δ_1 observed during the computations of ABMq that produces shares of $(a'_i)_{0 \leq i \leq N}$, δ_2 observed during the computations of SecAdd that produces $(b_i)_{0 \leq i \leq N}$, δ_3 observed during the switch and shift steps (steps 5 and 6), δ_4 observed during the $\oplus$ operation, δ_5 observed during the computations of SecAdd that produces $(a'_i)_{0 \leq i \leq N}$, δ_6 observed during the final shift step, and δ_7 observed

during the final FullXor step. Finally, we have $\sum_{i=1}^{7} \delta_i \leq \delta$.

We build the proof from right to left. The algorithm FullXor is N-NIo with public output u. As a consequence, all the observations from its call can be perfectly simulated with at most $\delta_7 \leq \delta$ shares of a' and with the knowledge of u. The shift algorithm is a linear operation and thus it is N-NI secure. Thus, all observations from its call can be perfectly simulated with at most $\delta_6 + \delta_7 \leq \delta$ shares of a' and the knowledge of t. The algorithm SecAdd is N-NI secure and then, similarly, all the observations from its call can be simulated with at most $\delta_5 + \delta_6 + \delta_7 \leq \delta$ shares of a' and $CONST$ (but the latter is a public constant). The $\oplus$ operation is also linear, so it is N-NI. Then, all the observations made from its call can be simulated with at most $\delta_4 + \delta_5 + \delta_6 + \delta_7 \leq \delta$ shares of a' and b, and with the knowledge of u. Actually, we remark that $\oplus$ is also a affine gadget. Thus, all the observations can be exactly simulated with at most x_1 shares of a' and x_2 shares of b such that $x_1 + x_2 = \delta_4 + \delta_5 + \delta_6 + \delta_7$. Let us consider now the switch and shift operations. They are linear so N-NI secure and thus all observations made from its call can be simulated with at most $\delta_3 + x_2 \leq \delta$ observations on b. Considering the first instance of SecAdd, its $N - $ NI security implies that all the observations from its call can be simulated with at most $\delta_2 + \delta_3 + x_2 \leq \delta$ shares of a' and MINUS_Q_HALF (but the latter is a public constant). Finally, we consider the algorithm ABMq which is N-NI secure. There are at most $x_1 + (\delta_2 + \delta_3 + x_2) = \sum_{i=2}^{7} \delta_i$ observations made on the ouputs and δ_1 made locally. Then, all the observations during Algorithm 6 can be simulated with at most $\sum_{i=1}^{8} \delta_i \leq \delta \leq N$ shares of the input $a^{(j)}$ and the knowledge of u.

□

With Lemma 15, one can directly derive the security of FullRound from the security of MaskedRound on each of the polynomial coefficients. Recall that we denote by u_j the application of MaskedRound to the j-th coefficient of the input $\mathbf{v}$ and obtain the following corollary.

Corollary 16. *The gadget* FullRound *is N-NIo secure with public output* $\left(u^{(j)}\right)_{0 \leq j < n}$.

Lemma 17. *The gadget* MaskedWR *in Algorithm 7 is N-NIo secure with public output r.*

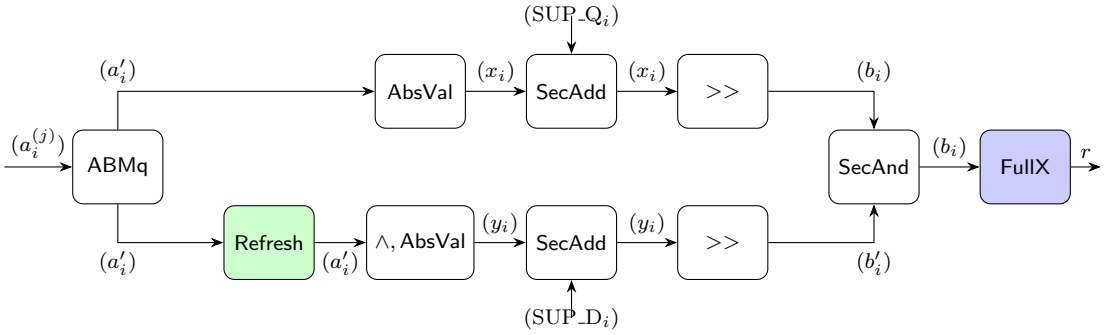

Fig. 5. Masked Well-Rounded structure (The green (resp. white, blue) gadgets are proved N-SNI (resp. N-NI, N-NIo))

Proof. A graphical representation of Algorithm 7 is in Figure 5. Let $\delta \leq N$ be the number of observations made by the attacker. Our goal is to prove that all these δ observations can be

perfectly simulated with at most δ shares of $(a_i)_{0\leq i\leq N}$ and the knowledge of the output r.
In the following, we consider the following distribution of the attacker's δ observations: δ_1 observed during the computation of ABMq that produces shares of $(a'_i)_{0\leq i\leq N}$, δ_2 observed during the computation of the upper AbsVal that produces the shares of $(x_i)_{0\leq i\leq N}$, δ_3 observed during the Refresh, δ_4 observed during the computations of the $\wedge$ and AbsVal that produces the shares of $(y_i)_{0\leq i\leq N}$, δ_5 observed during the SecAdd that produces $(x_i)_{0\leq i\leq N}$, δ_6 observed during the SecAdd that produces $(y_i)_{0\leq i\leq N}$, δ_7 observed during the shift step that produces $(b_i)_{0\leq i\leq N}$, δ_8 observed during the shift step that produces $(b'_i)_{0\leq i\leq N}$, δ_9 observed during the SecAnd, and finally δ_{10} observed during the final FullXor step. Finally, we have $\sum_{i=1}^{10}\delta_i \leq \delta$.

We build the proof from right to left. The algorithm FullXor is N-NIo with public output r. As a consequence, all the observations from its call can be perfectly simulated with at most $\delta_{10} \leq \delta$ shares of b and the knowledge of r. The SecAnd algorithm is N-NI secure. So, all the observations from its call can be perfectly simulated with at most $\delta_9 + \delta_{10} \leq \delta$ shares of b and b' and the knowledge of r. If we look at the lower gadgets of the figure, let us consider the shift that creates b', the SecAdd that creates y and the $\wedge$, AbsVal. All three gadgets are N-NI secure, so all observations at the right side of $\wedge$, AbsVal can be simulated with at most $\delta_4+\delta_6+\delta_8+\delta_9+\delta_{10} \leq \delta$ share of a' and the knowledge of r. We now consider the Refresh algorithm. Since it is N-SNI secure and since the output and local observations are still less than δ, all observations from its call can be perfectly simulated with at most $\delta_3 \leq \delta$ shares of a'. Now let us consider the upper gadgets. The shift that creates b, the SecAdd that creates x and the AbsVal are N-NI secure, so all observations at the right side of AbsVal can be simulated with at most $\delta_2+\delta_5+\delta_7+\delta_9+\delta_{10} \leq \delta$ shares of a' and the knowledge of r. Finally, we consider the algorithm ABMq which is N-NI secure. There are at most $\delta_3 + (\delta_2 + \delta_5 + \delta_7 + \delta_9 + \delta_{10}) \leq \delta$ observations made on the ouputs and δ_1 made locally. Thus, all the observations during MaskedWR can be simulated with at most $\delta_1 + \delta_2 + \delta_3 + \delta_5 + \delta_7 + \delta_9 + \delta_{10} \leq \delta \leq N$ shares of the input $a^{(j)}$ and the knowledge of r. □

Similarly to FullRound, recall that we denote by r_j the application of MaskedWR to the j-th coefficient of the input $\mathbf{w}$ and get the following corollary.

Corollary 18. *The gadget* FullWR *is N-NIo secure with public output $\left(r^{(j)}\right)_{0\leq j<n}$.*

Lemma 19. *The gadget* MaskedRS *in Algorithm 8 is N-NIo secure with public output rs.*

Proof. The rejection sampling is a succession of gadgets without cycle. Thus, prooving its N-NIo security remains to prove the N-NIo or N-NI security of each of its gadgets : ABMq, AbsVal, SecAdd, $>>$ and FullXor. As it is seen in Table 2, ABMq, AbsVal and SecAdd are N-NI. The $>>$ is linear for Boolean masking so it is N-NI. With 2, Table FullXor is N-NIo. Thus, rejection sampling is N-NIo. □

Again, recall that we denote by rs_j the application of MaskedRS to the j-th coefficient the input $\mathbf{z}$ and obtain the following corollary.

Corollary 20. *The gadget* FullRS *is N-NIo secure with public output $\left(rs^{(j)}\right)_{0\leq j<n}$.*

Lemma 21. *Assuming that the bubble sort in step 2 is a N-NI subgadget, the gadget* MaskedCheck *in Algorithm 10 is N-NIo secure with public output ms.*[10]

Proof. Algorithm 10 is a succession of non interferent gadgets with no cycles and the last gadget is N-NIo. Thus, the whole algorithm is N-NIo secure with public output ms. □

[10]This public output is clearly independent from the secret, therefore in the following, it will be ommitted.

Lemma 22. *The gadget* GaussGen *in Algorithm 9 is N-NI secure.*

Proof. A graphical representation of Algorithm 9 is in Figure 6. First, assume that each iteration (without the refresh) is N-NI. Then the algorithm is a linear succession of N-NI and N-SNI gadgets. The only subtelty is that the element (r_i) forms cycles that are broken with the N-SNI property of the refresh.
Now let us show that each iteration denoted iter is N-NI. The result comes from the structure of this subgadget: it has only one cycle which is broken by the affine property of the $\oplus$ gadget.

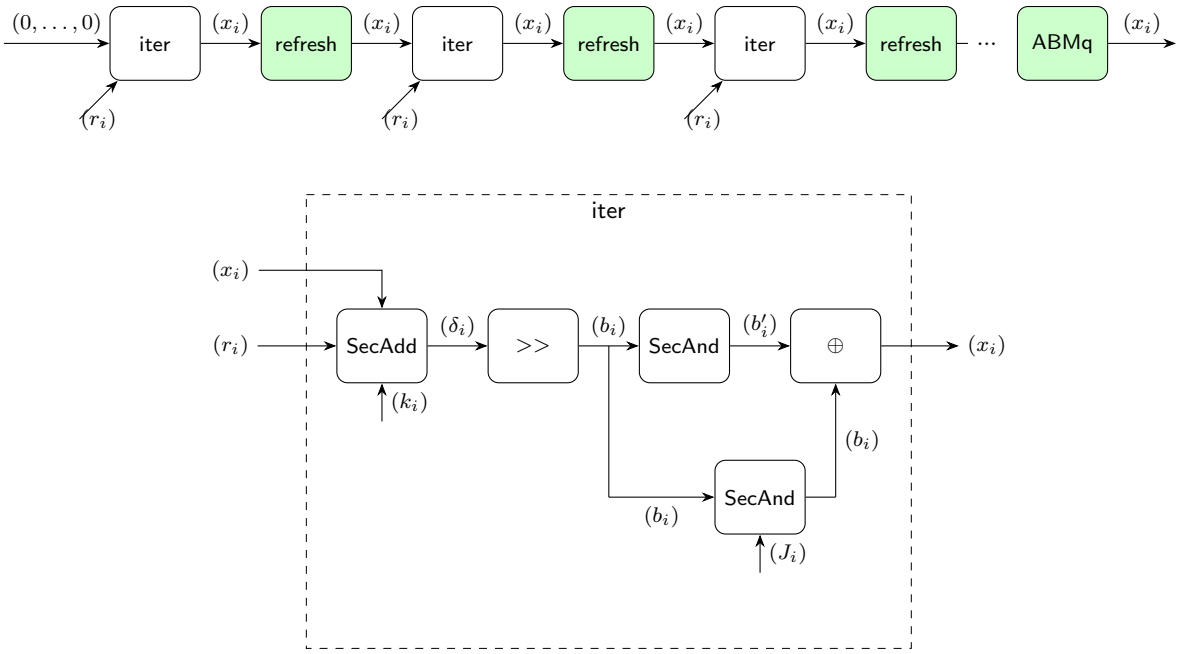

Fig. 6. Masked GaussGen structure (The green (resp. white) gadgets are proved N-SNI (resp. N-NI))

□

5 Practical aspects

Our masking scheme has been implemented inside the reference code of qTESLA available on the repository of their project [29].

5.1 Implementation details

We added two new files called base_gadgets.c and sign_gadgets.c containing all the algorithms manipulating masked values. The actual masked signature (Algorithm 11) is available in sign.c. Beside, some modifications related to the new modulus have been made in various places but the overall structure of the code is the same as before. The random oracle of the signature is implemented with cSHAKE.

Randomness. The generation of random numbers plays an important role in the performances of the scheme since most of the basic gadgets need fresh randomness in the form of unsigned

32-bit integers. Our function retrieving randomness is called `rand_uint32()`. It is defined as a macro in `params.h` in order to easily be disabled for testing purpose. Our tests with the randomness enabled were performed using xoshiro128** [5], a really fast PRNG that has been recently used to speed-up public parameters generation in a lattice-based cryptosystem [6]. One looking for real life application of our technique and believing masking need strong randomness would maybe want to use a cryptographically secure PRNG instead. Another option could be to expand a seed with the already available `cSHAKE` function but as we will see in the next section, it might be pretty expensive as the number of random bytes required grows very fast with the number of shares.

5.2 Performances

We benchmarked our code on a laptop with a CPU Intel Core i7-6700HQ running at 2.60GHz as well as on a cortex-M4 microcontroller for the masking of order 1.

Individual gadgets. The result for individual gadgets over 1 000 000 executions can be found in Table 3. The table is divided in two parts: the top part contains measurements for the signing gadgets implementing functionalities of the signature and the bottom part contains measurements for the base gadgets implementing elementary operations. Unsurprisingly, we see that the most expensive signing gadget is MaskedWR. Indeed, it has to perform two absolute value computations in addition to two comparisons. Nevertheless, an actual substantial overall gain of performances would rather come from an improvement of the conversion from arithmetic to boolean masking since it the slowest base gadget and is used in all signing gadgets. Furthermore, it should be also pointed out that most gadgets have a non negligible dependency on the speed of SecAnd since it is called multiple times in SecAdd which itself appears multiple times in signing gadgets.

Signing procedure. The results for the full signature are given in Tables 4 and 5. Since a large portion of the execution time is spent in calls to the random number generator, we decided to benchmark with and without the PRNG. The mention RNG off means that `rand_uint32()` was set to return 0. The mention RNG on means that `rand_uint32()` was set to return the next value of xoshiro128**. The purpose is to give an idea of how the algorithm itself is scaling, regardless of the speed at which the device is able to provide randomness. At the same time, the discrepancy between the values with and without the RNG underlines how masking schemes of this magnitude are sensitive to randomness sampling. In table 7, we also computed the average number of calls to `rand_uint32()` to see how much randomness is needed for each order. Each call is retrieving a uniformly random 32-bit integer. As expected, this number is growing fast when the masking order is increased. The results for the masked signature at order 1 on cortex-M4 microcontroller are given in Table 6. We speculate that the scaling difference between the microcontroller and the computer is due to the fact that architectural differences matter less for the masking code than for the base signature code. Furthermore, we can see that `qTESLA-III` is scaling better than `qTESLA-I`. Beside the natural variance of the experiments, we explain this result by the fact that increasing the masking order reduces the impact of the polynomial multiplication on the timing of the whole signature in favor of masking operations. Factoring out polynomial operations, `qTESLA-III` is scaling better because the probability of rejection for this parameters set is lower than for `qTESLA-I`. Hence, even if n is twice as large, less than twice the masking operations are performed overall.

Table 3. Median speed of principal gadgets in clock cycles over 1000000 executions

Masking order	Order 1	Order 2	Order 3	Order 4	Order 5
RG	98	410	840	1 328	2 416
MaskedRound	164	1 400	2 454	4 314	6 142
MaskedWR	280	2 080	3 914	6 432	9 034
MaskedRS	178	1 440	2 496	4 432	6 254
SecAdd	44	294	592	870	1 192
SecAnd	20	28	44	70	96
GenSecArithBoolModQ	96	786	1 152	3 148	3 500
SecBoolArith	20	42	108	288	884

Table 4. Median speed of masked signature in clock cycles over 10000 executions for `qTESLA-I` on Intel Core i7-6700HQ running at 2.60GHz

Masking order	Unmasked	Order 1	Order 2	Order 3	Order 4	Order 5
`qTESLA-I` (RNG off)	645 673	2 394 085	7 000 117	9 219 826	16 577 823	24 375 359
`qTESLA-I` (RNG on)	671 169	2 504 204	13 878 830	24 582 943	39 967 191	59 551 027
`qTESLA-I` (RNG on) Scaling	1	×4	×21	×37	×60	×89

As noted in [25], the power of two modulus allows to get a reasonable penalty factor for low masking orders. Without such a modification, the scheme would have been way slower. Besides, our implementation seems to outperform the masked implementation of Dilithium as given in [25]. The timing of our order 1 masking for `qTESLA-I` is around 1.3 ms, and our order 2 is around 7.1 ms. This result comes with no surprise because the unmasked version of `qTESLA` already outperformed Dilithium. However, we do not know if our optimizations on the gadgets could lead to a better performance for a masked Dilithium.

Table 5. Median speed of masked signature in clock cycles over 10000 executions `qTESLA-III`

Masking order	Unmasked	Order 1	Order 2	Order 3	Order 4	Order 5
`qTESLA-III` (RNG off)	1 252 645	4 511 179	9 941 571	14 484 664	25 351 066	34 415 499
`qTESLA-III` (RNG on)	1 318 868	4 138 907	21 932 379	33 520 922	59 668 280	83 289 124
`qTESLA-III` (RNG on) Scaling	1	×3	×17	×25	×45	×63

Table 6. Median speed of masked signature in clock cycles over 1000 executions for `qTESLA-I` on cortex-M4 microcontroller

Masking order	Unmasked	Order 1
`qTESLA-I` CortexM4	11 304 025	23 519 583

Table 7. Average number of calls to `rand_uint32`()

Masking order	Order 1	Order 2	Order 3	Order 4	Order 5
`qTESLA-I`	85 810	1 383 459	2 761 525	4 923 709	7 638 422
`qTESLA-III`	115 392	1 826 545	3 721 800	6 482 130	10 005 714

6 Conclusion

In this paper, we described and implemented a provably secure masked version of the signing procedure of `qTESLA`. This work is part of a common effort from the community to study different aspects of NIST's post-quantum competition candidates. While the masking of `qTESLA` is naturally similar to other Fiat-Shamir lattice-based signatures, some specificities had to be taken into consideration in order to get a fully masked scheme. Unlike previous work, we used state-of-the-art algorithms for all the gadgets and specialized ones for masking of order 1. Furthermore, thanks to small modifications to the scheme itself, namely the removal of the PRF and the usage of a power of two modulus, the cost of masking is reasonable, at least for small orders. This indicates that some design elements that seem to be a good idea for the unprotected scheme might be actually problematic in practice. We backed up these claims by providing benchmarks with a C implementation inside the original code of the designers of the scheme.

Acknowledgements

We thank Sonia Belaïd for interesting insights about the masking proofs. We acknowledge the support of the French Programme d'Investissement d'Avenir under national project RISQ P14158. This work is also partially supported by the European Union's H2020 Programme under PROMETHEUS project (grant 780701). This research has been partially funded by ANRT under the programs CIFRE N 2016/1583.

References

[1] E. Alkim et al. *The Lattice-Based Digital Signature Scheme qTESLA*. Cryptology ePrint Archive, Report 2019/085. https://eprint.iacr.org/2019/085. 2019.

[2] S. Bai and S. D. Galbraith. "An Improved Compression Technique for Signatures Based on Learning with Errors". In: *CT-RSA 2014*. Ed. by J. Benaloh. Vol. 8366. LNCS. Springer, Heidelberg, Feb. 2014, pp. 28–47.

[3] G. Barthe et al. "Masking the GLP Lattice-Based Signature Scheme at Any Order". In: *EUROCRYPT 2018, Part II*. Ed. by J. B. Nielsen and V. Rijmen. Vol. 10821. LNCS. Springer, Heidelberg, 2018, pp. 354–384.

[4] G. Barthe et al. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *ACM CCS 2016*. Ed. by E. R. Weippl et al. ACM Press, Oct. 2016, pp. 116–129.

[5] D. Blackman and S. Vigna. "Scrambled Linear Pseudorandom Number Generators". In: *CoRR* abs/1805.01407 (2018). arXiv: 1805.01407.

[6] J. W. Bos et al. *Fly, you fool! Faster Frodo for the ARM Cortex-M4*. Cryptology ePrint Archive, Report 2018/1116. https://eprint.iacr.org/2018/1116. 2018.

[7] L. G. Bruinderink and P. Pessl. "Differential Fault Attacks on Deterministic Lattice Signatures". In: *IACR TCHES* 2018.3 (2018). https://tches.iacr.org/index.php/TCHES/article/view/7267, pp. 21–43.
[8] J.-S. Coron. *High-Order Conversion From Boolean to Arithmetic Masking*. Cryptology ePrint Archive, Report 2017/252. http://eprint.iacr.org/2017/252. 2017.
[9] J.-S. Coron. "High-Order Conversion from Boolean to Arithmetic Masking". In: *CHES 2017*. Ed. by W. Fischer and N. Homma. Vol. 10529. LNCS. Springer, Heidelberg, Sept. 2017, pp. 93–114.
[10] J.-S. Coron. "Higher Order Masking of Look-Up Tables". In: *EUROCRYPT 2014*. Ed. by P. Q. Nguyen and E. Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 441–458.
[11] J.-S. Coron et al. "Secure Conversion between Boolean and Arithmetic Masking of Any Order". In: *CHES 2014*. Ed. by L. Batina and M. Robshaw. Vol. 8731. LNCS. Springer, Heidelberg, Sept. 2014, pp. 188–205.
[12] J.-S. Coron et al. "Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity". In: *FSE 2015*. Ed. by G. Leander. Vol. 9054. LNCS. Springer, Heidelberg, Mar. 2015, pp. 130–149.
[13] J.-S. Coron et al. "Higher-Order Side Channel Security and Mask Refreshing". In: *FSE 2013*. Ed. by S. Moriai. Vol. 8424. LNCS. Springer, Heidelberg, Mar. 2014, pp. 410–424.
[14] A. Duc et al. "Unifying Leakage Models: From Probing Attacks to Noisy Leakage". In: *EUROCRYPT 2014*. Ed. by P. Q. Nguyen and E. Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 423–440.
[15] L. Ducas et al. "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme". In: *IACR TCHES* 2018.1 (2018). https://tches.iacr.org/index.php/TCHES/article/view/839, pp. 238–268.
[16] L. Goubin. "A Sound Method for Switching between Boolean and Arithmetic Masking". In: *CHES 2001*. Ed. by Çetin Kaya Koç et al. Vol. 2162. LNCS. Springer, Heidelberg, May 2001, pp. 3–15.
[17] D. Goudarzi et al. *Unifying Leakage Models on a Rényi Day*. Cryptology ePrint Archive, Report 2019/138. https://eprint.iacr.org/2019/138. 2019.
[18] T. Güneysu et al. "Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems". In: *CHES 2012*. Ed. by E. Prouff and P. Schaumont. Vol. 7428. LNCS. Springer, Heidelberg, Sept. 2012, pp. 530–547.
[19] Y. Ishai et al. "Private Circuits: Securing Hardware against Probing Attacks". In: *CRYPTO 2003*. Ed. by D. Boneh. Vol. 2729. LNCS. Springer, Heidelberg, Aug. 2003, pp. 463–481.
[20] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *CRYPTO'96*. Ed. by N. Koblitz. Vol. 1109. LNCS. Springer, Heidelberg, Aug. 1996, pp. 104–113.
[21] A. Langlois and D. Stehlé. *Hardness of decision (R)LWE for any modulus*. Cryptology ePrint Archive, Report 2012/091. http://eprint.iacr.org/2012/091. 2012.
[22] V. Lyubashevsky. "Lattice Signatures without Trapdoors". In: *EUROCRYPT 2012*. Ed. by D. Pointcheval and T. Johansson. Vol. 7237. LNCS. Springer, Heidelberg, Apr. 2012, pp. 738–755.
[23] V. Lyubashevsky et al. "On Ideal Lattices and Learning with Errors over Rings". In: *EUROCRYPT 2010*. Ed. by H. Gilbert. Vol. 6110. LNCS. Springer, Heidelberg, 2010, pp. 1–23.
[24] D. Micciancio and M. Walter. "Gaussian Sampling over the Integers: Efficient, Generic, Constant-Time". In: *CRYPTO 2017, Part II*. Ed. by J. Katz and H. Shacham. Vol. 10402. LNCS. Springer, Heidelberg, Aug. 2017, pp. 455–485.
[25] V. Migliore et al. *Masking Dilithium: Efficient Implementation and Side-Channel Evaluation*. Cryptology ePrint Archive, Report 2019/394. https://eprint.iacr.org/2019/394. 2019.
[26] D. M'Raïhi et al. "Computational Alternatives to Random Number Generators". In: *SAC 1998*. Ed. by S. E. Tavares and H. Meijer. Vol. 1556. LNCS. Springer, Heidelberg, Aug. 1999, pp. 72–80.
[27] D. Poddebniak et al. *Attacking Deterministic Signature Schemes using Fault Attacks*. Cryptology ePrint Archive, Report 2017/1014. http://eprint.iacr.org/2017/1014. 2017.
[28] M. Rivain and E. Prouff. "Provably Secure Higher-Order Masking of AES". In: *CHES 2010*. Ed. by S. Mangard and F.-X. Standaert. Vol. 6225. LNCS. Springer, Heidelberg, Aug. 2010, pp. 413–427.
[29] qTESLA team. https://qtesla.org/.