The Lattice-Based Digital Signature Scheme qTESLA

Erdem Alkim[1], Paulo S. L. M. Barreto[2], Nina Bindel[3], Patrick Longa[4] and Jefferson E. Ricardini[5]

[1] Ondokuz Mayis University, Turkey
erdemalkim@gmail.com

[2] University of Washington Tacoma.
pbarreto@uw.edu

[3] Technische Universität Darmstadt, Germany
nbindel@cdc.informatik.tu-darmstadt.de

[4] Microsoft Research, USA
plonga@microsoft.com

[5] University of São Paulo, Brazil
jricardini@larc.usp.br

Abstract. We present qTESLA, a family of post-quantum digital signature schemes based on the ring learning with errors (R-LWE) problem that exhibits several attractive features such as simplicity, high-performance, strong security guarantees against quantum adversaries, and built-in protection against certain side-channel and fault attacks. qTESLA—selected for the first round of NIST's post-quantum cryptography standardization project—consolidates a series of recent proposals of R-LWE-based signature schemes originating in works by Lyubashevsky, and Bai and Galbraith, leading to the best performance among lattice-based signature schemes instantiated against state-of-the-art quantum attacks and implemented with protection against timing and cache side-channels.
We provide full-fledged, constant-time reference and AVX2-optimized implementations that showcase the high-speed and simplicity of our scheme. As part of our implementations, we present an efficient and portable Gaussian sampler that gets by without using floating-point operations and is easily implementable in constant-time. While the Gaussian sampling is solely used in qTESLA's key generation, variants of it are used in most lattice-based primitives and, hence, our approach is of independent interest for other lattice-based implementations.

Keywords: Post-quantum cryptography, lattice-based cryptography, digital signatures, provable security, efficient implementation, Gaussian sampling.

1 Introduction

The potential advent of quantum computers has prompted the cryptographic community to look for *quantum-resistant* alternatives to classical schemes based on factoring and (elliptic curve) discrete logarithm problems. Among the available options, lattice-based cryptography has emerged as one of the most promising branches of quantum-resistant cryptography, as it enables elegant and practical schemes that come with strong security guarantees against quantum attackers.

In this work, we introduce a family of lattice-based digital signature schemes called `qTESLA`, which consolidates a series of recent efforts to design an efficient and provably-secure signature scheme based on the so-called ring learning with errors (R-LWE) problem [LPR10]. Under the `qTESLA` family, we distinguish two variants:

`Heuristic qTESLA`. Parameters are generated according to the hardness level provided by the R-LWE instance that corresponds to a certain `qTESLA` instance, without taking into account the explicit security reduction. Instantiations in this variant feature high-speed with relatively small signature and key sizes.

`Provably-secure qTESLA`. Parameters are generated according to the provided security reduction, i.e., instantiations of the scheme in this case *provably guarantee* a certain security level as long as the corresponding R-LWE instances give a certain hardness level. Thus, these instantiations provide a stronger security argument.

`qTESLA`'s design and implementation bring together the following relevant features:

Simplicity and efficiency. `qTESLA` was designed to be simple and easy to implement, with special emphasis on the most used functions in a signature scheme, namely, signing and verification. In particular, Gaussian sampling, arguably the most complex part of traditional lattice-based signature schemes, is relegated exclusively to key generation. This design approach enables the realization of compact and efficient portable implementations that are easy to scale to support multiple security levels.

Security foundation. `qTESLA` is based on the hardness of the R-LWE problem, and comes accompanied by a tight and explicit security proof in the quantum random oracle model (QROM) [BDF+11], i.e., a quantum adversary is allowed to ask the random oracle in superposition. The explicitness of the reduction enables choosing parameters according to the reduction, while its tightness enables smaller parameters and, thus, better performance when choosing provably-secure parameters.

Flexible choice of parameters. Our *two* `qTESLA` variants, "heuristic" and "provably-secure", allow us to target a wide range of applications, from embedded and high-performance applications to highly sensitive scenarios that require a strong confidence level.

Practical security. `qTESLA` facilitates realizations that are secure against implementation attacks. For example, it supports *constant-time* implementations (i.e., implementations that are secure against timing and cache side-channel attacks by avoiding secret memory access and secret-dependent branching), and is inherently protected against certain simple yet powerful fault attacks.

High speed. `qTESLA` achieves very high performance for the operations that are typically time-critical, namely, signing and verification. This is accomplished at the expense of a moderately more expensive key generation, which is usually performed offline.

Related work. `qTESLA` is the result of a long line of research and consolidates the most relevant features of the prior works. The first work in this line is the signature scheme proposed by Bai and Galbraith [BG14a], which is based on the Fiat-Shamir construction of Lyubashevsky [Lyu12, Lyu09]. The Bai-Galbraith scheme is constructed over standard lattices and comes with a (non-tight) security reduction from the LWE and the short integer solution (SIS) problems in the Random Oracle Model (ROM). Dagdelen *et al.* [DBG+15] presented improvements and the first implementation of the Bai-Galbraith scheme. The scheme was subsequently studied under the name TESLA by Alkim *et al.* [ABB+17], who provided an alternative (tight) security reduction from the LWE problem in the QROM. A variant of TESLA over ideal lattices was derived under the name ring-TESLA [ABB+16]. `qTESLA` is a direct successor of this scheme, with several modifications aimed at improving

its security, correctness and implementation, the most important of which are: `qTESLA` includes a new *correctness requirement* that prevents occasional rejections of valid signatures during ring-TESLA's verification; `qTESLA`'s *security reduction* is proven in the QROM while ring-TESLA's reduction was only given in the ROM; in addition to the provably-secure parameter generation, `qTESLA` includes a new approach to choose parameters, namely **`heuristic`** `qTESLA`, which achieves better performance with reduced signature and key sizes; the *security estimations* of ring-TESLA are not state-of-the-art and are limited to classical algorithms while `qTESLA`'s instantiations are with respect to state-of-the-art classical *and* quantum attacks; the *number of R-LWE samples* in `qTESLA` is flexible, not fixed to two samples as in ring-TESLA, which enables instantiations with better efficiency; and our `qTESLA` implementations are protected against several implementation attacks while known implementations of ring-TESLA are not (e.g., do not run in constant-time). In addition, we note that `qTESLA` follows the standard security practice of generating fresh public polynomials a_i at each keypair generation.
Another variant of the Bai-Galbraith scheme is the recently proposed lattice-based signature scheme Dilithium [DKL+18], which is constructed over module lattices. While `qTESLA` and Dilithium share several properties such as a tight security reduction in the QROM [KLS18], `qTESLA` offers *provably-secure* parameters that are chosen according to this security reduction, in addition to the heuristic parameters also offered by Dilithium. Moreover, Dilithium signatures are deterministic, whereas `qTESLA` signatures are probabilistic and come with built-in protection against some powerful fault attacks such as the simple and easy-to-implement fault attack in [PSS+17, BP18]. It is also important to remark that, in general, side-channel attacks are more difficult to carry out against probabilistic signatures.
Two other signature schemes played a major role in the history of Fiat-Shamir-based lattice-based signature schemes, namely, GLP [GLP12] and BLISS [DDLL13b]. For example, the former scheme was inspirational for some of `qTESLA`'s building blocks, such as the encoding function.
In a separate category we mention other lattice-based signature schemes such as Falcon [FHK+17], pqNTRUSign [CHZ17], and DRS [PSDS17], which are not based on the Fiat-Shamir paradigm. In comparison to `qTESLA`, these schemes follow rather complex design principles and are not as easy to implement. Some of these schemes also have a complicated history in cryptanalysis. For example, Yu and Ducas presented a statistical attack against DRS [YD18]. The same attack idea [NR06, DN12] was also used against pqNTRUSign's predecessor NTRUSign [HHP+03].

As an additional contribution that may have independent interest, we show how to implement an efficient and portable Gaussian sampler for `qTESLA` that only requires integer operations and can be easily written in constant-time. The method combines the well-known technique of cumulative distribution tables (CDT) with Batcher's odd-even mergesort algorithm [Bat68] to sample from a Gaussian distribution; see §5.2 for details. This solves an open problem for `qTESLA`, and other similar schemes, which originally relied on a Gaussian sampler requiring floating-point arithmetic [BLN+16].

Software release. We released our portable and AVX2-optimized implementations as open source: `https://github.com/Microsoft/qTESLA-Library`. The implementation package submitted to NIST's Post-Quantum Cryptography Standardization process is available here: `https://github.com/qtesla/qTesla`.

Outline. After describing some preliminary details in §2, we present the signature scheme in §3. In §4, we describe the security foundation and the proposed parameter sets. We discuss thorough implementation details of the scheme in §5, including the proposed Gaussian sampler and our reference and AVX2-optimized implementations. Finally, §6 gives our experimental results and a comparison with state-of-the-art signature schemes.

Acknowledgments

NB is partially supported by the German Research Foundation (DFG) as part of project P1 within the CRC 1119 CROSSING. JR is partially supported by the joint São Paulo Research Foundation (FAPESP)/Intel Research grant 2015/50520-6 “Efficient Post-Quantum Cryptography for Building Advanced Security Applications” and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001".

2 Preliminaries

2.1 Notation

Rings. Let $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ denote the quotient ring of integers modulo q, and let $\mathcal{R}$ and $\mathcal{R}_q$ denote the rings $\mathbb{Z}[x]/\langle x^n + 1\rangle$ and $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$, respectively. Given $f = \sum_{i=0}^{n-1} f_i x^i \in \mathcal{R}$, we define the reduction of f modulo q to be $(f \bmod q) = \sum_{i=0}^{n-1}(f_i \bmod q)x^i \in \mathcal{R}_q$. Let $\mathbb{H}_{n,h} = \{f \in \mathcal{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i,\ f_i \in \{-1, 0, 1\},\ \sum_{i=0}^{n-1} |f_i| = h\}$, and $\mathcal{R}_{q,[B]} = \{f \in \mathcal{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i,\ f_i \in [-B, B]\}$.

Rounding operators. Let $d \in \mathbb{N}$ and $c \in \mathbb{Z}$. For an even (odd) modulus $m \in \mathbb{Z}_{\geq 0}$, define $c' = c \bmod^{\pm} m$ as the unique element $-m/2 < c' \leq m/2$ (resp. $-\lfloor m/2\rfloor \leq c' \leq \lfloor m/2\rfloor$) such that $c' = c \bmod m$. We then define the functions $[\cdot]_L : \mathbb{Z} \to \mathbb{Z},\ c \mapsto c \bmod^{\pm} 2^d$, and $[\cdot]_M : \mathbb{Z} \to \mathbb{Z},\ c \mapsto (c \bmod^{\pm} q - [c]_L)/2^d$. These function definitions are extended to polynomials by applying the operators to each polynomial coefficient; that is, $[f]_L = \sum_{i=0}^{n-1} [f_i]_L\, x^i$ and $[f]_M = \sum_{i=0}^{n-1} [f_i]_M\, x^i$ for a given $f = \sum_{i=0}^{n-1} f_i x^i \in \mathcal{R}$.

Infinity norm. Given $f \in \mathcal{R}$, the function $\max_k(f)$ returns the k-th largest absolute coefficient of f. That is, if the coefficients of f are reordered as to produce a polynomial g with coefficients ordered (without losing any generality) as $|g_1| \geq |g_2| \geq \ldots \geq |g_n|$, then $\max_i(f) = g_i$. For an element $c \in \mathbb{Z}_q$, we have that $\|c\|_\infty = |c \bmod^{\pm} q|$, and define the infinity norm for a polynomial $f \in \mathcal{R}$ as $\|f\|_\infty = \max_i \|f_i\|_\infty$.

Representation of polynomials and bit strings. We write a given polynomial $f \in \mathcal{R}_q$ as $\sum_{i=0}^{n-1} f_i x^i$ or, in some instances, as the coefficient vector $(f_0, f_1, \ldots, f_{n-1}) \in \mathbb{Z}_q^n$. When it is clear by the context, we represent some specific polynomials with a subscript (e.g., to represent polynomials $a_1, \ldots, a_k$). In these cases, we write $a_j = \sum_{i=0}^{n-1} a_{j,i} x^i$, and the corresponding vector representation is given by $a_j = (a_{j,0}, a_{j,1}, \ldots, a_{j,n-1}) \in \mathbb{Z}_q^n$.
In the case of sparse polynomials $c \in \mathbb{H}_{n,h}$, these polynomials are encoded as the two arrays $pos\_list \in \{0, \ldots, n-1\}^h$ and $sign\_list \in \{-1, 1\}^h$ representing the positions and signs of the nonzero coefficients of c, respectively. We denote this by $c \triangleq \{pos\_list, sign\_list\}$.
In some cases, s-bit strings $r \in \{0,1\}^s$ are written as vectors over the set $\{0, 1\}$, in which an element in the i-th position is represented by r_i. This applies analogously to other sets. Multiple instances of the same set are represented by appending an additional superscript. For example, $\{0,1\}^{s,t}$ corresponds to t s-bit strings each defined over the set $\{0, 1\}$.

Distributions. The centered discrete Gaussian distribution for $c \in \mathbb{Z}$ with standard deviation σ is defined to be $\mathcal{D}_\sigma = \rho_\sigma(c)/\rho_\sigma(\mathbb{Z})$, where $\sigma > 0$, $\rho_\sigma(c) = \exp(\frac{-c^2}{2\sigma^2})$, and $\rho_\sigma(\mathbb{Z}) = 1 + 2\sum_{c=1}^{\infty} \rho_\sigma(c)$. We write $c \leftarrow_\sigma \mathbb{Z}$ to denote sampling of a value c with distribution $\mathcal{D}_\sigma$. For a polynomial $f \in \mathcal{R}$, we write $f \leftarrow_\sigma \mathcal{R}$ to denote sampling each coefficient of f with distribution $\mathcal{D}_\sigma$. For a finite set S, we denote sampling the element s uniformly from S with $s \leftarrow_\$ S$.

2.2 The number theoretic transform (NTT)

Polynomial multiplication over a finite field is one of the fundamental operations in R-LWE based schemes such as qTESLA. In this setting, this operation can be efficiently carried out by satisfying the condition $q \equiv 1 \pmod{2n}$ and, thus, enabling the use of the NTT.

Since qTESLA specifies the generation of the polynomials $a_1, \dots, a_k$ directly in the NTT domain for efficiency purposes (see §5), we need to define polynomials in such a domain. Let ω be a primitive n-th root of unity in $\mathbb{Z}_q$, i.e., $\omega^n \equiv 1 \mod q$, and let ϕ be a primitive $2n$-th root of unity in $\mathbb{Z}_q$ such that $\phi^2 = \omega$. Then, given a polynomial $a = \sum_{i=0}^{n-1} a_i x^i$ the forward transform is defined as

$$\mathsf{NTT} : \mathbb{Z}_q[x]/\langle x^n + 1\rangle \to \mathbb{Z}_q^n, \quad a \mapsto \tilde{a} = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} a_j \phi^j \omega^{ij} \right) x^i,$$

where $\tilde{a} = \mathsf{NTT}(a)$ is said to be in *NTT domain*. Similarly, the inverse transformation of a polynomial $\tilde{a}$ in NTT domain is defined as

$$\mathsf{NTT}^{-1} : \mathbb{Z}_q^n \to \mathbb{Z}_q[x]/\langle x^n + 1\rangle, \quad \tilde{a} \mapsto a = \sum_{i=0}^{n-1} \left(n^{-1} \phi^{-i} \sum_{j=0}^{n-1} \tilde{a}_j \omega^{-ij} \right) x^i.$$

It then holds that $\mathsf{NTT}^{-1}(\mathsf{NTT}(a)) = a$ for all polynomials $a \in \mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$. The polynomial multiplication of a and $b \in \mathcal{R}_q$ can be performed as $a \cdot b = \mathsf{NTT}^{-1}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$, where $\cdot$ is the polynomial multiplication in $\mathcal{R}_q$ and $\circ$ is the coefficient wise multiplication in $\mathbb{Z}_q^n$.

2.3 The ring learning with errors (R-LWE) problem

The security of qTESLA is based on the hardness of the R-LWE problem, which was proposed by Lyubashevsky, Peikert and Regev in 2010 [LPR10]. It can be defined as a *search* or a *decision* problem. Since qTESLA is based on the *decisional* R-LWE problem, we omit the definition of the search version. First, we define the R-LWE distribution.

Definition 1 (R-LWE Distribution)**.** Let $n, q > 0$ be integers, $s \in \mathcal{R}$, and χ be a distribution over $\mathcal{R}$. We define by $\mathcal{D}_{s,\chi}$ the R-LWE distribution which outputs $(a, \langle a, s\rangle + e) \in \mathcal{R}_q \times \mathcal{R}_q$, where $a \leftarrow_\$ \mathcal{R}_q$ and $e \leftarrow \chi$.

Definition 2 (Decisional R-LWE problem R-LWE$_{n,k,q,\chi}$)**.** Let $n, q > 0$ be integers and χ be a distribution over $\mathcal{R}$. Moreover, let $s \leftarrow \chi$ and $\mathcal{D}_{s,\chi}$ be the R-LWE distribution. Given k tuples $(a_1, t_1), \dots, (a_k, t_k)$, the decisional R-LWE problem R-LWE$_{n,k,q,\chi}$ is to distinguish whether $(a_i, t_i) \leftarrow_\$ \mathcal{R}_q \times \mathcal{R}_q$ or $(a_i, t_i) \leftarrow \mathcal{D}_{s,\chi}$ for all i.

Note that the above definition follows the so-called *normal form* of the LWE definition [LPR10]. That is, the secret and error polynomials follow the same distribution, whereas in the original definition the secret is chosen uniformly random over $\mathcal{R}_q$.
In qTESLA χ is instantiated with the centered discrete Gaussian distribution with standard deviation σ.

3 The signature scheme qTESLA

In this section, we describe the signature scheme qTESLA, its most relevant design features, and all the system parameters. We start with the description of the scheme.

3.1 Description of the scheme

qTESLA is parameterized by λ, κ, n, k, q, σ, L_E, L_S, B, d, h, and b_{GenA}; see Table 1 in §3.4 for a detailed description of all the system parameters. The following functions are required for the implementation of the scheme:

- The pseudorandom function $\mathsf{PRF}_1 : \{0,1\}^\kappa \to \{0,1\}^{\kappa, k+3}$, which takes as input a seed $\mathsf{pre\text{-}seed}$ that is κ bits long and maps it to $(k+3)$ seeds of κ bits each.
- The collision-resistant hash function $\mathsf{G} : \{0,1\}^* \to \{0,1\}^{512}$, which maps a message m to a 512-bit string.
- The pseudorandom function $\mathsf{PRF}_2 : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^{512} \to \{0,1\}^\kappa$, which takes as inputs seed_y and the random value r, each κ bits long, and the hash G of a message m, which is 512-bit long, and maps them to the κ-bit seed rand.
- The generation function of the public polynomials $a_1, \ldots, a_k$, $\mathsf{GenA} : \{0,1\}^\kappa \to \mathcal{R}_q^k$, which takes as input the κ-bit seed seed_a and maps it to k polynomials $a_i \in \mathcal{R}_q$.
- The Gaussian sampler function $\mathsf{GaussSampler} : \{0,1\}^\kappa \times \mathbb{Z} \to \mathcal{R}$, which takes as inputs a κ-bit seed $\mathsf{seed} \in \{\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}\}$ and a nonce $\mathsf{counter} \in \mathbb{Z}_{>0}$, and outputs a secret or error polynomial in $\mathcal{R}$ sampled according to the Gaussian distribution $\mathcal{D}_\sigma$. To realize $\mathsf{GaussSampler}$, we propose a simple yet efficient constant-time algorithm. This is described in §5.2.
- The encoding function $\mathsf{Enc} : \{0,1\}^\kappa \to \{0, \ldots, n-1\}^h \times \{-1,1\}^h$. This function encodes a κ-bit hash value c' as a polynomial $c \in \mathbb{H}_{n,h}$. The polynomial c is in turn encoded as the two arrays $pos\_list \in \{0, \ldots, n-1\}^h$ and $sign\_list \in \{-1,1\}^h$ containing the positions and signs of its nonzero coefficients, respectively.
- The sampling function $\mathsf{ySampler} : \{0,1\}^\kappa \times \mathbb{Z} \to \mathcal{R}_{q,[B]}$, which samples a polynomial $y \in \mathcal{R}_{q,[B]}$ taking as inputs a κ-bit seed rand and a nonce $\mathsf{counter} \in \mathbb{Z}_{>0}$.
- The hash-based function $\mathsf{H} : \mathcal{R}_q^k \times \{0,1\}^* \to \{0,1\}^\kappa$. This function takes as inputs k polynomials $v_1, \ldots, v_k \in \mathcal{R}_q$ and computes $[v_1]_M, \ldots, [v_k]_M$. The result is then hashed together with the hash G of a given message m to a string κ bits long.
- The correctness check function checkE, which gets an error polynomial e as input and rejects it if $\sum_{k=1}^h \max_k(e)$ is greater than some bound L_E; see Algorithm 1. The function checkE guarantees the correctness of the signature scheme by ensuring that $\|e_i c\|_\infty \leq L_E$ for $i = 1, \ldots, k$ during key generation, as described in Appendix A.
- The simplification check function checkS, which gets a secret polynomial s as input and rejects it if $\sum_{k=1}^h \max_k(s)$ is greater than some bound L_S; see Algorithm 2. checkS ensures that $\|sc\|_\infty \leq L_S$, which is used to simplify the security reduction.

We are now in position to describe qTESLA's algorithms for key generation, signing and verification, which are depicted in Algorithms 3, 4 and 5, respectively.

Key generation. First, the public polynomials $a_1, \ldots, a_k$ are generated uniformly random distributed over $\mathcal{R}_q$ (lines 2–4) by expanding the seed seed_a using PRF_1. Then, a secret polynomial s is sampled with Gaussian distribution $\mathcal{D}_\sigma$. This polynomial must fulfill the requirement check in checkS (lines 5–8). A similar procedure is followed to sample the secret error polynomials $e_1, \ldots, e_k$. In this case, these polynomials must fulfill the correctness check in checkE (lines 10–13). To generate pseudorandom bit strings during the Gaussian sampling the corresponding value from $\{\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}\}$ is used as seed, and a counter is used as nonce to provide domain separation between the different calls to the sampler. Accordingly, this counter is initialized at 1 and then increased by 1 after each invocation to the Gaussian sampler. Finally, the secret key sk consists of $s, e_1, \ldots, e_k$ and the seeds seed_a and seed_y, and the public key pk consists of seed_a and the polynomials $t_i = a_i s + e_i \mod q$ for $i = 1, \ldots, k$. All the seeds required during key generation are generated by expanding a pre-seed $\mathsf{pre\text{-}seed}$ using PRF_1.

Signature generation. To sign a message m, first a polynomial $y \in \mathcal{R}_{q,[B]}$ is chosen uniformly at random (lines 1–4). To this end, a counter initialized at one is used as nonce, and a random string rand, computed as $\mathsf{PRF}_2(\mathsf{seed}_y, r, \mathsf{G}(m))$ with seed_y, a random string r and the digest $\mathsf{G}(m)$ of the message m, is used as seed. The counter is used to provide domain separation between the different calls to sample y. Accordingly, it is increased by 1 every time the algorithm restarts if any of the security or correctness tests fail to compute a valid signature (see below). Next, seed_a is expanded to generate the polynomials $a_1, \ldots, a_k$ (line 5) which are then used to compute the polynomials $v_i = a_i y \bmod^{\pm} q$ for $i = 1, \ldots, k$ (lines 6–8). Afterwards, the hash-based function H computes $[v_1]_M, \ldots, [v_k]_M$ and hashes these together with the digest $\mathsf{G}(m)$ in order to generate c'. This value is then mapped deterministically to a pseudorandomly generated polynomial $c \in \mathbb{H}_{n,h}$ which is encoded as the two arrays $pos\_list \in \{0, \ldots, n-1\}^h$ and $sign\_list \in \{-1, 1\}^h$ representing the positions and signs of the nonzero coefficients of c, respectively. In order for the *potential* signature $(z \leftarrow sc + y, c')$ at line 11 to be returned by the signing algorithm, it needs to pass a *security* and a *correctness* check, which are described next.

The security check (lines 12–15), also called the *rejection sampling*, is used to ensure that the signature does not leak any information about the secret s. It is realized by checking that $z \notin \mathcal{R}_{q,[B-L_S]}$. If the check fails, the algorithm discards the current pair (z, c') and repeats all the steps beginning with the sampling of y. Otherwise, the algorithm goes on with the correctness check.

The correctness check (lines 18–21) ensures the correctness of the signature scheme, i.e., it guarantees that every valid signature generated by the signing algorithm is accepted by the verification algorithm. It is realized by checking that $\| [w_i]_L \|_\infty < 2^{d-1} - L_E$ and $\|w_i\|_\infty < \lfloor q/2 \rceil - L_E$. If the check fails, the algorithm discards the current pair (z, c') and repeats all the steps beginning with the sampling of y. Otherwise, the algorithm returns the signature (z, c') on m.

Verification. The verification algorithm, upon input of a message m and a signature (z, c'), computes $\{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$, expands seed_a to generate $a_1, \ldots, a_k \in \mathcal{R}_q$ and then computes $w_i = a_i z - b_i c \mod q$ for $i = 1, \ldots, k$. The hash-based function H computes $[w_1]_M, \ldots, [w_k]_M$ and hashes these together with the digest $\mathsf{G}(m)$. If the bit string resulting from the previous computation matches the signature bit string c', and $z \in \mathcal{R}_{q,[B-L_S]}$, the signature is accepted; otherwise, it is rejected.

3.2 Correctness of qTESLA

In this section, we explain the correctness of qTESLA informally; a formal proof can be found in Appendix A.

In general, a signature scheme consisting of a tuple $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ of algorithms is correct if, for every message m in the message space $\mathcal{M}$, we have that

$$\Pr\left[\mathsf{Verify}(\mathrm{pk}, m, \sigma) = 0 : (\mathrm{sk}, \mathrm{pk}) \leftarrow \mathsf{KeyGen}(), \sigma \leftarrow \mathsf{Sign}(\mathrm{sk}, m) \text{ for } m \in \mathcal{M}\right] = 1,$$

where the probability is taken over the randomness of the probabilistic algorithms.

In particular, to guarantee the correctness of qTESLA it must hold that for a signature (z, c') of a message m generated by Algorithm 4: (i) $z \in \mathcal{R}_{q,[B-L_S]}$ and (ii) the output of the hash-based function H at signing (line 9 of Algorithm 4) is the same as the analogous output at verification (line 6 of Algorithm 5). Requirement (i) is ensured by the security check during signing (line 12 of Algorithm 4). To ensure (ii), the correctness check at signing is used (line 18 of Algorithm 4). Essentially, this check ensures that, for $i = 1, \ldots, k$, $[a_i z - t_i c \bmod^{\pm} q]_M = [a_i(y+sc) - (a_i s + e_i)c \bmod^{\pm} q]_M = [a_i y + a_i sc - a_i sc - e_i c \bmod^{\pm} q]_M = [a_i y - e_i c \bmod^{\pm} q]_M = [a_i y \bmod^{\pm} q]_M$.

Algorithm 1 checkE

Require: $e \in \mathcal{R}$
Ensure: $\{0,1\}$ ▷ true, false

```
1: if $\sum_{i=1}^{h} \max_i(e) > L_E$ then
2:     return 1
3: end if
4: return 0
```

Algorithm 2 checkS

Require: $s \in \mathcal{R}$
Ensure: $\{0,1\}$ ▷ true, false

```
1: if $\sum_{i=1}^{h} \max_i(s) > L_S$ then
2:     return 1
3: end if
4: return 0
```

Algorithm 3 qTESLA's key generation

Require: -
Ensure: secret key $sk = (s, e_1, \ldots, e_k, \mathsf{seed}_a, \mathsf{seed}_y)$, and public key $pk = (\mathsf{seed}_a, t_1, \ldots, t_k)$

```
1: counter ← 1                                                                  }
2: pre-seed $\leftarrow_{\$} \{0,1\}^{\kappa}$                                   } Generation of $a_1, \ldots, a_k \in \mathcal{R}_q$.
3: $\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}, \mathsf{seed}_a, \mathsf{seed}_y \leftarrow \mathsf{PRF}_1(\mathsf{pre\text{-}seed})$   }
4: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{seed}_a)$
5: do                                                                           }
6:     GaussSampler($\mathsf{seed}_s$, counter)                                 } Generation of $s \leftarrow_{\sigma} \mathcal{R}$
7:     counter ← counter + 1                                                    } using $\mathsf{seed}_s$.
8: while checkS($s$) $\neq 0$
9: for $i = 1, \ldots, k$ do
10:     do                                                                      }
11:         GaussSampler($\mathsf{seed}_{e_i}$, counter)                        } Generation of $e_1, \ldots, e_k \leftarrow_{\sigma} \mathcal{R}$
12:         counter ← counter + 1                                               } using $\mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}$.
13:     while checkE($e_i$) $\neq 0$
14:     $t_i \leftarrow a_i s + e_i \mod q$
15: end for
16: $sk \leftarrow (s, e_1, \ldots, e_k, \mathsf{seed}_a, \mathsf{seed}_y)$           }
17: $pk \leftarrow (\mathsf{seed}_a, t_1, \ldots, t_k)$                          } Return public and secret key.
18: return $sk$, $pk$                                                           }
```

Algorithm 4 qTESLA's signature generation

Require: message m, and secret key $sk = (s, e_1, \ldots, e_k, \mathsf{seed}_a, \mathsf{seed}_y)$
Ensure: signature (z, c')

```
1: counter ← 1                                              } Sampling of $y \leftarrow_\$ \mathcal{R}_{q,[B]}$.
2: $r \leftarrow_\$ \{0,1\}^\kappa$                         }
3: $\mathsf{rand} \leftarrow \mathsf{PRF}_2(\mathsf{seed}_y, r, \mathsf{G}(m))$   }
4: $y \leftarrow \mathsf{ySampler}(\mathsf{rand}, \mathsf{counter})$               }
5: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{seed}_a)$
6: for $i = 1, \ldots, k$ do
7:     $v_i = a_i y \bmod^{\pm} q$
8: end for
9: $c' \leftarrow \mathsf{H}(v_1, \ldots, v_k, \mathsf{G}(m))$            } Computation of hash value.
10: $c \triangleq \{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$   } Generation of sparse $c$.
11: $z \leftarrow y + sc$                                    } Potential signature $(z, c')$.
12: if $z \notin \mathcal{R}_{q,[B-L_S]}$ then               } Check to ensure security
13:     counter ← counter + 1                                 } (the "rejection sampling").
14:     Restart at step 4                                     }
15: end if
16: for $i = 1, \ldots, k$ do                                } Check to ensure correctness.
17:     $w_i \leftarrow v_i - e_i c \bmod^{\pm} q$            }
18:     if $\|[w_i]_L\|_\infty \geq 2^{d-1} - L_E \vee \|w_i\|_\infty \geq \lfloor q/2 \rfloor - L_E$ then   }
19:         counter ← counter + 1                             }
20:         Restart at step 4                                 }
21:     end if                                                }
22: end for                                                   }
23: return $(z, c')$                                         } Return signature for $m$.
```

Algorithm 5 qTESLA's signature verification

Require: message m, signature (z, c'), and public key $pk = (\mathsf{seed}_a, t_1, \ldots, t_k)$
Ensure: $\{0, -1\}$ ▷ accept, reject signature

```
1: $c \triangleq \{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$
2: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{seed}_a)$
3: for $i = 1, \ldots, k$ do
4:     $w_i \leftarrow a_i z - t_i c \bmod^{\pm} q$
5: end for
6: if $z \notin \mathcal{R}_{q,[B-L_S]} \vee c' \neq \mathsf{H}(w_1, \ldots, w_k, \mathsf{G}(m))$ then
7:     return $-1$                                           } Reject signature $(z, c')$ for $m$.
8: end if
9: return 0                                                  } Accept signature $(z, c')$ for $m$.
```

3.3 Design features

Some of the most relevant design features of `qTESLA` are summarized next.

Simplicity and efficiency. `qTESLA` was designed with simplicity and efficiency in mind. The Gaussian sampling, arguably the most complex function in `qTESLA`, is only required during key generation, while the most used signature functions, i.e., signing and verification, only use very simple arithmetic operations that are easy to implement. This enables the realization of compact and portable implementations that achieve high performance. For instance, our reference implementation written in portable C and supporting all our `heuristic` `qTESLA` parameter sets consists of about 350 lines of code [1]. Despite this com-

[1]This count excludes the parameter-specific packing functions, header files, NTT constants and (c)SHAKE functions.

pactness, qTESLA outperforms all the state-of-the-art post-quantum lattice-based schemes that are implemented in constant-time on, e.g., modern x64 platforms; see §6.

Gaussian sampling during key generation. As stated before, one of the main advantages of qTESLA is that the Gaussian sampler is restricted to key generation. This contributes to the high performance and simplicity of the signing and verification algorithms, and reduces the attack surface to carry out recent timing, cache and power attacks, such as [EFGT17, BHLY16]. Still, we remark that qTESLA only requires a relatively simple, easy-to-implement Gaussian sampler, as demonstrated by the efficient Gaussian sampler described in §5.2.

Probabilistic signatures. qTESLA offers *built-in* defenses against several attack scenarios, thanks to its probabilistic nature. Specifically, the seed used to generate the randomness y is produced by hashing the value seed_y that is part of the secret key, some fresh randomness r and the digest $\mathsf{G}(m)$ of the message m. The use of seed_y makes qTESLA resilient to a catastrophic failure of the Random Number Generator (RNG) during generation of the fresh randomness, protecting against fixed-randomness attacks such as the one demonstrated against Sony's Playstation 3 [CPBS10]. Likewise, the random value r guarantees the use of a fresh y at each signing operation, which increases the difficulty to carry out side-channel attacks against the scheme. Moreover, this fresh y prevents some easy-to-implement but powerful fault attacks against deterministic signature schemes [PSS$^+$17, BP18]; see [BP18, §6] for a relevant discussion. We note that the use of a PRF (in our case, PRF_2) reduces the need for a high-quality source of randomness to generate r.

Compactness of the public key. The key generation, signature generation and verification algorithms expand a seed seed_a, stored as part of the secret and public keys, to generate the public polynomials $a_1, \ldots, a_k$. The use of fresh $a_1, \ldots, a_k$ per keypair makes the introduction of backdoors more difficult and reduces drastically the scope of all-for-the-price-of-one attacks [ADPS16, BLN$^+$16]. Moreover, storing only a seed instead of the full polynomials permits to save bandwidth since we only need κ bits to store seed_a instead of the $kn\lceil\log_2(q)\rceil$ bits that are required to represent the full polynomials.

3.4 Parameter description

qTESLA's system parameters and their corresponding bounds are summarized in Table 1. Let λ be the security parameter, i.e., the targeted bit security of a given instantiation. In the targeted R-LWE setting, we have $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where the dimension n is a power of two, i.e., $n = 2^l$ for $l \in \mathbb{N}$. Let σ be the standard deviation of the centered discrete Gaussian distribution that is used to sample the coefficients of the secret and error polynomials. Depending on the specific function, the parameter κ defines the input and/or output lengths of the hash-based and pseudorandom functions. This parameter is specified to be larger or equal to the security level λ. This is consistent with the use of the hash in a Fiat-Shamir style signature scheme such as qTESLA, for which preimage resistance is relevant while collision resistance is much less. Accordingly, we take the hash size to be enough to resist preimage attacks.
The parameter $b_{\mathsf{GenA}} \in \mathbb{Z}_{>0}$ represents the number of blocks requested in the first call to cSHAKE128 during the generation of the public polynomials $a_1, \ldots, a_k$ (see Algorithm 7 in §5.1). The values of b_{GenA} were chosen as to allow the generation of (slightly) more bytes than are necessary to fill out all the coefficients of the polynomials $a_1, \ldots, a_k$.

Bound parameters and acceptance probabilities. The values L_S and L_E are used to bound the coefficients of the secret and error polynomials in the evaluation functions checkS and checkE, respectively. Bounding the size of those polynomials restricts the size of the key space; accordingly we compensate the security loss by choosing a larger bit hardness as

Table 1: Description and bounds of all the system parameters.

Param.	Description	Requirement
λ	security parameter	-
q_h, q_s	number of hash and sign queries	-
n	dimension ($n-1$ is the poly. degree)	power-of-two
σ	standard deviation of centered discrete Gaussian distribution	-
k	#R-LWE samples	-
q	modulus	$q \equiv 1 \bmod 2n$, $q > 4B$ For provably secure parameters: $q^{nk} \geq \lvert\Delta\mathbb{S}\rvert \cdot \lvert\Delta\mathbb{L}\rvert \cdot \lvert\Delta\mathbb{H}\rvert$, $q^{nk} \geq 2^{4\lambda+nkd} 4q_s^3(q_s+q_h)^2$
h	# of nonzero entries of output elements of Enc	$2^h \cdot \binom{n}{h} \geq 2^{2\lambda}$
κ	output length of hash-based function H and input length of GenA, PRF$_1$, PRF$_2$, Enc, GaussSampler, and ySampler	$\kappa \geq \lambda$
L_E, η_E	bound in checkE	$\eta_E \cdot h \cdot \sigma$
L_S, η_S	bound in checkS	$\eta_S \cdot h \cdot \sigma$
B	determines interval the randomness is chosen from during signing	$B \geq \frac{\sqrt[k\cdot n]{M}+2L_S-1}{2(1-\sqrt[k\cdot n]{M})}$, near a power-of-two
d	number of rounded bits	$\left(1-\frac{2\cdot L_E+1}{2^d}\right)^{k\cdot n} \geq 0.3$, $d > \log_2(B)$
b_{GenA}	number of blocks requested to SHAKE128 for GenA	$b_{\mathsf{GenA}} \in \mathbb{Z}_{>0}$
$\lvert\Delta\mathbb{H}\rvert$		$\sum_{j=0}^{h}\sum_{i=0}^{h-j}\binom{kn}{2i}2^{2i}\binom{kn-2i}{j}2^j$
$\lvert\Delta\mathbb{S}\rvert$	see definition in the text	$(4(B-L_S)+1)^n$
$\lvert\Delta\mathbb{L}\rvert$		$(2^d+1)^{nk}$
δ_z	acceptance probability of z in line 12 during signing	determined experimentally
δ_w	acceptance probability of w in line 18 during signing	determined experimentally
δ_{keygen}	acceptance probability of keypairs during key generation	determined experimentally
sig size	theoretical size of signature [byte]	$\kappa + n(\lceil\log_2(B-L_S)\rceil+1)$
pk size	theoretical size of public key [byte]	$kn(\lceil\log_2(q)\rceil)+\kappa$
sk size	theoretical size of secret key [byte]	$n(k+1)(\lceil\log_2(t\cdot\sigma+1)\rceil)+2\kappa$

explained in §4.2. Both bounds, L_S and L_E, impact the rejection probability during the signature generation as follows. If one increases the values of L_S and L_E, the acceptance probability during key generation, referred to as δ_{keygen}, increases (see lines 8 and 13 in Algorithm 3), while the acceptance probabilities of z and w during signature generation, referred to as δ_z and δ_w resp., decrease (see lines 12 and 18 in Algorithm 4). We determine a good trade-off between the acceptance probabilities during key generation and signing experimentally. To this end, we start by choosing $L_S = \eta_S \cdot h \cdot \sigma$ (resp., $L_E = \eta_E \cdot h \cdot \sigma$) with $\eta_S = \eta_E = 2.8$ and compute the corresponding values for the parameters B, d and q (which are chosen as explained later). We then carefully tune these parameters by trying different values for η_S and η_E in the range $[2.0, \ldots, 3.0]$ until we find a good trade-off between the different probabilities and, hence, runtimes.
The parameter B defines the interval of the random polynomial y (see line 4 of Algorithm 4), and it is determined by the parameters M and L_S as follows:

$$\left(\frac{2B-2L_S+1}{2B+1}\right)^{k\cdot n} \geq M \Leftrightarrow B \geq \frac{\sqrt[k\cdot n]{M}+2L_S-1}{2(1-\sqrt[k\cdot n]{M})},$$

where $M = 0.3$ is a value of our choosing. Once B is chosen, we select the value d that determines the rounding functions $[\cdot]_M$ and $[\cdot]_L$ to be larger than $\log_2(B)$ and such that the acceptance probability of the check $\|[w]_L\|_\infty \geq 2^{d-1} - L_E$ in line 18 of Algorithm 4 is upper bounded by 0.7.
The acceptance probabilities δ_z, δ_w and δ_{keygen} obtained experimentally, following the procedure above, are summarized in Table 2.

The modulus q. This parameter is chosen to fulfill several bounds and assumptions that are motivated by efficiency requirements and `qTESLA`'s security reduction. To enable the use of fast polynomial multiplication using the NTT, q must be a prime integer such that $q \bmod 2n = 1$. Moreover, we choose $q > 4B$. To choose parameters according to the security reduction, i.e., for the case of `provably-secure qTESLA`, it is first convenient to simplify our security statement. To this end we ensure that $q^{nk} \geq |\Delta\mathbb{S}| \cdot |\Delta\mathbb{L}| \cdot |\Delta\mathbb{H}|$ with the following definition of sets: $\mathbb{S}$ is the set of polynomials $z \in \mathcal{R}_{q,[B-U]}$ and $\Delta\mathbb{S} = \{z - z' \ : \ z, z' \in \mathbb{S}\}$, $\mathbb{H}$ is the set of polynomials $c \in \mathcal{R}_{q,[1]}$ with exactly h nonzero coefficients and $\Delta\mathbb{H} = \{c - c' \ : \ c, c' \in \mathbb{H}\}$, and $\Delta\mathbb{L} = \{x - x' : x, x' \in \mathcal{R}$ and $[x]_M = [x']_M\}$. Then, the following equation (see Theorem 1 in §4.1) has to hold:

$$\frac{2^{3\lambda+nkd+2} q_s^3 (q_s + q_h)^2}{q^{nk}} \leq 2^{-\lambda} \Leftrightarrow q \geq \left(2^{4\lambda+nkd+2} q_s^3 (q_s + q_h)^2\right)^{1/nk}.$$

Key and signature sizes. The theoretical bitlengths of the signatures and public keys are given by $\kappa + n \cdot (\lceil \log_2(B - L_S) \rceil + 1)$ and $k \cdot n \cdot (\lceil \log_2(q) \rceil) + \kappa$, respectively. To determine the size of secret keys we note that $Pr_{x \leftarrow_\sigma \mathbb{Z}} [|x| > t\sigma] \leq 2e^{-t^2/2}$ for $t > 0$. Then, it follows that the size of the secret key is given by $n(k+1)(\lceil \log_2(t \cdot \sigma) \rceil + 1) + 2\kappa$ bits.

4 Security and instantiation of the signature scheme

In this section, we discuss the security of `qTESLA` and the security proof in the QROM that reduces the decisional R-LWE problem to the security of the scheme. Afterwards, we elaborate on the relation between the hardness of R-LWE and `qTESLA`'s security, and describe our two approaches to instantiate the scheme. Finally, we explain how we estimate the hardness of R-LWE and propose parameter sets based on the two `qTESLA` variants.

4.1 Provable security in the quantum random oracle model (QROM)

The standard security requirement for signature schemes, namely Existential Unforgeability under Chosen-Message Attack (EUF-CMA), dates back to Goldwasser, Micali, and Rivest [GMR88]: The adversary can obtain q_S signatures via signing oracle queries on messages of their own choosing, and must output one valid signature on a message not queried to the oracle. In the QROM [BDF+11], which we consider for our security statements, the adversary is granted access to a quantum random oracle.

Our main security statement is given in Theorem 1, which gives a tight reduction from the R-LWE problem to the EUF-CMA security of `qTESLA` in the QROM. Currently, Theorem 1 holds assuming a conjecture, as explained below.

Theorem 1. *Let the parameters be as in Table 1. Furthermore, assume that Conjecture 1 holds. If there exists an adversary $\mathcal{A}$ that forges a signature of the signature scheme* `qTESLA` *described in §3 in time t_Σ and with success probability ϵ_Σ, then there exists an algorithm $\mathcal{S}$ that solves the $R\!-\!LWE_{n,k,q,\sigma}$ problem in time $t_{LWE} \approx t_\Sigma$ with $\epsilon_\Sigma \leq \frac{2^{3\lambda+nkd+2} q_s^3 (q_s+q_h)^2}{q^{nk}} + \frac{2q_h+5}{2^\lambda} + \epsilon_{LWE}$.*

The reduction idea follows [ABB+17] that gives the security reduction for `qTESLA`'s successor TESLA. The proof uses the reductionist's approach that assumes the existence of an adversary $\mathcal{A}$ that forges a `qTESLA` signature after some time t_Σ and with probability ϵ_Σ. We then construct an algorithm that solves the (decisional) R-LWE problem in time $t_{LWE} \approx t_\Sigma$ and with a success bias ϵ_{LWE} close to ϵ_Σ. Under the assumption that the R-LWE problem is computationally hard (i.e., that t_{LWE} is large and ϵ_{LWE} is small), it must follow that `qTESLA` is secure (i.e., that ϵ_Σ must be small and t_Σ is large).

Specifically, the idea of the reduction is as follows. Let $\mathcal{A}$ be an algorithm that breaks `qTESLA`, i.e., given an "expanded" `qTESLA` public key $(a_1, \ldots, a_k, t_1, \ldots, t_k)$, algorithm $\mathcal{A}$ outputs (c', σ, m) after some time t_Σ. Let forge$(a_1, \ldots, a_k, t_1, \ldots, t_k)$ denote the event that the forger $\mathcal{A}$ successfully produces a *valid* signature for the public key $(a_1, \ldots, a_k, t_1, \ldots, t_k)$, i.e., with probability $\Pr[\text{forge}(a_1, \ldots, a_k, t_1, \ldots, t_k)]$, (c', σ) is a *valid* signature for message m. We model the hash-based function H as a random oracle. Since our goal is to prove that `qTESLA` is EUF-CMA secure in the QROM, algorithm $\mathcal{A}$ is allowed to make (at most) q_h quantum queries to a quantum random oracle $\mathsf{H}(\cdot)$ and (at most) q_s classical queries to a `qTESLA` signing oracle. However, the message m that is returned by $\mathcal{A}$ must not be queried to the signing oracle. We then build an algorithm $\mathcal{S}$ that solves the decisional R-LWE problem with a runtime that is close to that of $\mathcal{A}$ and with a success bias close to $\Pr[\text{forge}(a_1, \ldots, a_k, t_1, \ldots, t_k)]$.

The solver $\mathcal{S}$ gets as input a tuple $(a_1, \ldots, a_k, t_1, \ldots, t_k)$ and must decide whether the tuple follows the R-LWE distribution (see Definition 1) or the uniform distribution over $\mathcal{R}_q^{2k}$. It forwards its own input tuple $(a_1, \ldots, a_k, t_1, \ldots, t_k)$ as the public key to $\mathcal{A}$. In the reduction, $\mathcal{S}$ must simulate the responses to $\mathcal{A}$'s quantum and classical queries to the hash and sign oracles, respectively. It is then shown, that if $(a_1, \ldots, a_k, t_1, \ldots, t_k)$ follows the R-LWE distribution then the probability with which $\mathcal{S}$ answers *correctly* is close to $\Pr[\text{forge}(a_1, \ldots, a_k, t_1, \ldots, t_k)]$. Furthermore, if $(a_1, \ldots, a_k, t_1, \ldots, t_k)$ follows the uniform distribution over $\mathcal{R}_q^{2k}$ then $\mathcal{S}$ returns the *wrong* answer with negligible probability.

The formal proof follows the approach proposed in [ABB+17] except for the computation of the two probabilities $\text{coll}(a, e)$ and $\text{nwr}(a, e)$ that we explain in the following.

We define $\Delta\mathbb{L}$ to be the set $\{x - x' : x, x' \in \mathcal{R} \text{ and } [x]_M = [x']_M\}$. For any $f \in \mathcal{R}_{q,[B]}$ it holds that all the coefficients are at most B in absolute value; we call such a polynomial B-*short*. In addition, we call f *well-rounded* if it is $(\lfloor q/2 \rfloor - L_E)$-short and $[f]_L$ is $(2^{d-1} - L_E)$-short. Furthermore, we define the following quantities for keys $(a_1, \ldots, a_k, t_1, \ldots, t_k)$, $(s, e_1, \ldots, e_k)$, where we denote $\overrightarrow{a} = (a_1, \ldots, a_k)$ and $\overrightarrow{e} = (e_1, \ldots, e_k)$:

$$\text{nwr}(\overrightarrow{a}, \overrightarrow{e}) \overset{\text{def}}{=} \Pr_{(y,c) \in \mathbb{Y} \times \mathbb{H}} [a_i y - e_i c \text{ not well-rounded for at least one } i \in \{1, \ldots, k\}], \quad (1)$$

$$\text{coll}(\overrightarrow{a}, \overrightarrow{e}) \overset{\text{def}}{=} \max_{(w_1, \ldots, w_k) \in \mathbb{W}^k} \left\{ \Pr_{(y,c) \in \mathbb{Y} \times \mathbb{H}} [[a_1 y - e_1 c]_M = w_1, \ldots, [a_k y - e_k c]_M = w_k] \right\}. \quad (2)$$

Informally speaking $\text{nwr}(\overrightarrow{a}, \overrightarrow{e})$ refers to the probability over random (y, c) that $a_i y - e_i c$ is not well-rounded for some i. This quantity varies as a function of $a_1, \ldots, a_k, e_1, \ldots, e_k$. In contrast to [ABB+17], we cannot upper bound this in general in the ring setting. Hence, we first assume that $\text{nwr}(\overrightarrow{a}, \overrightarrow{e}) < 3/4$ and afterwards check experimentally that this holds true. As our acceptance probability δ_w of w_i at signing (line 18 of Algorithm 4) is at least $1/4$ for all the parameter sets (see Table 2), the bound $\text{nwr}(\overrightarrow{a}, \overrightarrow{e}) < 3/4$ holds.

Secondly, we need to bound the probability $\text{coll}(\overrightarrow{a}, \overrightarrow{e})$. In [ABB+17, Lemma 4] the corresponding probability $\text{coll}(A, E)$ for standard lattices is upper bounded. Unfortunately, we were not able to transfer the proof to the ring setting for the following reason. In the proof of [ABB+17, Lemma 4], it is used that if the randomness y is not equal to 0, the vector Ay is uniformly random distributed over $\mathbb{Z}_q$ and, hence, also $Ay - Ec$ is uniformly random distributed over $\mathbb{Z}_q$. This does not necessarily hold if `qTESLA`'s polynomial y is chosen uniformly in $\mathcal{R}_{q,[B]}$. Moreover, in Equation (99) in [ABB+17], ψ denotes the probability that a random vector $x \in \mathbb{Z}_q^m$ is in $\Delta\mathbb{L}$, i.e.,

$$\psi \overset{\text{def}}{=} \Pr_{x \in \mathbb{Z}_q^m} [x \in \Delta\mathbb{L}] \leq \left(\frac{2^d + 1}{q} \right)^m. \quad (3)$$

The quantity ψ is a function of the TESLA parameters q, m, d, and it is negligibly small. We cannot prove a similar statement for the signature scheme `qTESLA` over ideals. Instead, we *conjecture* the following.

Conjecture 1. *Let I be a nonzero ideal in $\mathcal{R}_q$ and let $r \in \mathcal{R}_q$ be a fixed choice of ring elements. Then, it holds that the probability that $x + r \in \Delta\mathbb{L}$ for a uniformly distributed element $x \leftarrow_\$ I$ is negligibly small.*

The intuition behind our conjecture is as follows. Let ψ_I denote the probability that a random element from the ideal I lands in $\Delta\mathbb{L}$. We know that ψ_I is small when the ideal $I = \mathcal{R}_q$, i.e., a negligibly small fraction of elements from $\mathcal{R}_q$ are in $\Delta\mathbb{L}$. Furthermore, the set $\Delta\mathbb{L}$ appears to have no relationship with the ideal structure of the ring, so it seems reasonable to view each ideal as a "random" subset of $\mathcal{R}_q$ in the following sense: no larger or smaller portion of elements in the ideal I is in $\Delta\mathbb{L}$ than that portion of elements of $\mathcal{R}_q$ that is in $\Delta\mathbb{L}$.
Hence, the corresponding statement described above and needed in [ABB$^+$17, Lemma 4] translates to the following in the case of `qTESLA`. If $y \neq 0$ then $a_i y$ is a uniformly random element of some non-zero ideal I for all i. The polynomial c is fixed and the polynomials $e_1, \ldots, e_k$ are independent of the polynomials $a_1, \ldots, a_k$, and y. Hence, by our conjecture (with $x = a_i y$ and $r = e_i c$) it holds that the probability of Equation (107) in [ABB$^+$17] is negligibly small. Thus, assuming that our conjecture holds true, [ABB$^+$17, Lemma 4] and, hence, the security reduction in [ABB$^+$17] holds for `qTESLA` as well.

Remark 1. Our explanation above assumes an "expanded" public key $(a_1, ..., a_k, t_1, ..., t_k)$. In the description of `qTESLA`, however, the public polynomials $a_1, ..., a_k$ are generated from seed_a which is part of the secret and public key. This assumption can be justified by another reduction in the QROM: assume there exists an algorithm $\mathcal{A}$ that breaks the original `qTESLA` scheme with public key $(\mathsf{seed}_a, t_1, ..., t_k)$. Then we can construct an algorithm $\mathcal{B}$ that breaks a variant of `qTESLA` with "expanded" public key $(a_1, ..., a_k, t_1, ..., t_k)$. To this end, we model $\mathsf{GenA}(\cdot)$ as a (programmable) random oracle. The algorithm $\mathcal{B}$ chooses first $\mathsf{seed}'_a \leftarrow_\$ \{0,1\}^\kappa$ and reprograms $\mathsf{GenA}(\mathsf{seed}'_a) = (a_1, ..., a_k)$. Afterwards, it forwards $(\mathsf{seed}'_a, t_1, ..., t_k)$ as the input tuple to $\mathcal{A}$. Quantum queries to $\mathsf{GenA}(\cdot)$ by $\mathcal{A}$ can be simulated by $\mathcal{B}$ according to the construction of Zhandry based on $2q_h$-wise independent functions [Zha12]. Hence, the assumption above also holds in the QROM.

4.2 `qTESLA` variants: relation between the R-LWE hardness and `qTESLA` security

The security reduction given by Theorem 1 in §4.1, provides an *explicit* reduction from the hardness of the R-LWE problem, enabling the selection of parameters according to this reduction. To offer high flexibility for a wide range of applications, however, we propose *two* different approaches to instantiate `qTESLA`:

`Provably-secure qTESLA.` For this variant, parameters are chosen according to the security reduction provided in Theorem 1. That is, parameters are chosen such that $\epsilon_{LWE} \approx \epsilon_\Sigma$ and $t_\Sigma \approx t_{LWE}$, which guarantees that the bit hardness of the R-LWE instance is *theoretically* the same as the bit security of our signature scheme, by virtue of the security reduction and its tightness. The reduction provably guarantees that the scheme has the selected security level as long as the corresponding R-LWE instance gives the assumed hardness level [2]. This approach provides a *stronger* security argument.

`Heuristic qTESLA.` For this variant, the scheme is instantiated such that the corresponding R-LWE parameters (i.e., n, σ and q) provide an R-LWE instance of a certain hardness. It is then *assumed* that its bit security is *theoretically* the same as the bit hardness of the corresponding R-LWE instance, without taking into account the security reduction. The

[2]We emphasize that our provably-secure parameters are chosen according to their security reductions from R-LWE but not according to reductions from underlying existing worst-case to average-case reductions from SIVP or GapSVP to R-LWE [LPR10].

assumption is that Theorem 1 still holds. So far no attack that exploits this heuristic is known. This approach features high-speed performance and a small memory footprint while requiring relatively compact keys and signatures.

Remark 2. In practical instantiations of `qTESLA`, the bit security does not exactly match the bit hardness of R-LWE (see Table 2). This is because the bit security does not only depend on the bit hardness of R-LWE, but also on the probability of rejected/accepted keypairs and on the security of other building blocks such as the encoding function Enc. First, in all our parameter sets, *heuristic* and *provably-secure*, the key space is reduced by the rejection of polynomials $s, e_1, \ldots, e_k$ with large coefficients via checkE and checkS. In particular, depending on the instantiation the size of the key space is decreased by $\lceil |\log_2(\delta_{\mathsf{KeyGen}})| \rceil$ bits. We compensate this security loss by choosing an R-LWE instance of larger bit hardness. Hence, the corresponding R-LWE instances give at least $\lambda + \lceil |\log_2(\delta_{\mathsf{KeyGen}})| \rceil$ bits of hardness against currently known (classical and quantum) attacks. Finally, we instantiate the encoding function Enc such that it is λ-bit secure.

4.3 Hardness estimation of R-LWE

Since its introduction in [LPR10], it has remained an open question to determine whether the R-LWE problem is as hard as the LWE problem for instances typically used in signature schemes. Several results exist that exploit the structure of some ideal lattices [GGH13, CGS14, CDPR16, ELOS15]. However, up to now, these results do not seem to apply to R-LWE instances that are typically used in practice. Consequently, we assume that the R-LWE problem is as hard as the LWE problem, and estimate the hardness of R-LWE using state-of-the-art attacks against LWE.
Albrecht, Player, and Scott [APS15] presented the *LWE-Estimator*, a software to estimate the hardness of LWE given the matrix dimension n, the modulus q, the relative error rate $\alpha = \frac{\sigma}{q}$, and the number of given LWE samples. The LWE-Estimator determines the hardness against the fastest classical and quantum LWE solvers currently known, i.e., it outputs an upper (conservative) bound on the number of operations an attack needs to break a given LWE instance. In particular, the following attacks are considered in the LWE-Estimator: the meet-in-the-middle exhaustive search, the coded Blum-Kalai-Wassermann algorithm [GJS15], the recent dual lattice attacks in [Alb17], the enumeration approach by Linder and Peikert [LP11], the primal attack [AFG13, BG14b], the Arora-Ge algorithm [AG11] using Gröbner bases [ACF$^+$15], and the latest analysis to compute the block sizes used in the lattice basis reduction BKZ by Albrecht *et al.* [AGVW17]. Moreover, quantum speedups for the sieving algorithm used in BKZ [Laa16, LMP13] are also considered. We use the LWE-Estimator with commit-id `a2296b8` on 2018-10-31 and with the BKZ cost model of $0.265\beta + 16.4 + \log_2(8d)$, where β is the BKZ blocksize and d is the lattice dimension, for the hardness estimation of our parameters.

4.4 Parameter sets

We propose *five* parameter sets corresponding to the two `qTESLA` variants introduced in §4.2: three *heuristic* instantiations called `qTESLA-I`, `qTESLA-III-speed`, and `qTESLA-III-size`, and two *provably-secure* instantiations called `qTESLA-p-I` and `qTESLA-p-III`.
The notation "`I`" and "`III`" indicate that the corresponding parameter sets provide 95 and 160 bits of *post-quantum* security, respectively. We note that the heuristic parameter sets `qTESLA-III-speed` and `qTESLA-III-size` target the same security level but are optimized for different purposes: `qTESLA-III-speed` gives very fast runtimes whereas `qTESLA-III-size` is optimized for small key and signature sizes.
Table 2 summarizes all the parameters of the proposed parameter sets. Following the NIST's call for proposals [Nat16, §4.A.4], we choose the number of classical queries to the

sign oracle to be $q_s = 2^{64}$ for all our parameter sets. Moreover, we choose the number of queries of a hash function to be $q_h = 2^{128}$.
To determine the size of the secret keys, we follow §3.4. For Level-`I` and Level-`III` parameter sets the probability $Pr_{x \leftarrow_{\sigma} \mathbb{Z}}[|x| > t\sigma]$ is less or equal to 2^{-95} and 2^{-160} for $t = 11.6$ and $t = 15$, respectively. These values are then plugged into the equation $n(k+1)(\lceil \log_2(t \cdot \sigma) \rceil + 1) + 2\kappa$ from Table 1 to obtain the secret key sizes displayed in Table 2.

Table 2: Parameters for each of the proposed *heuristic* and *provably-secure* parameter sets with $q_h = 2^{128}$ and $q_s = 2^{64}$; we choose $M = 0.3$.

Param.	`qTESLA-I`	`qTESLA-III-speed`	`qTESLA-III-size`	`qTESLA-p-I`	`qTESLA-p-III`
λ	95	160	160	95	160
κ	256	256	256	256	256
n	512	1 024	1 024	1 024	2 048
σ	22.93	10.2	7.64	8.5	8.5
k	1	1	1	4	5
q	4 205 569 $\approx 2^{22}$	8 404 993 $\approx 2^{23}$	4 206 593 $\approx 2^{22}$	485 978 113 $\approx 2^{29}$	1 129 725 953 $\approx 2^{30}$
h	30	48	48	25	40
L_E, η_E	1 586, 2.223	1 147, 2.34	910, 2.23	554, 2.61	901, 2.65
L_S, η_S	1 586, 2.223	1 233, 2.52	910, 2.23	554, 2.61	901, 2.65
B	$2^{20} - 1$	$2^{21} - 1$	$2^{20} - 1$	$2^{21} - 1$	$2^{23} - 1$
d	21	22	21	22	24
b_{GenA}	19	38	38	108	180
δ_w	0.31	0.43	0.27	0.36	0.32
δ_z	0.49	0.54	0.42	0.80	0.81
δ_{sign}	0.15	0.23	0.11	0.29	0.26
δ_{keygen}	0.64	0.57	0.96	0.57	0.44
sig size [byte]	1,376	2,848	2,720	2 848	6 176
pk size [byte]	1,504	3,104	2,976	14 880	39 712
sk size [byte]	1,344	2,368	2,112	5 184	12 352
classical bit hardness	103	178	180	132	270
quantum bit hardness	96	164	166	123	247

5 Implementation aspects

5.1 Implementation of basic functions

Pseudorandom bit generation. Several functions used for the implementation of `qTESLA` require hashing and pseudorandom bit generation. This functionality is provided by so-called extendable output functions (XOFs). In the case of `qTESLA` we use the XOF function SHAKE [Dwo15] in the realization of the functions G and H, and cSHAKE128 [Kel16] in the realization of the functions GenA and Enc. To implement the functions PRF_1, PRF_2, $\mathsf{ySampler}$ and $\mathsf{GaussSampler}$ implementers are free to pick a cryptographic PRF of their choice. For simplicity purposes, in our implementations we use SHAKE (in the case of PRF_1 and PRF_2) and cSHAKE (in the case of $\mathsf{ySampler}$ and $\mathsf{GaussSampler}$). With the exception of GenA and Enc (which always use cSHAKE128), Level-`I` parameter sets use (c)SHAKE128 and Level-`III` parameter sets use (c)SHAKE256.
For the remainder, we use $\mathsf{XOF}(\mathsf{X}, \mathsf{L}, \mathsf{S})$ to denote a call to a XOF, where X is the input string, L specifies the output length in bytes, and S specifies an *optional* domain separator [3].

[3]The domain separator S is used with cSHAKE, but ignored when SHAKE is used.

Algorithm 6 Seed generation, PRF_1

```
Require: $\mathsf{pre\text{-}seed} \in \{0,1\}^\kappa$
Ensure: $(\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}, \mathsf{seed}_a)$, where each seed is $\kappa$ bits long
1: $\langle\mathsf{seed}_s\rangle \| \langle\mathsf{seed}_{e_1}\rangle \| \ldots \| \langle\mathsf{seed}_{e_k}\rangle \| \langle\mathsf{seed}_a\rangle \| \langle\mathsf{seed}_y\rangle \leftarrow \mathsf{XOF}(\mathsf{pre\text{-}seed}, \kappa \cdot (k+3)/8)$, where each
   $\langle\mathsf{seed}\rangle \in \{0,1\}^\kappa$
2: return $(\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}, \mathsf{seed}_a)$
```

Algorithm 7 Generation of public polynomials a_i, GenA

```
Require: $\mathsf{seed}_a \in \{0,1\}^\kappa$. Set $b = \lceil(\log_2 q)/8\rceil$ and the SHAKE128 rate constant $\mathrm{rate}_{\mathsf{XOF}} = 168$
Ensure: $a_i \in \mathcal{R}_q$ for $i = 1, \ldots, k$
1: $S \leftarrow 0,\ b' \leftarrow b_{\mathsf{GenA}}$
2: $\langle c_0\rangle \| \langle c_1\rangle \| \ldots \| \langle c_T\rangle \leftarrow$ cSHAKE128$(\mathsf{seed}_a, \mathrm{rate}_{\mathsf{XOF}} \cdot b', S)$, where each $\langle c_t\rangle \in \{0,1\}^{8b}$
3: $i \leftarrow 0$, $pos \leftarrow 0$
4: while $i < k \cdot n$ do
5:     if $pos > \lfloor(\mathrm{rate}_{\mathsf{XOF}} \cdot b')/b\rfloor - 1$ then
6:         $S \leftarrow S + 1,\ pos \leftarrow 0,\ b' \leftarrow 1$
7:         $\langle c_0\rangle \| \langle c_1\rangle \| \ldots \| \langle c_T\rangle \leftarrow$ cSHAKE128$(\mathsf{seed}_a, \mathrm{rate}_{\mathsf{XOF}} \cdot b', S)$, where each $\langle c_t\rangle \in \{0,1\}^{8b}$
8:     end if
9:     if $q > c_{pos} \bmod 2^{\lceil\log_2 q\rceil}$ then
10:        $a_{\lfloor i/n\rfloor+1, i-n\cdot\lfloor i/n\rfloor} \leftarrow c_{pos} \bmod 2^{\lceil\log_2 q\rceil}$, where a polynomial $a_x$ is interpreted as a vector
   of coefficients $(a_{x,0}, a_{x,1}, \ldots, a_{x,n-1})$
11:        $i \leftarrow i + 1$
12:    end if
13:    $pos \leftarrow pos + 1$
14: end while
15: return $(a_1, \ldots, a_k)$
```

Generation of seeds, PRF_1. qTESLA requires the generation of the seeds seed_s, seed_{e_1}, $\ldots$, seed_{e_k}, seed_a, and seed_y during key generation. These seeds, of κ bits each, are then used to produce the polynomials s, e_1, $\ldots$, e_k, a_1, $\ldots$, a_k, and y, respectively. In our implementations, the seeds are generated by first calling the system RNG to produce a pre-seed of size κ bits (line 2 of Algorithm 3), and then expanding this pre-seed using SHAKE as the XOF function; see Algorithm 6.

Generation of $\mathbf{a_1, \ldots, a_k}$. The procedure to generate $a_1, \ldots, a_k$ is as follows. The seed seed_a produced by PRF_1 is expanded to $(\mathrm{rate}_{\mathsf{XOF}} \cdot b_{\mathsf{GenA}})$ bytes using cSHAKE128, where $\mathrm{rate}_{\mathsf{XOF}}$ is the SHAKE128 rate constant 168 [Dwo15] and b_{GenA} is a qTESLA parameter (see §3.4). Then, the algorithm proceeds to do rejection sampling over each $8 \cdot \lceil\log_2(q)\rceil$-bit string of the cSHAKE output modulo $2^{\lceil\log_2(q)\rceil}$, discarding every package that has a value equal or greater than the modulus q. Since there is a possibility that the cSHAKE output is exhausted before all the $k \cdot n$ coefficients are filled out, the algorithm permits successive (and as many as necessary) calls to the function requesting $\mathrm{rate}_{\mathsf{XOF}}$ bytes each time.
The procedure above, which is depicted in Algorithm 7, produces polynomials with uniformly random coefficients. Thus, following a standard practice, qTESLA assumes that the resulting polynomials $a_1, \ldots, a_k$ are already in the NTT domain, eliminating the need of their NTT conversion during the polynomial multiplications. This permits an important speedup of the polynomial operations without affecting security.
It should be noted that the value $S = 0$ is used as domain separator in the first call to cSHAKE128 in Algorithm 7. This value is incremented by one at each subsequent call, if required.

Algorithm 8 Sampling y, ySampler

Require: seed rand $\in \{0,1\}^\kappa$ and nonce $S \in \mathbb{Z}_{>0}$. Set $b = \lceil(\log_2 B + 1)/8\rceil$
Ensure: $y \in \mathcal{R}_{q,[B]}$

```
 1: $pos \leftarrow 0,\ n' \leftarrow n,\ S' \leftarrow S \cdot 2^8$
 2: $\langle c_0\rangle \| \langle c_1\rangle \| \ldots \| \langle c_{n'-1}\rangle \leftarrow \mathsf{XOF}(\mathsf{rand}, b \cdot n', S')$, where each $\langle c_i\rangle \in \{0,1\}^{8b}$
 3: while $i < n$ do
 4:     if $pos \geq n'$ then
 5:         $S' \leftarrow S' + 1,\ pos \leftarrow 0,\ n' \leftarrow \lfloor \mathrm{rate}_{\mathsf{XOF}}/b \rfloor$
 6:         $\langle c_0\rangle \| \langle c_1\rangle \| \ldots \| \langle c_{n'-1}\rangle \leftarrow \mathsf{XOF}(\mathsf{rand}, \mathrm{rate}_{\mathsf{XOF}}, S')$, where each $\langle c_i\rangle \in \{0,1\}^{8b}$
 7:     end if
 8:     $y_i \leftarrow c_{pos} \bmod 2^{\lceil \log_2 B\rceil + 1} - B$
 9:     if $y_i \neq B + 1$ then
10:         $i \leftarrow i + 1$
11:     end if
12:     $pos \leftarrow pos + 1$
13: end while
14: return $y = (y_0, y_1, \ldots, y_{n-1}) \in \mathcal{R}_{q,[B]}$
```

Sampling of y. The sampling of the polynomial y (line 4 of Algorithm 4) can be performed by generating n ($\lceil \log_2 B\rceil + 1$)-bit values uniformly at random, and then correcting each value to the range $[-B, B+1]$ with a subtraction by B. Since values need to be in the range $[-B, B]$, coefficients with value $B + 1$ need to be rejected, which in turn might require the generation of additional random bits.
For the pseudorandom bit generation, the seed rand produced by PRF_2 (see line 3 of Algorithm 4) is used as input string to a XOF, while the nonce S (written as counter in Algorithm 4) is intended for the computation of the values for the domain separation.
Algorithm 8 depicts the procedure used in our implementations. The first call to the XOF function uses the value $S' \leftarrow S \cdot 2^8$ as domain separator. Each subsequent call to the XOF increases S' by 1. Since S is initialized at 1 by the signing algorithm, and then increased by 1 at each subsequent call to sample y, the successive calls to the sampler use nonces S' initialized at $2^8, 2 \cdot 2^8, 3 \cdot 2^8$, and so on, providing proper domain separation between the different uses of the XOF in the signing algorithm.
Our implementations use cSHAKE as the XOF function.

Hash-based function H. This function takes as inputs k polynomials $v_1, \ldots, v_k \in \mathcal{R}_q$ and computes $[v_1]_M, \ldots, [v_k]_M$. The result is hashed together with the hash G of a message m to a string c' that is κ bits long. The detailed procedure is as follows. Let each polynomial v_i be interpreted as a vector of coefficients $(v_{i,0}, v_{i,1}, \ldots, v_{i,n-1})$, where $v_{i,j} \in (-q/2, q/2]$, i.e., $v_{i,j} = v_{i,j} \bmod^{\pm} q$. We first compute $[v_i]_L$ by reducing each coefficient modulo 2^d and subtracting the result by 2^d if it is greater than 2^{d-1}. This guarantees a result in the range $(-2^{d-1}, 2^{d-1}]$, as required by the definition of $[\cdot]_L$. Next, we compute $[v_i]_M$ as $(v_i - [v_i]_L)/2^d$. Since each resulting coefficient is guaranteed to be very small it is stored as a byte, which in total makes up a string of $k \cdot n$ bytes. Finally, SHAKE is used to hash the resulting $k \cdot n$-byte string together with the 64-byte digest $\mathsf{G}(m)$ to the κ-bit string c'. This procedure is depicted in Algorithm 9.

Encoding function. This function maps the bit string c' to a polynomial $c \in \mathbb{H}_{n,h} \subset \mathcal{R}_q$ of degree $n - 1$ with coefficients in $\{-1, 0, 1\}$ and weight h, i.e., c has h coefficients that are either 1 or -1. For efficiency, c is encoded as two arrays *pos_list* and *sign_list* that contain the positions and signs of its nonzero coefficients, respectively.
For the implementation of the encoding function Enc we follow [DDLL13c, ABB+16]. Basically, the idea is to use a XOF to generate values uniformly at random that are

Algorithm 9 Hash-based function H

Require: polynomials $v_1, \ldots, v_k \in \mathcal{R}_q$, where $v_{i,j} \in (-q/2, q/2]$, for $i = 1, \ldots, k$ and $j = 0, \ldots, n-1$, and the hash G of a message m, $\mathsf{G}(m)$, of length 64 bytes.
Ensure: $c' \in \{0,1\}^\kappa$

```
1: for $i = 1, 2, \ldots, k$ do
2:     for $j = 0, 1, \ldots, n-1$ do
3:         val $\leftarrow v_{i,j} \bmod 2^d$
4:         if val $> 2^{d-1}$ then
5:             val $\leftarrow$ val $- 2^d$
6:         end if
7:         $w_{(i-1)\cdot n+j} \leftarrow (v_{i,j} - \text{val})/2^d$
8:     end for
9: end for
10: $\langle w_{k\cdot n}\rangle \| \langle w_{k\cdot n+1}\rangle \| \ldots \| \langle w_{k\cdot n+63}\rangle \leftarrow \mathsf{G}(m)$, where each $\langle w_i\rangle \in \{0,1\}^8$
11: $c' \leftarrow \text{SHAKE}(w, \kappa/8)$, where $w$ is the byte array $\langle w_0\rangle \| \langle w_1\rangle \| \ldots \| \langle w_{k\cdot n+63}\rangle$
12: return $c' \in \{0,1\}^\kappa$
```

interpreted as the positions and signs of the h nonzero entries of c. The outputs are stored as entries to the two arrays *pos_list* and *sign_list*.
The pseudocode of our implementation of this function is depicted in Algorithm 10. This works as follows.

The algorithm first requests $\text{rate}_{\mathsf{XOF}}$ bytes from a XOF, and the output stream is interpreted as an array of 3-byte packets in little endian format. Each 3-byte packet is then processed as follows, beginning with the least significant packet. The $\lceil \log_2 n \rceil$ least significant bits of the lowest two bytes in every packet are interpreted as an integer value in little endian representing the position *pos* of a nonzero coefficient of c. If such value already exists in the *pos_list* array, the 3-byte packet is rejected and the next packet in line is processed; otherwise, the packet is accepted, the value is added to *pos_list* as the position of a new coefficient, and then the third byte is used to determine the coefficient's sign as follows. If the least significant bit of the third byte is 0, the coefficient is assumed to be positive $(+1)$, otherwise, it is taken as negative (-1). In our implementations, *sign_list* encodes positive and negative coefficients as 0's and 1's, respectively.
The procedure above is executed until *pos_list* and *sign_list* are filled out with h entries each. If the XOF output is exhausted before completing the task then additional calls are invoked, requesting $\text{rate}_{\mathsf{XOF}}$ bytes each time. `qTESLA` uses cSHAKE128 as the XOF function, with the value $S = 0$ as domain separator for the first call. S is incremented by one at each subsequent call.

Polynomial multiplication and the number theoretic transform. As mentioned earlier, the outputs $a_1, \ldots, a_k$ of GenA are assumed to be in NTT domain. In particular, let $\tilde{a_i}$ be the output a_i in NTT domain. Polynomial multiplications $a_i \cdot b$ can be efficiently realized as $\mathsf{NTT}^{-1}(\tilde{a_i} \circ \mathsf{NTT}(b))$.
To compute the NTT in our implementations, we adopt butterfly algorithms that efficiently merge the powers of ϕ and ϕ^{-1} with the powers of ω (see §2.2), and that at the same time avoid the need of the so-called bit-reversal operation which is required by some implementations [PG14, RVM$^+$14, ADPS16]. Specifically, we use an algorithm that computes the forward NTT based on the Cooley-Tukey butterfly that absorbs the products of the root powers in bit-reversed ordering. This algorithm receives the inputs of a polynomial a in standard ordering and produces a result in bit-reversed ordering. Similarly, for the inverse NTT we use an algorithm based on the Gentleman-Sande butterfly that absorbs the inverses of the products of the root powers in the bit-reversed ordering. The

Algorithm 10 Encoding function, Enc

Require: $c' \in \{0,1\}^\kappa$
Ensure: arrays $pos\_list \in \{0,\dots,n-1\}^h$ and $sign\_list \in \{-1,1\}^h$ containing the positions and signs, resp., of the nonzero elements of $c \in \mathbb{H}_{n,h}$

```
1:  $S \leftarrow 0$, $cnt \leftarrow 0$
2:  $\langle r_0\rangle \| \langle r_1\rangle \| \dots \| \langle r_T\rangle \leftarrow$ cSHAKE128$(c', \text{rate}_{\mathsf{XOF}}, S)$, where each $\langle r_t\rangle \in \{0,1\}^8$
3:  $i \leftarrow 0$
4:  Set all coefficients of $c$ to 0
5:  while $i < h$ do
6:      if $cnt > (\text{rate}_{\mathsf{XOF}} - 3)$ then
7:          $S \leftarrow S+1$, $cnt \leftarrow 0$
8:          $\langle r_0\rangle \| \langle r_1\rangle \| \dots \| \langle r_T\rangle \leftarrow$ cSHAKE128$(c', \text{rate}_{\mathsf{XOF}}, S)$, where each $\langle r_t\rangle \in \{0,1\}^8$
9:      end if
10:     $pos \leftarrow (r_{cnt} \cdot 2^8 + r_{cnt+1}) \bmod n$
11:     if $c_{pos} = 0$ then
12:         if $r_{cnt+2} \bmod 2 = 1$ then
13:             $c_{pos} \leftarrow -1$
14:         else
15:             $c_{pos} \leftarrow 1$
16:         end if
17:         $pos\_list_i \leftarrow pos$
18:         $sign\_list_i \leftarrow c_{pos}$
19:         $i \leftarrow i+1$
20:     end if
21:     $cnt \leftarrow cnt + 3$
22: end while
23: return $\{pos\_list_0, \dots, pos\_list_{h-1}\}$ and $\{sign\_list_0, \dots, sign\_list_{h-1}\}$
```

algorithm receives the inputs of a polynomial $\tilde{a}$ in the bit-reversed ordering and produces an output in standard ordering. Efficient versions of these algorithms, which we follow for our implementations, can be found in [Sei18, Algorithm 1 and 2].

Algorithm 11 Sparse Polynomial Multiplication

Require: $g = \sum_{i=0}^{n-1} g_i x^i \in \mathcal{R}_q$ with $g_i \in \mathbb{Z}_q$, and list arrays $pos\_list \in \{0,\dots,n-1\}^h$ and $sign\_list \in \{-1,1\}^h$ containing the positions and signs, resp., of the nonzero elements of a polynomial $c \in \mathbb{H}_{n,h}$
Ensure: $f = g \cdot c \in \mathcal{R}_q$

```
1:  Set all coefficients of $f$ to 0
2:  for $i = 0, \dots, h-1$ do
3:      $pos \leftarrow pos\_list_i$
4:      for $j = 0, \dots, pos-1$ do
5:          $f_j \leftarrow f_j - sign\_list_i \cdot g_{j+n-pos}$
6:      end for
7:      for $j = pos, \dots, n-1$ do
8:          $f_j \leftarrow f_j + sign\_list_i \cdot g_{j-pos}$
9:      end for
10: end for
11: return $f$
```

Sparse multiplication. While standard polynomial multiplications can be efficiently carried out using the NTT as explained above, *sparse multiplications* with a polynomial $c \in \mathbb{H}_{n,h}$, which only contain h nonzero coefficients in $\{-1, 1\}$, can be realized more efficiently with a specialized algorithm that exploits the sparseness of the input. In our implementations we use Algorithm 11 to realize the multiplications in lines 11 and 17 of Algorithm 4 and in line 4 of Algorithm 5, which have as inputs a given polynomial $g \in \mathcal{R}_q$ and a polynomial $c \in \mathbb{H}_{n,h}$ encoded as the position and sign arrays *pos_list* and *sign_list* (as outputted by Enc).

5.2 An efficient and portable constant-time Gaussian sampler

Gaussian sampling has received significant attention in the last few years given its relevance in the design of lattice-based schemes [DG14, BLN+16, HKR+18, ZSS18]. A well-established technique is based on the cumulative distribution table (CDT) of the normal distribution, which consists of precomputing, to a given β-bit precision, a table $\mathsf{CDT}[i] := \lfloor 2^{\beta} \Pr[c \leqslant i \mid c \leftarrow_{\sigma} \mathbb{Z}] \rfloor$, for $i \in [-t+1 \ldots t-1]$ with the smallest t such that $\Pr[|c| \geqslant t \mid c \leftarrow_{\sigma} \mathbb{Z}] < 2^{-\beta}$. To obtain a Gaussian sample, one picks a uniform sample $u \leftarrow_{\$} \mathbb{Z}/2^{\beta}\mathbb{Z}$, looks it up in the table, and returns the value z such that $\mathsf{CDT}[z] \leqslant u < \mathsf{CDT}[z+1]$.
A CDT-based approach has apparently first been considered for cryptographic purposes by Peikert [Pei10] (in a somewhat more complex form). The approach was assessed and deemed mostly impractical by Ducas *et al.* [DDLL13a], since it would take $\beta t \sigma$ bits. Yet, they only considered a scenario where the standard deviation σ was at least 107, and as high as 271. As a result, table sizes around 78 Kbytes are reported (presumably for $\sigma = 271$ with roughly 160-bit sampling precision). For the qTESLA parameter sets, however, the values of σ are much smaller, making the CDT approach feasible, as one can see in Table 3.

Table 3: CDT dimensions (precision in bits : size in bytes).

qTESLA-I	qTESLA-III-speed	qTESLA-III-size	qTESLA-p-I	qTESLA-p-III
96 : 3072	160 : 2980	160 : 2240	96 : 1152	160 : 2500
64 : 1672	128 : 2160	128 : 1616	64 : 632	128 : 1792

The naïve approach to CDT-based sampling would be to perform table lookups via binary search, but this is susceptible to side-channel attacks since the branching depends on the private uniform samples. To prevent this, two techniques are possible:

- On platforms where a reasonably large number of Gaussian samples can be generated at once, one can sort a list of uniformly random samples together with the CDT itself, then identify the CDT entries between which each sample is located. The cost of sorting, which can be implemented in a constant-time fashion using, e.g., Batcher's odd-even mergesort [Bat68] (also called merge-exchange sorting [Knu97, §5.2.2 Algorithm M (Merge exchange)]), is thus amortized among all samples.
  Constant-time sorting networks have previously been adopted in a cryptographic context to uniformly sample permutations or fixed weight vectors [BCL+, BCLV]. Its proposed use to sample from a non-uniform distribution (specifically, the Gaussian distribution, though the same idea clearly generalizes to any distribution) appears to be new.

- For memory-constrained platforms, where only one or a few samples can be generated at a time, one can adapt sequential search to always scan the whole table, keeping track of the index z in a constant-time fashion. Interestingly, this approach may even be somewhat faster than the sorting approach when the CDT is very small (see Table 4).

Amortized sorting approach. Assume that BatcherMergeExchange($\langle sequence\rangle$, key:$\langle key\rangle$, data:$\langle data\rangle$) is a constant-time implementation of the Batcher merge-exchange sorting algorithm for $\langle sequence\rangle$, using the specified $\langle key\rangle$ field of each of its entries for ordering, and carrying the corresponding $\langle data\rangle$ field(s) as associated data. Algorithm 12 generates n Gaussian samples, a chunk of $c \mid n$ samples at a time, in a constant-time fashion.

The advantages of this approach are manifold. This method can be easily written in constant-time, amortizing Batcher's merge-exchange over many samples or resorting to simple sequential search. This flexibility enables its implementation in a wide range of platforms, from desktop/server computers to embedded devices. Moreover, it supports efficient portable implementations without the need of floating-point arithmetic. Methods relying on floating-point arithmetic are more complex and, more importantly, cannot be directly implemented on the many devices that do not include a floating-point unit (FPU). The method is also flexible with regard to the target security level (tailored tables can be readily precomputed and conditionally compiled), since the sampling precision can be easily adjusted.

Implementation details. For our `qTESLA` implementations the sampling precision is set to $\beta > \lambda/2$. Specifically, for Level-`I` and Level-`III` parameter sets we use $\beta = 64$ and 128, respectively, for platforms with computer wordsize $w \in \{32, 64\}$. Likewise, we fix the chunk size to $c = 512$.
For the pseudorandom bit generation required by Algorithm 12, we use cSHAKE as XOF using a seed seed produced by PRF_1 (see line 3 of Algorithm 3) as input string, and a nonce S (written as counter in Algorithm 3) as domain separator.

Table 4 shows the CDT-based methods, sampling precision, and observed speedups when generating chunks of $c = 512$ Gaussian samples at a time, compared to the original `qTESLA` implementation submitted to NIST on November 2017 [Nat17], which was based on the Bernoulli-based rejection sampling from [BLN+16] which in turn was based on [DDLL13a].

Table 4: CDT speedups compared to Bernoulli-based rejection sampling.

`qTESLA-I`	`qTESLA-III-speed`	`qTESLA-III-size`	`qTESLA-p-I`	`qTESLA-p-III`
Batcher	Batcher	Batcher	sequential	Batcher
64 bits	128 bits	128 bits	64 bits	128 bits
26%	42%	52%	82%	47%

5.3 Reference implementation

Our reference implementations, written exclusively in portable C, distinguish between the two variants, *heuristic* and *provably-secure*, in order to maximize simplicity and efficiency in the former case by exploiting the fact that the number of R-LWE samples k for the heuristic parameters is restricted to 1 [4]. Moreover, our implementations exploit the simplicity and scalability of `qTESLA` to provide a common codebase for the different security levels, with only a few minor differences in some packing functions and system constants that are instantiated at compilation time.

All our implementations avoid the use of secret address accesses and secret branches and, hence, are protected against timing and cache side-channel attacks. Whenever appropriate we write "constant-time" code that is branch-free using masking and logical operations. This is the case of the function H, checkE, checkS, polynomial multiplication using the NTT, sparse multiplication and all the polynomial operations requiring modular reductions

[4]We remark that, since the provably-secure implementation uses a generalization of the scheme with $k \geq 1$, it is straightforward to extend this implementation to support `heuristic` `qTESLA` parameters.

Algorithm 12 Constant-time CDT-based Gaussian sampling, GaussSampler

Input: seed seed $\in \{0,1\}^{\kappa}$ and nonce $S \in \mathbb{Z}_{>0}$.
Output: a sequence z of n Gaussian samples.
Global: dimension n, cdt_v: the t-entry right-hand-sided, β-bit precision CDT; c: chunk size, s.t. $c \mid n$; and computer wordsize w.
Local: *samp*: a list of $c + t$ triples of form (k, s, g). ▷ Denote its j-th entry fields as $samp[j].k$, $samp[j].s$, and $samp[j].g$, respectively.

```
1: for $0 \leqslant i < n$ do
 ▷ Prepare a sequence of $c$ uniformly random sorting keys of $\beta$-bit precision and keep
   track of their original sampling order, with an initially null Gaussian index. Invoke
   cSHAKE(seed, $\lceil \beta/8 \rceil$, $S$) for generating the required pseudorandom values:
2:     for $0 \leqslant j < c$ do
3:         $samp[j].k \leftarrow_{\$} \mathbb{Z}/2^{\beta}\mathbb{Z}$
4:         $samp[j].s \leftarrow j$
5:         $samp[j].g \leftarrow 0$   // placeholder
6:     end for
 ▷ Append the $t$ entries of the CDT and keep track of the corresponding sequence of the
   Gaussian indices:
7:     for $0 \leqslant j < t$ do
8:         $samp[c + j].k \leftarrow$ cdt_v$[j]$
9:         $samp[c + j].s \leftarrow \infty$   // search sentinel
10:        $samp[c + j].g \leftarrow j$
11:    end for
 ▷ Sort samp in constant-time according to the $k$ field (the uniformly random samples):
12:    BatcherMergeExchange($samp$, key: $k$, data: $s, g$)
 ▷ Set each entry's Gaussian index, including its sign:
13:    $p\_inx \leftarrow 0$
14:    for $0 \leqslant j < c + t$ do
15:        $inx \leftarrow samp[j].g$
16:        $p\_inx \leftarrow p\_inx \oplus (inx \oplus p\_inx)$ & $((p\_inx - inx) \gg (w - 1))$
17:        $sign \leftarrow_{\$} \mathbb{Z}/2\mathbb{Z}$   // sample the sign
18:        $samp[j].g \leftarrow (sign$ & $-p\_inx) \oplus (\sim sign$ & $p\_inx)$
19:    end for
 ▷ Sort samp in constant-time according to the $s$ field (the sampling order):
20:    BatcherMergeExchange($samp$, key: $s$, data: $g$)   // no need to involve $k$ anymore
 ▷ Discard the trailing entries of samp (corresponding to the CDT):
21:    for $0 \leqslant j < c$ do
22:        $z[i + j] \leftarrow samp[j].g$
23:    end for
24:    $i \leftarrow i + c$
25: end for
26: return $z$
```

or corrections. All the functions that perform some form of rejection sampling, such as the security and correctness tests at signing, GenA, $\mathsf{ySampler}$ and Enc potentially leak the timing of the failure to some internal test, but this information is independent of the secret data. Table lookups performed in our implementation of the Gaussian sampler are done with linear passes over the full table and extracting entries via masking with logical operations.

For the polynomial multiplication we use iterative algorithms for the forward and inverse NTTs, as described in §5.1, using a signed 32-bit datatype for the inputs and outputs. Intermediate results after additions and subtractions are let to grow throughout the execution, and are only reduced or corrected when there is a chance of exceeding 32 bits of length, after a multiplication, or when a result needs to be prepared for final packing (e.g., when outputting secret and public keys). In the NTT and pointwise multiplication the results of multiplications are reduced via Montgomery reductions. To minimize the cost of converting to/from Montgomery representation we use the following approach. First, twiddle factors are scaled *offline* by multiplying with R, where R is the Montgomery constant $2^{32} \bmod q$. Similarly, the coefficients of the outputs a_i from GenA are scaled to remainders $r' = rn^{-1}R \pmod q$ by multiplying with the constant $R^2 \cdot n^{-1}$. This enables an efficient use of Montgomery reductions during the NTT-based polynomial multiplication $\mathsf{NTT}^{-1}(\tilde{a} \circ \mathsf{NTT}(b))$, where $\tilde{a} = \mathsf{NTT}(a)$ is the output in NTT domain of GenA. Multiplications with the twiddle factors during the computation of $\mathsf{NTT}(b)$ naturally cancel out the Montgomery constant. The same happens during the pointwise multiplication with $\tilde{a}$, and finally during the inverse NTT, which naturally outputs values in standard representation without the need of explicit conversions.

5.4 AVX2 optimizations

We have optimized three functions with hand-written assembly implementations exploiting AVX2 vector instructions, namely, polynomial multiplication, sparse multiplication and the XOF expansion for sampling y.

Our polynomial multiplication follows the recent approach by Seiler [Sei18], and the realization of the method has some similarities with the implementation from [DKL+18]. That is, our implementation processes 32 coefficients loaded in 8 AVX2 registers simultaneously, in such a way that butterfly computations are carried out through multiple NTT levels without the need of storing and loading intermediate results, whenever possible. Let us illustrate the procedure we apply for a polynomial a of dimension $n = 512$ written as the vector of coefficients $(a_0, a_1, \ldots, a_{511})$. We split the coefficients in 8 subsets a'_i equally distributed, namely, $a'_0 = (a_0, \ldots, a_{63}), a'_1 = (a_{64}, \ldots, a_{127})$, and so on. We start by loading the first 4 coefficients of each subset a'_i, filling out 8 AVX2 registers in total, and then performing 3 levels of butterfly computations between the corresponding pairs of subsets according to the Cooley-Tukey algorithm. We repeat this procedure 16 times using the subsequent 4 coefficients from each subset a'_i each time. Note that the 3 levels can be completed at once without the need of storing and loading intermediate results. A similar procedure applies to level 4. However, in this case we instead split the coefficients in 16 subsets a'_i such that $a'_0 = (a_0, \ldots, a_{31}), a'_1 = (a_{32}, \ldots, a_{63})$, and so on. We first compute over the first 8 subsets, and then over the other 8. In each case, the butterfly computation is iterated 8 times to cover all the coefficients (again, 4 coefficients are taken at a time from each of the 8 subsets). After level 4, the coefficients are split again in the same 16 subsets a'_i. Conveniently, remaining butterflies need to only be computed between coefficients that belong to the *same* subset. Hence, the NTT computation can be completed by running 16 iterations of butterfly computations, where each iteration computes levels 5–9 at once for each subset a'_i. Therefore, these remaining NTT levels can be computed without additional stores and loads of intermediate results.

One difference with [Sei18, DKL+18] is that our NTT coefficients are represented as 32-bit *signed* integers, which motivates a speedup in the butterfly computation by avoiding the extra additions that are required to make the result of subtractions positive when using an unsigned representation. Moreover, we implement a *full* polynomial multiplication $\mathsf{NTT}^{-1}(\tilde{a} \circ \mathsf{NTT}(b))$ that integrates the pointwise multiplication and the forward and inverse NTTs. This allows us to further optimize the implementation by eliminating multiple load/store operations and some data processing to pack coefficients in the AVX2 registers.
With our approach we reduce the cost of the reference polynomial multiplication from $25,300$ to only $5,800$ cycles for dimension $n = 512$ on an Intel Skylake processor using gcc for compilation. For $n = 1024$, we reduce the cost from $58,200$ to $12,700$ cycles.

Sampling of y is sped up by using the AVX2 implementation of SHAKE by Bertoni *et al.* [BDH+], which allows us to sample up to 4 coefficients in parallel.

We note that it is possible to modify GenA to favor a vectorized computation of the XOF expansion inside this function. However, we avoid this optimization because we prioritize performance on platforms with no vector instruction support.

6 Performance and comparison

Performance. We evaluated the performance of our implementations on a x64 machine powered by a 3.4GHz Intel Core i7-6700 (Skylake) processor running Ubuntu 16.04.3 LTS. As is standard practice, TurboBoost was disabled during the tests. For compilation we used gcc version 7.2.0 with the command `gcc -O3 -march=native -fomit-frame-pointer`. The results for the reference and AVX2-optimized implementations are summarized in Tables 5 and 6, respectively.

Table 5: Performance (in thousands of cycles) of the reference implementations of `qTESLA` on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Results for the median and average (in parenthesis) are rounded to the nearest 10^2 cycles. Signing is performed on a message of 59 bytes.

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	1,123.3 (1,142.3)	377.5 (507.5)	82.6 (83.2)	460.1 (590.7)
`qTESLA-III-speed`	2,904.9 (3,218.4)	536.1 (705.2)	171.0 (171.3)	707.1 (876.5)
`qTESLA-III-size`	1,932.1 (2,010.1)	975.5 (1,363.2)	176.5 (176.8)	1,152.0 (1,540.0)
`qTESLA-p-I`	4,854.5 (5,051.7)	1,208.9 (1,552.5)	499.7 (500.2)	1,708.6 (2,052.7)
`qTESLA-p-III`	26,025.8 (26,507.3)	5,033.1 (6,131.2)	2,519.6 (2,519.6)	7,552.7 (8,650.8)

Our results showcase the high performance of `heuristic qTESLA` with a simple and compact implementation written entirely in portable C: the combined (median) time of signing and verification on the Skylake platform is of approximately 135.3, 208.0 and 338.8 microseconds for `qTESLA-I`, `qTESLA-III-speed` and `qTESLA-III-size`, respectively. Likewise, `provably-secure qTESLA` computes the same operations in approximately 0.50 and 2.22 milliseconds with `qTESLA-p-I` and `qTESLA-p-III`, respectively. This demonstrates

Table 6: Performance (in thousands of cycles) of the AVX2 implementations of `qTESLA` on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Results for the median and average (in parenthesis) are rounded to the nearest 10^2 cycles. Signing is performed on a message of 59 bytes.

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	$1,109.7$ ($1,140.1$)	248.4 (321.8)	62.8 (63.4)	311.2 (385.2)
`qTESLA-III-speed`	$2,862.2$ ($3,156.5$)	328.8 (414.5)	126.2 (126.9)	455.0 (541.4)
`qTESLA-III-size`	$1,883.3$ ($1,964.4$)	557.2 (751.9)	130.1 (130.5)	687.3 (882.4)

that the speed of `provably-secure qTESLA`, although slower, can still be considered practical for most applications.

The AVX2 optimizations improve the performance by a factor between 1.5–1.7x, approximately. The speedup is mainly due to the AVX2 implementation of the polynomial multiplication, which is responsible for $\sim 70\%$ of the total speedup. The combined (median) time of signing and verification on the Skylake platform is of about 91.5, 133.8 and 202.1 microseconds for `qTESLA-I`, `qTESLA-III-speed` and `qTESLA-III-size`, respectively. Similar results were observed on an Intel Haswell processor; see Appendix B.

Comparison. Table 7 compares `qTESLA` to some representative state-of-the-art signature schemes in terms of bit security, signature and key sizes, and performance of reference and AVX2-optimized implementations (if available). If both median and average of cycle counts are provided in the literature, we report the average for signing and the median for verify. To have a fair comparison, we state the bit security of `qTESLA`, pqNTRUSign, Falcon and Dilithium assuming the same BKZ cost model of $0.265\beta + 16.4 + \log_2(8d)$ with β being the BKZ blocksize and d being the lattice dimension (for some schemes that use other cost models we write in brackets the bit security stated in the corresponding papers). As can be seen, with the exception of FALCON-512, `heuristic qTESLA` achieves the best performance among schemes instantiated against state-of-the-art classical *and* quantum attacks [5]. This is accomplished while featuring competitive signature sizes.
It is important to note that, although FALCON-512 exhibits the fastest reference implementation and has the smallest (pk + sig) size among all the post-quantum signature schemes shown in the table, the Falcon scheme has some serious shortcomings due to its high complexity. In particular, this scheme relies on very complex Fourier sampling methods and requires floating-point arithmetic, which is not supported by many devices. All this makes the scheme significantly hard to implement in general, and hard to protect against side-channel and fault attacks in particular. In fact, the results reported in Table 7 correspond to a reference implementation of Falcon that is unprotected against timing and cache attacks. A fully protected implementation is expected to be much more costly.
Schemes based on other underlying problems, such as SPHINCS$^+$ and MQDSS, offer compact public keys at the expense of having very long signatures. In contrast, `qTESLA` has smaller signature sizes, and is significantly faster for signing and verifying. For example, signature generation with `qTESLA-III-speed` is about 21 times faster compared to MGDSS-31-64, when using AVX2 optimizations.

[5]The original instantiations of BLISS and BLISS-B are realized taking into account classical attacks only [DDLL13b, Duc14]. Moreover, known implementations of this scheme are not protected against timing and cache attacks [EFGT17].

In summary, qTESLA offers a good balance between performance and signature/key sizes, accompanied by a simple and compact design that facilitates secure implementations.

References

[ABB$^+$16] Sedat Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. An efficient lattice-based signature scheme with provably secure instantiation. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 2016*, volume 9646 of *LNCS*, pages 44–60. Springer, 2016.

[ABB$^+$17] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, Özgür Dagdelen, Edward Eaton, Gus Gutoski, Juliane Krämer, and Filip Pawlega. Revisiting TESLA in the quantum random oracle model. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 143–162. Springer, Heidelberg, 2017.

[ACD$^+$18] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the LWE, NTRU schemes! In *SCN 2018*, LNCS. Springer, 2018.

[ACF$^+$15] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic algorithms for LWE problems. *ACM Comm. Computer Algebra*, 49(2):62, 2015.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16*, pages 327–343. USENIX Association, 2016.

[AFG13] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-svp. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013*, volume 8565 of *LNCS*, pages 293–310. Springer, 2013.

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part I*, volume 6755 of *LNCS*, pages 403–415. Springer, Heidelberg, July 2011.

[AGVW17] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving uSVP and applications to LWE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 297–322. Springer, Heidelberg, December 2017.

[Alb17] Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 103–129. Springer, Heidelberg, April / May 2017.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[Bat68] K. E. Batcher. Sorting networks and their application. In *AFIPS Spring Joint Computer Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314, Atlantic City (NJ), 1968. Thomson Book Company, Washington D.C.

Table 7: Overview of different post-quantum signature schemes

	Scheme	Security [bit]	const. time	Sizes [B]			Cycle counts [k-cycles]				CPU
							Reference		AVX2		
				pk	sk	sig.	Sign	Verify	Sign	Verify	
Selected ideal lattice-based signatures	BLISS-BI [DDLL13b, Duc14]	128	✗	896	256	717	≈358.0	102.0	-	-	U
	pqNTRUSign (Uniform) [CHZ17, Saf]	183^b	✗	2, 048	2, 604	2, 048	202, 185.0	2, 533.0	-	-	U
	FALCON-512^a [FHK+17, Saf]	158^b	✗	897	4, 097	617	8, 360.0	640.0	-	-	S
	Dilithium-medium [DKL+18]	122^b	✓	1, 184	2, 800	2, 044	1, 510.6	273.6	448.9	118.8	S
	Dilithium-recommended [DKL+18]	160^b	✓	1, 472	3, 504	2, 701	2, 239.5	390.0	627.1	174.9	S
	`qTESLA-p-I` [a] (this paper)	95^b	✓	14 880	5 184	2 848	1, 552.5	499.7	-	-	S
	`qTESLA-p-III` [a] (this paper)	160^b	✓	39 712	12 352	6 176	6, 131.2	2, 519.6	-	-	S
	`qTESLA-I` (this paper)	95^b	✓	1, 504	1, 344	1, 376	507.5	82.6	321.8	62.8	S
	`qTESLA-III-size` (this paper)	160^b	✓	2, 976	2, 112	2, 720	1, 363.2	176.5	751.9	130.1	S
	`qTESLA-III-speed` (this paper)	160^b	✓	3, 104	2, 368	2, 848	705.2	171.0	414.5	126.2	S
Selected PQ signatures	SPHINCS+-128f[a] (Haraka) [BDE+17]	128^c	✓	32	64	16,976	56, 113.0	2, 417.0	-	-	H
	MQDSS-31-64 [CHR+, CHR+16]	128^c	✓	64	24	34, 032	84, 615.0	63, 210.0	8, 709.0	6, 183.0	H

[a] Parameters are chosen according to given security reduction in the ROM/QROM.
[b] Bit security analyzed against classical and quantum adversaries with BKZ cost model $0.265\beta + 16.4 + \log_2(8d)$ [ACD+18].
[c] Bit security analyzed against classical and quantum adversaries.
U: Unknown 3.4GHz Intel Core for BLISS, and unknown x64 CPU [Saf] for pqNTRUSign.
S: 2.6GHz Intel Core i7-6600U (Skylake) for Dilithium, and 3.4GHz Intel Core i7-6700 (Skylake) for **qTESLA** and FALCON-512.
H: 3.5GHz Intel Core i7-4770K (Haswell).

[BCL$^+$] D. Bernstein, T. Chou, T. Lange, I. v. Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang. Classic McEliece: conservative code-based cryptography. NIST post-quantum standardization submission. https://classic.mceliece.org.

[BCLV] D. Bernstein, C. Chuengsatiansup, T. Lange, and C. v. Vredendaal. NTRU Prime. NIST Post-Quantum Cryptography Standardization [Nat17]. https://ntruprime.cr.yp.to. Accessed on 2019-01-07.

[BDE$^+$17] Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, and Peter Schwabe. SPHINCS$^+$. NIST Post-Quantum Cryptography Standardization [Nat17], 2017. https://sphincs.org. Accessed on 2019-01-07.

[BDF$^+$11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 41–69. Springer, Heidelberg, December 2011.

[BDH$^+$] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, , Gilles Van Assche, and Ronny Van Keer. The eXtended Keccak Code Package (XKCP). https://github.com/XKCP/XKCP.

[BG14a] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *CT-RSA 2014*, volume 8366 of *LNCS*, pages 28–47. Springer, Heidelberg, February 2014.

[BG14b] Shi Bai and Steven D. Galbraith. Lattice decoding attacks on binary LWE. In Willy Susilo and Yi Mu, editors, *ACISP 14*, volume 8544 of *LNCS*, pages 322–337. Springer, Heidelberg, July 2014.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, Heidelberg, August 2016.

[BLN$^+$16] Paulo S. L. M. Barreto, Patrick Longa, Michael Naehrig, Jefferson E. Ricardini, and Gustavo Zanon. Sharper ring-LWE signatures. Cryptology ePrint Archive, Report 2016/1026, 2016. http://eprint.iacr.org/2016/1026.

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR TCHES*, 2018(3):21–43, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7267.

[CDPR16] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9666 of *LNCS*, pages 559–585. Springer, 2016.

[CGS14] Peter Campbell, Michael Groves, and Dan Shepherd. SOLILOQUY: A cautionary tale. ETSI 2nd Quantum-Safe Crypto Workshop, 2014. http://docbox.etsi.org/Workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf.

[CHR$^+$] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. MQDSS. NIST Post-Quantum Cryptography Standardization [Nat17]. http://mqdss.org/. Accessed on 2019-01-07.

[CHR$^+$16] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass MQ-based identification to MQ-based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 135–165. Springer, Heidelberg, December 2016.

[CHZ17] Cong Chen, Jeffrey Hoffstein, and William Whyteand Zhenfei Zhang. pqNTRUSign–A modular lattice signature scheme. NIST Post-Quantum Cryptography Standardization [Nat17], 2017. https://www.onboardsecurity.com/nist-post-quantum-crypto-submission . Accessed: 2018-07-23.

[CPBS10] H.M. Cantero, S. Peter, Bushing, and Segher. Console hacking 2010 – PS3 epic fail. 27th Chaos Communication Congress, 2010. https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf.

[DBG$^+$15] Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe. High-speed signatures from standard lattices. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology – LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 84–103. Springer, 2015.

[DDLL13a] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal gaussians. In *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56, Santa Barbara (CA), 2013. Springer.

[DDLL13b] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 40–56. Springer, Heidelberg, August 2013.

[DDLL13c] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. Cryptology ePrint Archive, Report 2013/383, 2013. https://eprint.iacr.org/2013/383.

[DG14] Nagarjun C. Dwarakanath and Steven D. Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159–180, Jun 2014.

[DKL$^+$18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR TCHES*, 2018(1):238–268, 2018. https://tches.iacr.org/index.php/TCHES/article/view/839.

[DN12] Léo Ducas and Phong Q. Nguyen. Learning a zonotope and more: Cryptanalysis of NTRUSign countermeasures. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 433–450. Springer, Heidelberg, December 2012.

[Duc14] Léo Ducas. Accelerating BLISS: the geometry of ternary polynomials. Cryptology ePrint Archive, Report 2014/874, 2014. http://eprint.iacr.org/2014/874.

[Dwo15] Morris J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. *Federal Inf. Process. Stds. (NIST FIPS) – 202*, 2015. Available at https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1857–1874. ACM Press, October / November 2017.

[ELOS15] Yara Elias, Kristin E. Lauter, Ekin Ozman, and Katherine E. Stange. Provably weak instances of ring-LWE. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference*, volume 9215 of *LNCS*, pages 63–92. Springer, 2015.

[FHK+17] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier Lattice-Based Compact Signatures over NTRU. NIST Post-Quantum Cryptography Standardization [Nat17], 2017. https://falcon-sign.info/. Accessed: 2018-07-23.

[GGH13] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17. Springer, Heidelberg, May 2013.

[GJS15] Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-bkw: Solving LWE using lattice codes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 23–42. Springer, 2015.

[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, Heidelberg, September 2012.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[HHP+03] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUSign: Digital signatures using the NTRU lattice. In Marc Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 122–140. Springer, Heidelberg, April 2003.

[HKR+18] J. Howe, A. Khalid, C. Rafferty, F. Regazzoni, and M. O'Neill. On practical discrete gaussian samplers for lattice-based cryptography. *IEEE Transactions on Computers*, 67(3):322–334, March 2018.

[Kel16] John Kelsey. SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. *NIST Special Publication*, 800:185, 2016. Available at http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf.

[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, Heidelberg, April / May 2018.

[Knu97] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 2ns edition, 1997.

[Laa16] Thijs Laarhoven. *Search problems in cryptography.* PhD thesis, Eindhoven University of Technology, 2016.

[LMP13] Thijs Laarhoven, Michele Mosca, and Joop Pol. Solving the Shortest Vector Problem in Lattices Faster Using Quantum Search. In Philippe Gaborit, editor, *Post-Quantum Cryptography*, volume 7932 of *Lecture Notes in Computer Science*, pages 83–101. Springer Berlin Heidelberg, 2013.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, Heidelberg, February 2011.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[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, 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, Heidelberg, April 2012.

[Nat16] National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. Accessed: 2018-07-23.

[Nat17] National Institute of Standards and Technology (NIST). Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography, 2017. Accessed: 2018-07-23.

[NR06] Phong Q. Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 271–288. Springer, Heidelberg, May / June 2006.

[Pei10] C. Peikert. An efficient and parallel Gaussian sampler for lattices. In *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97, Santa Barbara (CA), 2010. Springer.

[PG14] T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 2014.

[PSDS17] Thomas Plantard, Arnaud Sipasseuth, Cédric Dumondelle, and Willy Susilo. DRS: Diagonal dominant Reduction for lattice-based Signature. NIST Post-Quantum Cryptography Standardization [Nat17], 2017. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions . Accessed: 2018-07-23.

[PSS+17] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014, 2017. http://eprint.iacr.org/2017/1014.

[RVM+14] S. Sinha Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact Ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014.

[Saf] SafeCrypto. NIST Software Analysis–Signatures. https://www.safecrypto.eu/pqclounge/software-analysis-signatures/. Accessed: 2018-07-05.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.

[YD18] Yang Yu and Léo Ducas. Learning strikes again: The case of the DRS signature scheme. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 525–543. Springer, Heidelberg, December 2018.

[Zha12] Mark Zhandry. Secure identity-based encryption in the quantum random oracle model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 758–775. Springer, Heidelberg, August 2012.

[ZSS18] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. FACCT: FAst, Compact, and Constant-Time Discrete Gaussian Sampler over Integers. Cryptology ePrint Archive, Report 2018/1234, 2018. https://eprint.iacr.org/2018/1234.

A Correctness of qTESLA

To prove the correctness of qTESLA, we have to show that for every signature (z, c') of a message m generated by Algorithm 4 it holds that (i) $z \in \mathcal{R}_{q,[B-L_S]}$ and (ii) the output of the hash-based function H at signing (line 9 of Algorithm 4) is the same as the analogous output at verification (line 6 of Algorithm 5).

Requirement (i) is ensured by the security check during signing (line 12 of Algorithm 4). To ensure (ii), we need to prove that, for genuine signatures and for all $i = 1, \ldots, k$ it holds that $[a_i y \bmod^{\pm} q]_M = [a_i z - t_i c \bmod^{\pm} q]_M = [a_i(y + sc) - (a_i s + e_i)c \bmod^{\pm} q]_M = [a_i y + a_i sc - a_i sc - e_i c \bmod^{\pm} q]_M = [a_i y - e_i c \bmod^{\pm} q]_M$. From the definition of $[\cdot]_M$, this means proving that $(a_i y \bmod^{\pm} q - [a_i y \bmod^{\pm} q]_L)/2^d = (a_i y - e_i c \bmod^{\pm} q - [a_i y - e_i c \bmod^{\pm} q]_L)/2^d$, or simply $[a_i y \bmod^{\pm} q]_L = e_i c + [a_i y - e_i c \bmod^{\pm} q]_L$.

The above equality must hold component-wise, so let us prove the corresponding property for individual integers.

Assume that for integers α and ε it holds that $|[\alpha - \varepsilon \bmod^{\pm} q]_L| < 2^{d-1} - L_E$, $|\varepsilon| \leq L_E < \lfloor q/2 \rfloor$, $|\alpha - \varepsilon \bmod^{\pm} q| < \lfloor q/2 \rfloor - L_E$, and $-\lfloor q/2 \rfloor < \alpha \leq \lfloor q/2 \rfloor$ (i.e., $\alpha \bmod^{\pm} q = \alpha$). Then, we need to prove that

$$[\alpha]_L = \varepsilon + [\alpha - \varepsilon \bmod^{\pm} q]_L. \tag{4}$$

Proof. To prove equation (4), start by noticing that $|\varepsilon| \leq L_E < 2^{d-1}$ implies $[\varepsilon]_L = \varepsilon$. Thus, from $-2^{d-1} + L_E < [\alpha - \varepsilon \bmod^{\pm} q]_L < 2^{d-1} - L_E$ and $-L_E \leq [\varepsilon]_L \leq L_E$ it follows that

$$-2^{d-1} = -2^{d-1} + L_E - L_E < [\varepsilon]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L < 2^{d-1} - L_E + L_E = 2^{d-1},$$

and therefore

$$[[\varepsilon]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L]_L = [\varepsilon]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L = \varepsilon + [\alpha - \varepsilon \bmod^{\pm} q]_L. \quad (5)$$

Next we prove that

$$[[\varepsilon]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L]_L = [\alpha]_L. \quad (6)$$

We note that since $|\varepsilon| \leq L_E < \lfloor q/2 \rfloor$ it holds that $[\varepsilon]_L = [\varepsilon \bmod^{\pm} q]_L$. It holds further that

$$\begin{aligned} & [[\varepsilon \bmod^{\pm} q]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L]_L & (7) \\ = \; & ((\varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d + (\alpha - \varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d) \bmod^{\pm} 2^d & (8) \\ & \text{by the definition of } [\cdot]_L & \\ = \; & (\varepsilon \bmod^{\pm} q + (\alpha - \varepsilon \bmod^{\pm} q)) \bmod^{\pm} 2^d. & (9) \end{aligned}$$

Since $|\varepsilon| \leq L_E$ and $|\alpha - \varepsilon \bmod^{\pm} q| < \lfloor q/2 \rfloor - L_E$, it holds that $|\alpha - \varepsilon| + |\varepsilon| < (\lfloor q/2 \rfloor - L_E) + L_E = \lfloor q/2 \rfloor$. Hence, equation (9) is the same as

$$\begin{aligned} = \; & (\varepsilon + \alpha - \varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d = (\alpha \bmod^{\pm} q) \bmod^{\pm} 2^d = \alpha \bmod^{\pm} 2^d \\ = \; & [\alpha]_L. \end{aligned}$$

Combining equations (5) and (6) we deduce that $[\alpha]_L = \varepsilon + [\alpha - \varepsilon \bmod^{\pm} q]_L$, which is the equation we needed to prove. □

Now define $\alpha := (a_i y)_j$ and $\varepsilon := (e_i c)_j$ with $i \in \{1, \ldots, k\}$ and $j \in \{0, \ldots, n-1\}$. From line 18 of Algorithm 4, we know that for $i = 1, \ldots, k$, $\| [a_i y - e_i c]_L \|_\infty < 2^{d-1} - L_E$ and $\|a_i y - e_i c\|_\infty < \lfloor q/2 \rfloor - L_E$ for a valid signature, and that Algorithm 3 (line 13) guarantees $\|e_i c\|_\infty \leq L_E$. Likewise, by definition it holds that $L_E < \lfloor q/2 \rfloor$; see Section 3.4. Finally, $v_i = a_i y$ is reduced $\bmod^{\pm} q$ in line 7 of Algorithm 4 and, hence, v_i is in the centered range $-\lfloor q/2 \rfloor < a_i y \leq \lfloor q/2 \rfloor$.

In conclusion, we get the desired condition for ring elements, $[a_i y]_L = e_i c + [a_i y - e_i c]_L$, which in turn means $[a_i z - t_i c]_M = [a_i y]_M$ for $i = 1, \ldots, k$ as argued above.

B Performance of qTESLA on Haswell

Our benchmarking results on a 3.4GHz Intel Core i7-4770 (Haswell) processor are summarized in Table 8 for the reference implementation, and in Table 9 for the AVX2 implementation. As is standard practice, TurboBoost was disabled during the tests. For compilation we used gcc version 7.2.0 with the command `gcc -O3 -march=native -fomit-frame-pointer`.

Table 8: Performance (in thousands of cycles) of the reference implementation of `qTESLA` on a 3.4GHz Intel Core i7-4770 (Haswell) processor. Results for the median and average (in parenthesis) are rounded to the nearest 10^2 cycles. Signing is performed on a message of 59 bytes.

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	1,148.3 (1,170.4)	395.3 (527.5)	85.8 (86.4)	481.1 (613.9)
`qTESLA-III-speed`	2,921.2 (3,325.9)	564.7 (736.0)	177.6 (181.0)	745.3 (923.7)
`qTESLA-III-size`	1,927.2 (2,044.5)	1,036.0 (1,455.8)	182.1 (186.0)	1,218.1 (1,641.8)
`qTESLA-p-I`	4,959.7 (5,182.0)	1,273.0 (1,591.5)	518.0 (518.6)	1,791.0 (2,110.1)
`qTESLA-p-III`	26,342.3 (26,925.1)	5,275.9 (6,425.5)	2,636.7 (2,640.9)	7,912.6 (9,066.4)

Table 9: Performance (in thousands of cycles) of the AVX2 implementation of `qTESLA` on a 3.4GHz Intel Core i7-4770 (Haswell) processor. Results for the median and average (in parenthesis) are rounded to the nearest 10^2 cycles. Signing is performed on a message of 59 bytes.

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	1,125.2 (1,150.7)	257.3 (332.4)	65.7 (66.6)	323.0 (399.0)
`qTESLA-III-speed`	2,872.7 (3,192.2)	341.5 (436.2)	134.6 (136.7)	476.1 (572.9)
`qTESLA-III-size`	1,879.2 (1,959.2)	589.4 (787.0)	140.5 (140.7)	729.9 (927.7)