The Lattice-Based Digital Signature Scheme qTESLA

Erdem Alkim[1], Paulo S. L. M. Barreto[2], Nina Bindel[3], Juliane Krämer[4], Patrick Longa[5], and Jefferson E. Ricardini[6]

[1] *Ondokuz Mayis University and Fraunhofer SIT, erdemalkim@gmail.com*
[2] *University of Washington Tacoma, pbarreto@uw.edu*
[3] *University of Waterloo, nlbindel@uwaterloo.ca*
[4] *Technische Universität Darmstadt, jkraemer@cdc.informatik.tu-darmstadt.de*
[5] *Microsoft Research, plonga@microsoft.com*
[6] *LG Electronics, jefferson1.ricardini@lge.com*

December 13, 2019

Abstract

We present `qTESLA`, a family of post-quantum digital signature schemes that exhibits several attractive features such as simplicity and strong security guarantees against quantum adversaries, and built-in protection against certain side-channel and fault attacks. `qTESLA`—selected for round 2 of NIST's post-quantum cryptography standardization project—consolidates a series of recent schemes originating in works by Lyubashevsky, and Bai and Galbraith. We provide full-fledged, constant-time portable C implementations that showcase the code compactness of the proposed scheme, e.g., our code requires only about 300 lines of C code. Finally, we also provide AVX2-optimized assembly implementations that achieve a factor-1.5 speedup.

Keywords: Post-quantum cryptography, lattice-based cryptography, digital signatures, provable security.

Contents

1 Introduction

The potential advent of quantum computers has prompted the cryptographic community to look for *quantum-resistant* alternatives to classical schemes that are 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 (quantum) secure signature scheme. The security of `qTESLA` relies on the so-called decisional ring learning with errors (R-LWE) problem [42]. Parameters are generated according to the provided security reduction from R-LWE, i.e., instantiations of the scheme guarantee a certain security level as long as the corresponding R-LWE instances give a certain hardness[1].

The most relevant features of `qTESLA` are summarized as follows:

Simplicity. `qTESLA` is designed to be 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. `qTESLA`'s simple design makes it straightforward to easily support more than one security level and parameter set with a single and compact portable implementation. For instance, our reference implementation written in portable C and supporting all `qTESLA` parameter sets consists of only ~ 300 lines of code[2].

Security foundation. The security of `qTESLA` is ensured by a security reduction in the quantum random oracle model (QROM) [17], i.e., a quantum adversary is allowed to ask the random oracle in superposition. Moreover, the explicitness of the reduction enables choosing parameters according to the reduction, while its tightness enables smaller parameters and, thus, better performance for provably secure instantiations.

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 accesses and secret branches), and is inherently protected against certain simple yet powerful fault attacks [18, 45]. Moreover, it also comes with a built-in safeguard to protect against Key Substitution (KS) attacks [16, 43] (a.k.a. Duplicate Signature Key Selection (DSKS) attacks) and, thus, improved security in the multi-user setting; see also [33].

[1] It is important to note that the security reduction requires a conjecture to bound a probability explicitly. See §5 for details.

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

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 [11], which is based on the Fiat-Shamir construction of Lyubashevsky [39, 40]. The Bai-Galbraith scheme is constructed over standard lattices and comes with a (non-tight) security reduction from the LWE and the SIS problem in the random oracle model (ROM). Dagdelen, El Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, Sánchez, and Schwabe [23] presented improvements and the first implementation of the Bai-Galbraith scheme. The scheme was subsequently studied under the name TESLA by Alkim, Bindel, Buchmann, Dagdelen, Eaton, Gutoski, Krämer, and Pawlega [8], 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 [1]. 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; 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; 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, qTESLA adopts the next features: following a standard security practice, the public polynomials a_i are freshly generated at each key pair generation; the underlying algebraic support is extended to non-power-of-two cyclotomic rings; and the hash of the public key is included in the signature computation to protect against KS attacks [16], improving security in the multi-user setting.

Another variant of the Bai-Galbraith scheme is the lattice-based signature scheme Dilithium [26, 41] which is constructed over module lattices. While qTESLA and Dilithium share several properties such as a tight security reduction in the QROM [35], Dilithium signatures are *deterministic* by default[3], 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 [18, 45]. It is also important to remark that, arguably, 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 lattice signature schemes, namely, GLP [30] and BLISS [25]. These schemes were inspirational for some of qTESLA's building blocks, such as the encoding function.

[3]Recently, a variant of Dilithium that produces probabilistic signatures was included as a modification for round 2 of the NIST post-quantum project [41]. However, [41] suggests the deterministic version as the default option.

In a separate category we mention other lattice-based signature schemes such as Falcon [48] and pqNTRUSign [53], 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.

Software release. We have released our portable C and AVX2-optimized implementations as open source:

`https://github.com/Microsoft/qTESLA-Library`.

The implementation software 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, and describe the efficient realization of the scheme's basic functions in §4. In §5, we describe the security foundation of `qTESLA` and the proposed parameter sets. Finally, we describe implementation details of our portable C and AVX2-optimized implementations, as well as our experimental results and a comparison with state-of-the-art signature schemes, in §6.

Acknowledgments

We are grateful to the anonymous TCHES and PKC reviewers for their valuable comments on earlier versions of this paper. We thank Vadim Lyubashevsky for pointing out that the heuristic parameters proposed in a previous paper version were lacking security estimates with respect to the short integer solution problem. We also thank Greg Zaverucha for bringing up the vulnerability of some signature schemes, including a previous version of qTESLA, to key duplication attacks, and for several fruitful discussions. We are thankful to Edward Eaton for fruitful discussions about the conjecture used in Theorem 1 and carrying out the supporting experiments. Finally, we thank Fernando Virdia, Martin Albrecht and Shi Bai for fruitful discussions and helpful advice on the hardness estimation of SIS.

The work of EA was partially supported by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity, and was partially carried out during his tenure of the ERCIM 'Alain Bensoussan' Fellowship Programme. NB is supported by the NSERC Discovery Accelerator Supplement grant RGPIN-2016-05146. JK 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 q be an odd prime throughout this work. 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 $\sum_{i=0}^{n-1}(f_i \bmod q)x^i \in \mathcal{R}_q$. Let $\mathbb{H}_{n,h} = \{\sum_{i=0}^{n-1} f_i x^i \in \mathcal{R} \mid f_i \in \{-1, 0, 1\},\ \sum_{i=0}^{n-1} |f_i| = h\}$, and $\mathcal{R}_{[B]} = \{\sum_{i=0}^{n-1} f_i x^i \in \mathcal{R} \mid f_i \in [-B, B]\}$.

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 c' such that $-m/2 < c' \leq m/2$ (resp. $-\lfloor m/2\rfloor \leq c' \leq \lfloor m/2\rfloor$) and $c' = c \bmod m$. We then define the functions $[\cdot]_L : \mathbb{Z} \to \mathbb{Z},\ c \mapsto (c \bmod^{\pm} q) \bmod^{\pm} 2^d$, and $[\cdot]_M : \mathbb{Z} \to \mathbb{Z},\ c \mapsto (c \bmod^{\pm} q - [c]_L)/2^d$. Hence, $c \bmod^{\pm} q = 2^d \cdot [c]_M + [c]_L$ for $c \in \mathbb{Z}$. These definitions are extended to polynomials by applying the operators to each polynomial coefficient, i.e., $[f]_L = \sum_{i=0}^{n-1} [f_i]_L\, x^i$ 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. For an element $c \in \mathbb{Z}$, 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 $a_j = (a_{j,0}, a_{j,1}, \ldots, a_{j,n-1}) \in \mathbb{Z}_q^n$ for any $j \in \{1, \ldots, k\}$. 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 with standard deviation σ is defined to be $\mathcal{D}_\sigma = \rho_\sigma(c)/\rho_\sigma(\mathbb{Z})$ for $c \in \mathbb{Z}$, 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 $x \leftarrow_\sigma \mathbb{Z}$ to denote sampling a value x 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 $\mathcal{D}_\sigma$. Moreover, for a finite set S, we denote sampling s uniformly from S with $s \leftarrow_\$ S$ or $s \leftarrow \mathcal{U}(S)$.

2.2 The number-theoretic transform (NTT)

Polynomial multiplication over a finite field is one of the fundamental operations in lattice-based schemes. Satisfying the condition $q = 1 \bmod 2n$ enables the use of the NTT, leading to an efficient realization of polynomial multiplication. qTESLA specifies the generation of the polynomials $a_1, \ldots, a_k$ directly in the NTT domain for efficiency purposes. Hence, 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., $\omega^n = 1 \bmod q$, 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

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

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

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

It then holds that $\mathsf{NTT}^{-1}(\mathsf{NTT}(a)) = a$ for all polynomials $a \in \mathcal{R}_q$. 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 Hardness assumptions

The security of qTESLA is based on the hardness of the R-LWE problem. In the following definition we use $\mathcal{A}^{\mathcal{O}}$ to denote that $\mathcal{A}$ has access to an oracle $\mathcal{O}$.

Definition 1 ($\text{R-LWE}_{n,k,q,\chi}$)**.** *Let $n, q > 0$ be integers, χ be a distribution over $\mathcal{R}$, and $s \leftarrow \chi$. 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 $a \leftarrow \mathcal{U}(\mathcal{R}_q)$ and $e \leftarrow \chi$.*

Given k tuples $(a_1, t_1), \ldots, (a_k, t_k)$, the decisional R-LWE problem $\text{R-LWE}_{n,k,q,\chi}$ is to distinguish whether $(a_i, t_i) \leftarrow \mathcal{U}(\mathcal{R}_q \times \mathcal{R}_q)$ or $(a_i, t_i) \leftarrow \mathcal{D}_{s,\chi}$ for all i. The R-LWE advantage is defined as

$$\text{Adv}_{n,k,q,\chi}^{\text{R-LWE}}(\mathcal{A}) = \left| \Pr\left[\mathcal{A}^{\mathcal{D}_{\chi,s}(\cdot)} = 1\right] - \Pr\left[\mathcal{A}^{\mathcal{U}(\mathcal{R}_q \times \mathcal{R}_q)(\cdot)} = 1\right] \right|.$$

The above definition corresponds to the *normal form* of R-LWE [42], in which the secret and error polynomials follow the same distribution χ. In qTESLA, χ is instantiated with $\mathcal{D}_\sigma$.

3 The signature scheme qTESLA

In this section, we describe the signature scheme qTESLA, some of 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, E, S, B, d, h, and b_{GenA}; see Table 1 in §3.2 for a detailed description of all the system parameters. The following functions are required for the implementation of the scheme:

- The pseudorandom functions $\mathsf{PRF}_1 : \{0,1\}^\kappa \to \{0,1\}^{\kappa,k+3}$, which takes as input a seed pre-seed that is κ bits long and maps it to $(k+3)$ seeds of κ bits each.
- The collision-resistant hash function $\mathsf{G} : \{0,1\}^* \to \{0,1\}^{320}$, which maps a given input string to a 320-bit string.
- The pseudorandom function $\mathsf{PRF}_2 : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^{320} \to \{0,1\}^\kappa$, 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 320-bit long, and maps them to the κ-bit seed rand.
- The generation function of the 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 polynomial in $\mathcal{R}$ sampled according to the discrete Gaussian distribution $\mathcal{D}_\sigma$.
- The encoding function $\mathsf{Enc} : \{0,1\}^\kappa \to \{0,\ldots,n-1\}^h \times \{-1,1\}^h$ encodes a κ-bit hash value c' as a polynomial $c \in \mathbb{H}_{n,h}$. The polynomial c is represented 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 $\mathsf{ySampler} : \{0,1\}^\kappa \times \mathbb{Z} \to \mathcal{R}_{[B]}$ samples a polynomial $y \in \mathcal{R}_{[B]}$, taking as inputs a κ-bit seed rand and a nonce $\mathsf{counter} \in \mathbb{Z}_{>0}$.
- The hash-based function $\mathsf{H} : \mathcal{R}_q^k \times \{0,1\}^{320} \times \{0,1\}^{320} \to \{0,1\}^\kappa$. This function takes as inputs k polynomials $v_1, \ldots, v_k \in \mathcal{R}_q$ and first computes $[v_1]_M, \ldots, [v_k]_M$. The result is then hashed together with the hash $\mathsf{G}(m)$ for a given message m and the hash $\mathsf{G}(t_1, \ldots, t_k)$ to a string κ bits long.
- The correctness check function checkE, which gets an error polynomial e as input and rejects 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 $\|e_i c\|_\infty \leq E \in \{L_E, 2L_E\}$ for $i = 1, \ldots, k$ during key generation.
- 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 $\|sc\|_\infty \leq S \in \{L_S, 2L_S\}$, 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 at random over $\mathcal{R}_q$ (lines 2–4) by expanding the seed seed_a using PRF_1. Then, a secret polynomial s is sampled with discrete Gaussian distribution $\mathcal{D}_\sigma$. This polynomial must fulfill the requirement check in checkS (lines 5–8). A similar procedure to sample the secret error polynomials $e_1, \ldots, e_k$ follows. In this case, these polynomials must fulfill the correctness check in checkE (lines 10–13). To generate pseudorandom bit strings during the discrete 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 public key pk consists of seed_a and the polynomials $t_i = a_i s + e_i \mod q$ for $i = 1, \ldots, k$, and the secret key sk consists of $s, e_1, \ldots, e_k$, the seeds seed_a and seed_y, and the hash $g = \mathsf{G}(t_1, \ldots, t_k)$. 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 $y \in \mathcal{R}_{[B]}$ 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 digests $\mathsf{G}(m)$ and g 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 $z \notin \mathcal{R}_{[B-S]}$. 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 $\|[w_i]_L\|_\infty < 2^{d-1} - E$ and $\|w_i\|_\infty < \lfloor q/2 \rfloor - E$. If the check fails, the algorithm discards the current pair (z, c') and repeats all the steps beginning with the sampling of y. Otherwise, it returns the signature (z, c') on m.

Verification. The verification algorithm, upon input of a message m, a signature (z, c'), and a public key pk, 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 $w_i = a_i z - b_i c \bmod^{\pm} q$ for $i = 1, \ldots, k$. The hash-based function H computes $[w_1]_M, \ldots, [w_k]_M$ and hashes these together with the digests $\mathsf{G}(m)$ and $\mathsf{G}(t_1, \ldots, t_k)$. If the bit string resulting from the previous computation matches the signature bit string c', and $z \in \mathcal{R}_{[B-S]}$, the signature is accepted; otherwise, it is rejected.

Correctness of qTESLA. To guarantee the correctness of qTESLA it must hold for a signature (z, c') of a message m generated by Algorithm 4 that (i) $z \in \mathcal{R}_{[B-S]}$ and that (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 $i = 1, \ldots, k$, $[a_i z - t_i c]_M = [a_i(y + sc) - (a_i s + e_i)c]_M = [a_i y - e_i c]_M = [a_i y]_M$. A formal correctness proof can be found in Appendix A.

Design features. qTESLA's design comes with several built-in security features. First, the public polynomials $a_1, \ldots, a_k$ are freshly generated at each key generation, using the random seed seed_a. This seed is stored as part of both sk and pk so that the signing and verification operations can regenerate $a_1, \ldots, a_k$. This makes the introduction of backdoors more difficult and reduces drastically the scope of all-for-the-price-of-one attacks [9,13]. 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 required to represent the full polynomials.
To protect against KS attacks [16], we include the hash G of the polynomials $t_1, \ldots, t_k$ (which are part of the public key) in the secret key, in order to use it during the hashing operation to derive c'. This guarantees that any attempt by an attacker of modifying the public key will be detected during verification when checking the value c' (line 6 of Alg. 5).

Also, the seed used to generate the randomness y at signing 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 [19]. Likewise, the random value r guarantees the use of a fresh y at each signing operation, which makes qTESLA's signatures *probabilistic*. Probabilistic signatures are, arguably, more difficult to attack through side-channel analysis. Moreover, the fresh y

prevents some easy-to-implement but powerful fault attacks against deterministic signature schemes [18, 45]; see [18, §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.

Another design feature of `qTESLA` is that discrete Gaussian sampling, arguably the most complex function in many lattice-based signature schemes, is only required during key generation, while signing and verification, the most used functions of digital signature schemes, only use very simple arithmetic operations that are easy to implement. This facilitates the realization of compact and portable implementations that achieve high performance.

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

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

Algorithm 4 qTESLA's signature generation

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

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

Algorithm 5 qTESLA's signature verification

Require: message m, signature (z, c'), and public key $pk = (t_1, \ldots, t_k, \mathsf{seed}_a)$
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: $\quad w_i \leftarrow a_i z - t_i c \bmod^{\pm} q$
5: **end for**
6: **if** $z \notin \mathcal{R}_{[B-S]} \vee c' \neq \mathsf{H}(w_1, \ldots, w_k, \mathsf{G}(m), \mathsf{G}(t_1, \ldots, t_k))$ **then**
7: $\quad$ **return** -1 } Reject signature (z, c') for m.
8: **end if**
9: **return** 0 } Accept signature (z, c') for m.

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	2^ℓ
σ	standard deviation of $\mathcal{D}_\sigma$	-
k	# public polynomials $a_1, \ldots, a_k$	-
q	modulus	$q = 1 \bmod 2n$, $q > 2B$, $q > 2^{d+1}$ $q^{nk} \geq \lvert\Delta\mathbb{S}\rvert \cdot \lvert\Delta\mathbb{L}\rvert \cdot \lvert\Delta\mathbb{H}\rvert$, $q^{nk} \geq 2^{4\lambda+nkd} 4q_s^3(q_s+q_h)^2$
h	# of nonzero entries of output elements of Enc	$2^h \cdot \binom{n}{h} \geq 2^{2\lambda}$
κ	out-/input length of different functions	$\kappa \geq \lambda$
L_E, η_E	bound in checkE	$\lceil \eta_E \cdot h \cdot \sigma \rceil$
L_S, η_S	bound in checkS	$\lceil \eta_S \cdot h \cdot \sigma \rceil$
S, E	rejection parameters	$= L_S, L_E$
M^2	lower bound on the signature acceptance rate	-
B	determines interval randomness during sign	near a power-of-two, $B \geq \frac{\sqrt[n]{M}+2S-1}{2(1-\sqrt[n]{M})}$
d	#rounded bits	$d > \log_2(B), d \geq \log_2\left(\frac{2E+1}{1-M^{\frac{1}{nk}}}\right)$
b_{GenA}	# blocks requested to SHAKE128 for GenA	$b_{\mathsf{GenA}} \in \mathbb{Z}_{>0}$
$\lvert\Delta\mathbb{H}\rvert$	$\Delta\mathbb{H} = \{c - c' \ : \ c, c' \in \mathbb{H}_{n,h}\}$	$\sum_{j=0}^{h} \sum_{i=0}^{h-j} \binom{kn}{2i} 2^{2i} \binom{kn-2i}{j} 2^j$
$\lvert\Delta\mathbb{S}\rvert$	$\Delta\mathbb{S} = \left\{z - z' \ : \ z, z' \in \mathcal{R}_{[B-U]}\right\}$	$(4(B-S)+1)^n$
$\lvert\Delta\mathbb{L}\rvert$	$\Delta\mathbb{L} = \{x - x' : x, x' \in \mathcal{R} \text{ and } [x]_M = [x']_M\}$	$(2^d+1)^{nk}$
sig size	theoretical size of signature [bits]	$\kappa + n(\lceil \log_2(B-S) \rceil + 1)$
pk size	theoretical size of public key [bits]	$kn(\lceil \log_2(q) \rceil) + \kappa$
sk size	theoretical size of secret key [bits]	$n(k+1)(\lceil \log_2(t-1) \rceil + 1) + 2\kappa + 320$ with $t = 89, 110, 133, 78$ or 111

3.2 Parameter description

qTESLA's system parameters and their corresponding bounds are summarized in Table 1. The parameter λ is defined as the security parameter, i.e., the targeted bit security of a given instantiation. In the standard 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., $n = 2^\ell$ for $\ell \in \mathbb{N}$. The parameter $k \in \mathbb{Z}_{>0}$ is the number of ring learning with errors samples used by a given instantiation. 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 $a_1, \ldots, a_k$. The values of b_{GenA} are chosen experimentally such that they maximize performance on the targeted Intel platform; see §6.

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 explained in §5. Both bounds, L_S and L_E (and consequently S and E[4]), impact the rejection probability during 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 two 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 S as follows:

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

where M 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 $\|[w]_L\|_\infty \geq 2^{d-1} - E$ in line 18 of Algorithm 4 is lower bounded by M. This check determines the acceptance probability δ_w during signature generation. The acceptance probability of z, namely δ_z, is related to the value of M. The final acceptance probabilities δ_z, δ_w and δ_{keygen} obtained experimentally, following the procedure above, are summarized in Table 4.

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 $q > 2B$. To choose parameters according to the security reduction, it is first convenient to simplify our security statement. To this end we ensure that $q^{nk} \geq |\Delta\mathbb{S}| \cdot |\Delta\mathbb{L}| \cdot |\Delta\mathbb{H}|$; see Table 1 for the definition of the respective sets. Then, the following equation (see Theorem 1 in §5.1.1) has to hold:

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

Following the NIST's call for proposals [44, §4.A.4], we choose the number of classical queries to the sign oracle to be $q_s = \min\{2^{64}, 2^{\lambda/2}\}$ for all our parameter

[4] In an earlier version of this document we needed to distinguish L_S/L_E and S/E. Although this is not necessary in this version, we keep all four values L_S, S, L_E, E for consistency reasons.

sets. Moreover, we choose the number of queries of a hash function to be $q_h = \min\{2^{128}, 2^{\lambda}\}$.

Key and signature sizes. The theoretical bitlengths of the signatures and public keys are given by $\kappa + n \cdot (\lceil \log_2(B-S) \rceil + 1)$ and $k \cdot n \cdot (\lceil \log_2(q) \rceil) + \kappa$, respectively. To determine the size of the secret keys we first define t as the number of β-bit entries of the discrete Gaussian sampler's CDT tables (see Table 2) which corresponds to the maximum value that can be possibly sampled to generate the coefficients of secret polynomials s. Then, it follows that the theoretical size of the secret key is given by $n(k+1)(\lceil \log_2(t-1) \rceil + 1) + 2\kappa + 320$ bits.

4 Realization of basic functions

In this section, we describe the low-level algorithms for the basic functions required in the signature scheme.

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). For qTESLA we use the XOF function SHAKE [27] in the realization of the functions G and H, and cSHAKE128 [34] in the realization of the functions GenA and Enc. To implement the functions PRF_1, PRF_2, ySampler, and 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 ySampler and GaussSampler). With the exception of GenA and Enc (which always use cSHAKE128), our Level-I parameter set uses (c)SHAKE128 and our Level-III parameter set uses (c)SHAKE256.
For the remainder, we use $\mathsf{XOF}(\mathsf{X}, \mathsf{L}, \mathsf{D})$ to denote a call to a XOF, where X is the input string, L is the output length in bytes, and D is an *optional* domain separator[5].

Generation of seeds. qTESLA requires the generation of the seeds seed_s, seed_{e_1}, ..., 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, ..., e_k, a_1, ..., 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 $\mathbf{a_1}, \ldots, \mathbf{a_k}$. 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 [27] and b_{GenA} is a qTESLA parameter (see §3.2). Then, the algorithm proceeds to do rejection sampling over each $8 \cdot \lceil \log_2(q) \rceil$-bit string of the cSHAKE output modulo

[5]The domain separator D 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 κ 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** $(\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}, \mathsf{seed}_a)$

$2^{\lceil \log_2(q) \rceil}$, 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 first call to cSHAKE128 uses the value $D = 0$ as domain separator. This value is incremented by one at each subsequent call.
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 for their NTT conversion during the polynomial multiplications. This permits an important speedup of the polynomial operations without affecting security.

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: $D \leftarrow 0,\ b' \leftarrow b_{\mathsf{GenA}}$
2: $\langle c_0 \rangle \| \langle c_1 \rangle \| \ldots \| \langle c_T \rangle \leftarrow$ cSHAKE128$(\mathsf{seed}_a, \mathrm{rate}_{\mathsf{XOF}} \cdot b', D)$, with $\langle c_t \rangle \in \{0,1\}^{8b}$
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: $D \leftarrow D + 1,\ pos \leftarrow 0,\ b' \leftarrow 1$
7: $\langle c_0 \rangle \| \langle c_1 \rangle \| \ldots \| \langle c_T \rangle \leftarrow$ cSHAKE128$(\mathsf{seed}_a, \mathrm{rate}_{\mathsf{XOF}} \cdot b', D)$, with $\langle c_t \rangle \in \{0,1\}^{8b}$
8: **end if**
9: **if** $c_{pos} \bmod 2^{\lceil \log_2 q \rceil} < q$ **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** $(a_1, ..., a_k)$

Gaussian sampling. One of the advantages of `qTESLA` is that discrete Gaussian sampling is only required during key generation to sample $e_1, ..., e_k$, and s (see

Alg. 3). Nevertheless, certain applications might still require an efficient and secure implementation of key generation and one that is, in particular, portable and protected against timing and cache side-channel attacks. Accordingly, we employ a *constant-time* discrete Gaussian sampler based on the well-established technique of 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 discrete 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 $\mathsf{CDT}[z] \leqslant u < \mathsf{CDT}[z+1]$. In the case of `qTESLA`, this method is very efficient due to the values of σ being relatively small, as can be seen in Table 4. In our implementations, the CDT method is implemented by generating a chunk of $c \mid n$ samples at a time. For example, we fix the chunk size to $c = 512$ when the dimension n is a multiple of 512. Then, to generate each sample in a chunk the precomputed CDT table is fully scanned, using constant-time logical and arithmetic operations to produce a Gaussian sample. To generate the required uniform samples, we use cSHAKE as XOF using the seed seed produced by PRF_1 as input string, and a nonce D (written as $\mathsf{counter}$ in Alg. 3) as domain separator. For the precomputed CDT tables, the targeted sampling precision β is conservatively set to a value much greater than $\lambda/2$, as can be seen in Table 2.

Table 2: CDT dimensions used in our implementations (targeted precision β : implemented precision in bits : number of rows t : table size in bytes).

Parameter set	CDT parameters
`qTESLA-p-I`	64 : 63 : 78 : 624
`qTESLA-p-III`	128 : 125 : 111 : 1776

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 $[-B, B+1]$ by subtracting 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. Algorithm 8 depicts the procedure used in our implementations. 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 D (written as $\mathsf{counter}$ in Algorithm 4) is intended for the computation of the values for the domain separation. The first call to the XOF function uses the value $D' \leftarrow D \cdot 2^8$ as domain separator. Each subsequent call to the XOF increases D' by 1. Since D 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 D' 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.

Algorithm 8 Sampling y, ySampler

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

$pos \leftarrow 0,\ n' \leftarrow n,\ D' \leftarrow D \cdot 2^8$
$\langle c_0 \rangle \| \langle c_1 \rangle \| \ldots \| \langle c_{n'-1} \rangle \leftarrow \mathsf{XOF}(\mathsf{rand}, b \cdot n', D')$, where each $\langle c_i \rangle \in \{0,1\}^{8b}$
while $i < n$ **do**
  if $pos \geq n'$ **then**
    $D' \leftarrow D' + 1,\ pos \leftarrow 0,\ n' \leftarrow \lfloor \mathrm{rate}_{\mathsf{XOF}}/b \rfloor$
    $\langle c_0 \rangle \| \langle c_1 \rangle \| \ldots \| \langle c_{n'-1} \rangle \leftarrow \mathsf{XOF}(\mathsf{rand}, \mathrm{rate}_{\mathsf{XOF}}, D')$, where each $\langle c_i \rangle \in \{0,1\}^{8b}$
  end if
  $y_i \leftarrow c_{pos} \bmod 2^{\lceil \log_2 B \rceil + 1} - B$
  if $y_i \neq B + 1$ **then**
    $i \leftarrow i + 1$
  end if
  $pos \leftarrow pos + 1$
end while
return $y = (y_0, y_1, \ldots, y_{n-1}) \in \mathcal{R}_{[B]}$

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 and the hash $\mathsf{G}(t_1, \ldots, t_k)$ 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 decreasing 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 $(v_i \bmod^{\pm} q - [v_i]_L)/2^d$. 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 40-byte digests $\mathsf{G}(m)$ and $\mathsf{G}(t_1, \ldots, t_k)$ 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 $c \in \mathbb{H}_{n,h} \subset \mathcal{R}$ 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 [1,25]. Basically, the idea is to use an XOF to generate values uniformly at random that are 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 $\mathrm{rate}_{\mathsf{XOF}}$ 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, be-

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 digests $\mathsf{G}(m)$ and $\mathsf{G}(t_1, \ldots, t_k)$, each of length 40 bytes.
Ensure: $c' \in \{0,1\}^{\kappa}$

for $i = 1, 2, \ldots, k$ **do**
  for $j = 0, 1, \ldots, n-1$ **do**
    val $\leftarrow v_{i,j} \mod 2^d$
    if val $> 2^{d-1}$ **then**
      val $\leftarrow$ val $- 2^d$
    end if
    $w_{(i-1)\cdot n+j} \leftarrow (v_{i,j} - \text{val})/2^d$
  end for
end for
$\langle w_{k\cdot n}\rangle \| \langle w_{k\cdot n+1}\rangle \| \ldots \| \langle w_{k\cdot n+39}\rangle \leftarrow \mathsf{G}(m)$, where each $\langle w_i\rangle \in \{0,1\}^8$
$\langle w_{k\cdot n+40}\rangle \| \langle w_{k\cdot n+41}\rangle \| \ldots \| \langle w_{k\cdot n+79}\rangle \leftarrow \mathsf{G}(t_1, \ldots, t_k)$, where each $\langle w_i\rangle \in \{0,1\}^8$
$c' \leftarrow \text{SHAKE}(w, \kappa/8)$, where w is the byte array $\langle w_0\rangle \| \langle w_1\rangle \| \ldots \| \langle w_{k\cdot n+79}\rangle$
return $c' \in \{0,1\}^{\kappa}$

ginning 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 $D = 0$ as domain separator for the first call. D is incremented by one at each subsequent call.

Polynomial multiplication based on the NTT. As mentioned earlier, the outputs $a_1, \ldots, a_k$ of GenA are assumed to be in the NTT domain. Hence, the polynomial multiplications with the form $a_i \cdot b$, for some $b \in \mathcal{R}_q$, can be efficiently realized as $\mathsf{NTT}^{-1}(a_i \circ \mathsf{NTT}(b))$.
To compute the power-of-two NTT in our implementations, we adopt butterfly algorithms that efficiently merge the powers of ϕ and ϕ^{-1} with the powers of ω, and that at the same time avoid the need for the so-called bit-reversal operation which is required by some implementations [9, 46, 49]. 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

Algorithm 10 Encoding function, Enc

Require: $c' \in \{0,1\}^{\kappa}$

Ensure: arrays $pos\_list \in \{0,...,n-1\}^h$ 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: $D \leftarrow 0$, $cnt \leftarrow 0$
2: $\langle r_0 \rangle \| \langle r_1 \rangle \| \dots \| \langle r_T \rangle \leftarrow$ cSHAKE128$(c', \text{rate}_{\mathsf{XOF}}, D)$, 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: $D \leftarrow D + 1$, $cnt \leftarrow 0$
8: $\langle r_0 \rangle \| \langle r_1 \rangle \| \dots \| \langle r_T \rangle \leftarrow$ cSHAKE128$(c', \text{rate}_{\mathsf{XOF}}, D)$, where each $\langle r_t \rangle \in \{0,1\}^8$
9: **end if**
10: $pos \leftarrow (r_{cnt} \cdot 2^8 + r_{cnt+1}) \mod n$
11: **if** $c_{pos} = 0$ **then**
12: **if** $r_{cnt+2} \mod 2 = 1$ **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 $\{sign\_list_0, \dots, sign\_list_{h-1}\}$

an algorithm based on the Gentleman-Sande butterfly that absorbs the inverses of the products of the root powers in bit-reversed ordering. The algorithm receives the inputs of a polynomial $\tilde{a}$ in bit-reversed ordering and produces an output in standard ordering. Polished versions of these well-known algorithms, which we follow in our implementations, can be found in [51, Algorithm 1 and 2].

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. An efficient algorithm to compute these sparse multiplications over $\mathbb{Z}_q[x]/\langle x^n + 1 \rangle$ is presented in Algorithm 11. This algorithm is used for the multiplications in lines 11 and 17 of Algorithm 4 and in line 4 of

Algorithm 11 Sparse polynomial multiplication for power-of-two cyclotomic rings

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 $pos\_list \in \{0, \ldots, n-1\}^h$ 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: $f = g \cdot c \in \mathcal{R}_q$

Set all coefficients of f to 0
for $i = 0, \ldots, h-1$ **do**
    $pos \leftarrow pos\_list_i$
    for $j = 0, \ldots, pos-1$ **do**
        $f_j \leftarrow f_j - sign\_list_i \cdot g_{j+n-pos}$
    end for
    for $j = pos, \ldots, n-1$ **do**
        $f_j \leftarrow f_j + sign\_list_i \cdot g_{j-pos}$
    end for
end for
return f

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 output by Enc).

5 Security and instantiations of qTESLA

In this section, we discuss qTESLA's security and the security proof in the QROM. Afterwards, we describe our two main approaches to instantiate the scheme, the hardness estimation of R-LWE, and the proposed parameter sets.

5.1 Provable security in the QROM

The standard security requirement for signature schemes, namely Existential Unforgeability under Chosen-Message Attack (EUF-CMA), dates back to Goldwasser, Micali, and Rivest [29]: 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.
qTESLA's EUF-CMA security is supported by a security reduction in the QROM [17], in which the adversary is granted access to a quantum random oracle. Namely, Theorem 1 gives a reduction from the R-LWE problem to the EUF-CMA security of qTESLA in the QROM. It is very similar to [8, Theorem 1], which gives the security reduction for qTESLA's predecessor TESLA. It is important to note that to port the reduction idea from TESLA over standard lattices to qTESLA over ideal lattices, we assume that Conjecture 1 holds. The formal

statement of qTESLA's security and a sketch of the proof, together with the required conjecture, is given below. This includes an expanded explanation on why the conjecture should hold true, as well as experimental results confirming our statements.
Under the conjecture and with parameters corresponding to Table 1, the security reduction is tight and explicit. This allows for choosing efficient provably secure parameters, i.e., to choose parameters according to the provided security statement, as explained in §5.2.

Remark 1. *(Expansion of public keys) The explanations below assume an "expanded" public key $(t_1, \ldots, t_k, a_1, \ldots, a_k)$. In the description of* qTESLA*, however, $a_1, \ldots, a_k$ 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 $(t_1, \ldots, t_k, \mathsf{seed}_a)$. Then we can construct an algorithm $\mathcal{P}$ that breaks a variant of* qTESLA *with "expanded" public key $(t_1, \ldots, t_k, a_1, \ldots, a_k)$. To this end, we model $\mathsf{GenA}(\cdot)$ as a (programmable) random oracle. The algorithm $\mathcal{P}$ chooses first $\mathsf{seed}'_a \leftarrow_{\$} \{0,1\}^{\kappa}$ and reprograms $\mathsf{GenA}(\mathsf{seed}'_a) = (a_1, \ldots, a_k)$. Afterwards, it forwards $(t_1, \ldots, t_k, \mathsf{seed}'_a)$ as input tuple to $\mathcal{A}$. Quantum queries to $\mathsf{GenA}(\cdot)$ by $\mathcal{A}$ can be simulated by $\mathcal{P}$ according to the construction of Zhandry based on $2q_h$-wise independent functions [52].*

5.1.1 Security reduction from R-LWE

Our security statement in Theorem 1 gives a reduction from the R-LWE problem to qTESLA's EUF-CMA security in the QROM. Theorem 1 holds assuming a conjecture, as explained below.

Theorem 1 (Security reduction from R-LWE)**.** *Let the parameters be as in Table 1, in particular, let $q^{nk} \geq 2^{4\lambda+nkd} 4q_s^3(q_s+q_h)^2$. Assume that Conjecture 1 holds. Assume that there exists a quantum adversary $\mathcal{A}$ that forges a* qTESLA *signature in time t_Σ, making at most q_h (quantum) queries to its quantum random oracle and q_s (classical) queries to its signing oracle. Then there exists a reduction $\mathcal{S}$ that solves the R-LWE problem with*

$$\mathrm{Adv}^{EUF\text{-}CMA}_{\texttt{qTESLA}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{R-LWE}}_{k,n,q,\sigma}(\mathcal{S}) + \frac{2^{3\lambda+nkd} \cdot 4 \cdot q_s^3(q_s+q_h)^2}{q^{nk}} + \frac{2(q_h+1)}{\sqrt{2^h \binom{n}{h}}} \quad (1)$$

and in time t_{LWE} which is about the same as t_Σ in addition to the time to simulate the quantum random oracle.

The idea of the security reduction is as follows. Let $\mathcal{A}$ be an algorithm that breaks qTESLA, i.e., given an "expanded" public key $(t_1, \ldots, t_k, a_1, \ldots, a_k)$, algorithm $\mathcal{A}$ outputs (z, c', m) after some time t_Σ. Let $\mathrm{forge}(t_1, \ldots, t_k, a_1, \ldots, a_k)$ denote the event that the forger $\mathcal{A}$ successfully produces a *valid* signature for $(t_1, \ldots, t_k, a_1, \ldots, a_k)$, i.e., with probability $\Pr[\mathrm{forge}(t_1, \ldots, t_k, a_1, \ldots, a_k)]$, (z, c') is a *valid* signature of message m. We model the hash-based function H

as a quantum random oracle. In particular, algorithm $\mathcal{A}$ is allowed to make (at most) q_h quantum queries to a quantum random oracle H 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 probability of success close to $\Pr[\text{forge}(t_1, \ldots, t_k, a_1, \ldots, a_k)]$. The reduction $\mathcal{S}$ gets as input a tuple $(t_1, \ldots, t_k, a_1, \ldots, a_k)$ and must decide whether it follows the R-LWE distribution (see Definition 1) or $\mathcal{U}(\mathcal{R}_q^{2k})$. It forwards $(t_1, \ldots, t_k, a_1, \ldots, a_k)$ as the public key to $\mathcal{A}$. In the reduction, $\mathcal{S}$ must simulate the responses to $\mathcal{A}$'s queries to the hash and sign oracles. It is then shown that if $(t_1, \ldots, t_k, a_1, \ldots, a_k)$ follows the R-LWE distribution then the probability that $\mathcal{S}$ answers *correctly* is close to $\Pr[\text{forge}(t_1, \ldots, t_k, a_1, \ldots, a_k)]$. Furthermore, if $(t_1, \ldots, t_k, a_1, \ldots, a_k)$ follows the uniform distribution over $\mathcal{R}_q^{2k}$ then $\mathcal{S}$ returns the *wrong* answer with negligible probability.
The proof follows closely the approach proposed in [8], that shows the security of qTESLA's predecessor TESLA, except for the computation of the two probabilities $\text{coll}(\overrightarrow{a}, \overrightarrow{e})$ and $\text{nwr}(\overrightarrow{a}, \overrightarrow{e})$ that we define and explain next. For simplicity, we assume that the randomness is sampled uniformly random in $\mathcal{R}_{[B]}$. We call a polynomial f *well-rounded* if $f \in \mathcal{R}_{[\lfloor q/2 \rfloor - E]}$ and $[f]_L \in \mathcal{R}_{[(2^{d-1}-E)]}$. For our discussion we also define the following sets of polynomials:

$$\begin{aligned}
\mathbb{Y} &= \{y \in \mathcal{R}_{[B]}\},\\
\Delta\mathbb{Y} &= \left\{y - y' \ : \ y, y' \in \mathcal{R}_{[B]}\right\} = \mathcal{R}_{[2B]},\\
\Delta\mathbb{L} &= \{x - x' : x, x' \in \mathcal{R} \text{ and } [x]_M = [x']_M\},\\
\mathbb{H}_{n,h} &= \{c \in \mathcal{R}_{[1]} \mid \|c\|_2 = \sqrt{h}\},\\
\Delta\mathbb{H}_{n,h} &= \{c - c' \ : \ c, c' \in \mathbb{H}_{n,h}\},\\
\mathbb{W} &= \{[w]_M : w \in \mathcal{R}_q\}.
\end{aligned}$$

Moreover, we denote $\overrightarrow{a} = (a_1, ..., a_k)$ and $\overrightarrow{e} = (e_1, ..., e_k)$ and define $\text{nwr}(\overrightarrow{a}, \overrightarrow{e})$ $:= \Pr_{(y,c) \in \mathcal{R}_{[B]} \times \mathbb{H}_{n,h}}[a_i y - e_i c \text{ not well-rounded for at least one } i \in \{1, ..., k\}]$. This quantity varies as a function of $a_1, ..., a_k, e_1, ..., e_k$. In contrast to [8], we cannot upper bound this in general in the ring setting. Instead, we check experimentally that our acceptance probability for w_i in line 18 of Algorithm 4 (signature generation) is at least $1/4$ for our provably secure parameter sets (see Table 4). Hence, $\text{nwr}(\overrightarrow{a}, \overrightarrow{e}) < 3/4$.

Secondly, we need to bound the probability

$$\text{coll}(\overrightarrow{a}, \overrightarrow{e}) := \max_{(w_1, ..., w_k) \in \mathbb{W}^k} \left\{ \Pr_{(y,c) \in \mathcal{R}_{[B]} \times \mathbb{H}_{n,h}} [[a_i y - e_i c]_M = w_i \text{ for } i = 1, ..., k] \right\}.$$

In [8, Lemma 4] the corresponding probability $\text{coll}(A, E)$ for standard lattices is upper bounded, given $A \in \mathbb{Z}_q^{m \times n}$, $E \in \mathbb{Z}_q^{m \times n'}$, and $n, m, n' \in \mathbb{Z}$. Unfortunately, the proof does not carry over to the ring setting for the following reason. In the proof of [8, Lemma 4], it is used that if the randomness $y \in [-B, B]^n$ is not equal to 0, the vector $Ay \mod q$ 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 the *polynomial* y is chosen uniformly in $\mathcal{R}_{[B]}$. Moreover,

in [8, Equation (99)], ψ denotes the probability that a random vector $x \in \mathbb{Z}_q^m$ is in $\Delta\mathbb{L}$, i.e.,

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

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 need to *conjecture* the following.

Conjecture 1. *Let y be a random element of $\Delta\mathbb{Y}$ and $a_1, \ldots, a_k$ be k random elements in the ring $\mathcal{R}_q$. Then with only negligible probability it will be the case that for each i, each coefficient of $a_i y$ is in $\{-2^d - 2E + 1, \ldots, 2^d + 2E - 1\}$. More formally,*

$$\Pr_{(\overrightarrow{a}, y) \leftarrow_\$ \mathcal{R}^k \times \mathcal{R}_{[2B]}}[\forall i \in \{1, \ldots, k\} : \; a_i \cdot y \in \mathcal{R}_{[2^d+2E-1]}] \leq \frac{1}{2^{-(n+2\lambda)}} \cdot \frac{|\mathbb{H}_{n,h}|}{|\Delta\mathbb{H}_{n,h}|}.$$

We briefly describe why this conjecture is needed in the security reduction for `qTESLA`, and why it should be expected to be true.

This conjecture is needed in bounding the value $\mathrm{coll}(\overrightarrow{a}, \overrightarrow{e})$ which corresponds to the maximum likelihood that the values $a_i y - e_i c \mod q$ round to some specific values. For example, if during key generation all of the a_i's are set to be 0, then the rounding of any $a_i y - e_i c$ is 0, which is a poor choice of a public key.

In the proof of TESLA [8], to establish that such an event is unlikely to occur, the value $\mathbb{G}(A, E)$ was defined, and a relation was shown between the values $\mathrm{coll}(A, E)$ and $\mathbb{G}(A, E)$. For `qTESLA`, we can define $\mathbb{G}(\overrightarrow{a}, \overrightarrow{e})$ as

$$\mathbb{G}(\overrightarrow{a}, \overrightarrow{e}) = \{(y, c) \in \Delta\mathbb{Y} \times \Delta\mathbb{H}_{n,h} : \forall i, a_i y - e_i c \in \Delta\mathbb{L}\}. \tag{3}$$

A similar relation holds for `qTESLA` so that we can derive a bound on $\mathrm{coll}(\overrightarrow{a}, \overrightarrow{e})$ from a bound on $\mathbb{G}(\overrightarrow{a}, \overrightarrow{e})$. Specifically, following the same logic as [8, Lemma 5], we can see that

$$\mathrm{coll}(\overrightarrow{a}, \overrightarrow{e}) \leq \frac{\mathbb{G}(\overrightarrow{a}, \overrightarrow{e})}{|\mathbb{Y}| \cdot |\mathbb{H}_{n,h}|}. \tag{4}$$

In fact, we can largely drop the $\overrightarrow{e}$ part of the equation, and simply write $\mathbb{G}(\overrightarrow{a})$. Because each e_i is chosen so that $e_i c$ is always quite small, we can see that the rounding $[a_i y]_M$ of $a_i y$ is almost always the same as $[a_i y - e_i c]_M$, and ignore the effects of $e_i c$. By considering the maximum difference between two elements that round to the same value, we can replace $\Delta\mathbb{L}$ with $\mathcal{R}_{[2^d-1]}$. Then since each coefficient of $e_i c \in \Delta\mathbb{H}_{n,h}$ is at most $2E$, we can see that $a_i y - e_i c \in \Delta\mathbb{L}$ implies that $a_i y \in \mathcal{R}_{[2^d+2E-1]}$. This allows us to define the set $\mathbb{G}(\overrightarrow{a}) = \{y \in \Delta\mathbb{Y} : \forall i, a_i y \in \mathcal{R}_{[2^d+2E-1]}\}$ and establish that $|\mathbb{G}(\overrightarrow{a}, \overrightarrow{e})| \leq |\Delta\mathbb{H}_{n,h}| \cdot |\mathbb{G}(\overrightarrow{a})|$.

To demonstrate a bound on the size of $\mathbb{G}(\overrightarrow{a})$, we calculate the expected value and employ Markov's inequality. Very similarly to the logic of [8, AppendixB.9],

we determine that the expected size of $\mathbb{G}(\overrightarrow{a})$ is equal to $|\Delta\mathbb{Y}|$ times the probability that appears in Equation (1).

If this probability is lower than the bound in Equation (1), then we can employ Markov's inequality to establish that

$$\Pr_{\overrightarrow{a},\overrightarrow{e}}[\text{coll}(\overrightarrow{a},\overrightarrow{e}) \geq 2^{-\lambda}] \leq 2^{-\lambda}. \tag{5}$$

Here we sketch a brief argument as to why this conjecture should be expected to be true. The set $\mathcal{R}_{q,[2^d+2E-1]}$ forms an incredibly tiny fraction of our ring R_q. That fraction is $(2^{d+1}+4E-1)^n/q^n$. For the `qTESLA-p-III` parameter set, it is approximately $1/2^{10,000}$. So the closer picking a random $\overrightarrow{a}$ and y and computing the products is to picking k uniform elements, the closer we get to this fraction.

For invertible y, it is easy to see that this corresponds exactly to picking out k uniform elements, and so the probability is much lower than we need. All that must be accounted for is the non-invertible y. For these, it should hold that the ideal generated by y still only has a negligible fraction that is in $R_{q,[2^d+2E-1]}$, and indeed it should be the case that only a small part of $\Delta\mathbb{Y}$ is non-invertible.

To allow experimentation with our parameters, we wrote a script[6] that samples such a y and a and checks if their product is in $\mathcal{R}_{[2^d+2E-1]}$. After running the script over the parameter sets `qTESLA-p-I` and `qTESLA-p-III` 10,000 times each, we did not observe an instance in which a uniform element of $\mathcal{R}_q$ and $\mathcal{R}_{[2B]}$ was in $\mathcal{R}_{[2^d+2E-1]}$. This supports the claim that our conjecture holds true for the provably secure instantiations of `qTESLA`.

5.2 Relation between the R-LWE hardness and qTESLA's security

Our parameters are chosen such that $\epsilon_{LWE} \approx \epsilon_\Sigma$ and $t_\Sigma \approx t_{LWE}$[7], 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 and the aforementioned conjecture holds. This approach provides a strong security argument. We emphasize that our provably secure parameters are chosen according to their security reductions from R-LWE

[6] The script can be found at `/Supporting_Documentation/Script_for_conjecture/Script_for_experiments_supporting_the_security_conjecture.py` in the `qTESLA` submission package available at `https://qtesla.org`.

[7] To be precise, we assume that the time to simulate the (quantum) random oracle is smaller than the time to forge a signature. This assumption is commonly made in "provably secure" cryptography.

but not according to reductions from underlying existing worst-case to average-case reductions from SIVP or GapSVP to R-LWE [42]. In this work, we propose two *provably secure* parameter sets called `qTESLA-p-I` and `qTESLA-p-III`; see §5.4.

Remark 2. *In practical instantiations of* `qTESLA`*, the bit security does not exactly match the bit hardness of R-LWE (see Table 4). 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 key pairs and on the security of other building blocks such as the encoding function* Enc*. First, in all our parameter sets 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.*

5.3 Hardness estimation of our instances

Lattice reduction is arguably the most important building block in most efficient attacks against R-LWE instances. As the Block-Korkine-Zolotarev algorithm (BKZ) [20, 21] is considered the most efficient lattice reduction in practice, the model used to estimate the cost of BKZ determines the overall hardness estimation. While many different cost models for BKZ exist [4], we decided to adopt the BKZ cost model of $0.265\beta + 16.4 + \log_2(8d)$ for the hardness estimation of our parameters (denoted by `BKZ.qsieve`), where β is the BKZ block size and d is the lattice dimension. It corresponds to solving instances of the shortest vector problem of blocksize β with a quantum sieving algorithm [36, 37]. This cost model is conservative since it only takes into account the number of operations needed to solve a certain instance and assumes that the attacker can handle huge amounts of quantum memory. We compare our chosen hardness estimation for R-LWE in Table 3 with other BKZ models, including the one from [9] (denoted by `BKZ.ADPS16`) and the classical algorithms using sieving [14] (denoted by `BKZ.sieve`).

Table 3: Security estimation (bit hardness) under different BKZ cost models.

BKZ cost model	`qTESLA-p-I`	`qTESLA-p-III`
`BKZ.sieve`	150	304
`BKZ.qsieve`	**139**	**279**
`BKZ.ADPS16`	108	247

Hardness estimation of R-LWE. Since its introduction in [42], 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. Sev-

eral results exist that exploit the structure of some ideal lattices, e.g., [22, 28]. 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 [7] presented the *LWE-Estimator*, a software to estimate the hardness of LWE given the matrix dimension n, the modulus q, the relative error rate $\alpha = (\sqrt{2\pi}\sigma)/q$, 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 [31], the recent dual lattice attacks in [2], the enumeration approach by Linder and Peikert [38], the primal attack [5, 12], the Arora-Ge algorithm [10] using Gröbner bases [3], and the latest analysis to compute the block sizes used in the lattice basis reduction BKZ by Albrecht *et al.* [6]. Moreover, quantum speedups for the sieving algorithm used in BKZ [36, 37] are also considered. We integrated the LWE-Estimator with commit-id `3019847` on 2019-02-14 in the sage script that we wrote to perform the security estimation.

5.4 Parameter sets

We propose two parameter sets called `qTESLA-p-I` and `qTESLA-p-III`, which are summarized in Table 4.

6 Implementation and performance evaluation

6.1 Portable C implementation

Our compact reference implementation is written exclusively in portable C using approximately 300 lines of code. It exploits the fact that it is straightforward to write a `qTESLA` implementation with a common codebase, since the different parameter set realizations only differ in some packing functions and system constants that can be instantiated at compilation time. This illustrates the simplicity and scalability of software based on `qTESLA`.

Protection against side-channel attacks. All our implementations run in *constant-time*, i.e., they avoid the use of secret address accesses and secret branches and, hence, are protected against timing and cache side-channel attacks. The following functions are implemented securely via constant-time logical and arithmetic operations: H, checkE, checkS, the correctness test for rejection sampling, polynomial multiplication using the NTT, sparse multiplication, and all the polynomial operations requiring modular reductions or corrections.

Table 4: Parameters for each of the proposed parameter sets with $q_h = 2^{128}$ and $q_s = 2^{64}$; we choose $\kappa = 256$.

Param.	qTESLA-p-I	qTESLA-p-III
λ	95	160
n, k	1 024, 4	2 048, 5
σ	8.5	8.5
q	343 576 577 $\approx 2^{28}$	856 145 921 $\approx 2^{30}$
h	25	40
L_E, E L_S, S	554 554	901 901
B, d	$2^{19} - 1$, 22	$2^{21} - 1$, 24
b_{GenA}	108	180
δ_w, δ_z δ_{sign}, M δ_{keygen}	0.37, 0.34 0.13, 0.3 0.59	0.33, 0.42 0.14, 0.3 0.43
sig size [bytes] pk size [bytes] sk size [bytes]	2, 592 14, 880 5, 224	5, 664 38, 432 12, 392
classical bit hardness quantum bit hardness	150 139	304 279

Some of the functions that perform some form of rejection sampling, such as the security test at signing, GenA, 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 producing samples via constant-time logical and arithmetic operations.

Polynomial arithmetic. Our polynomial arithmetic, which is dominated by polynomial multiplications based on the NTT, uses a signed 32-bit datatype to represent coefficients. Throughout polynomial computations, intermediate results are let to grow 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 public keys). Accordingly, to avoid overflows the results of additions and subtractions are either corrected or reduced via Barrett reductions whenever necessary. We have performed a careful bound analysis for each of the proposed parameter sets in order to maximize the use of lazy reduction and cheap modular corrections in the polynomial arithmetic. In the case of multiplications, the results are reduced via Montgomery reductions. To minimize the cost of converting to/from Montgomery representation we use the following approach. First, the so-called "twiddle fac-

tors" in the NTT are scaled *offline* by multiplying with the Montgomery constant $R = 2^{32} \bmod q$. Similarly, the coefficients of the outputs a_i from GenA are scaled to remainders $r' = rn^{-1}R \bmod q$ 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 of GenA which is assumed to be in NTT domain. 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 for explicit conversions.

6.2 AVX2 optimizations

We optimized two functions with hand-written assembly exploiting AVX2 vector instructions, namely, polynomial multiplication and XOF expansion during sampling of y.

Our polynomial multiplication follows the recent approach by Seiler [51], and the realization of the method has some similarities with the implementation from [26]. 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 for storing and loading intermediate results, whenever possible. We illustrate our procedure for a polynomial a of dimension $n = 512$ written as the vector of coefficients $(a_0, a_1, \ldots, a_{511})$. First, 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 for 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 [26, 51] is that our NTT coefficients are represented as 32-bit *signed* integers, which motivates a speedup in the butterfly computation by

Table 5: Performance (in thousands of cycles) of the portable C and 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
Portable C implementation			
qTESLA-p-I	2,358.6 (2,431.9)	2,299.0 (3,089.9)	814.3 (814.5)
qTESLA-p-III	13,151.4 (13,312.4)	5,212.3 (7,122.6)	2,102.3 (2,102.6)
AVX2 implementation			
qTESLA-p-I	2,212.4 (2,285.0)	1,370.4 (1,759.0)	678.4 (678.5)
qTESLA-p-III	12,791.0 (13,073.4)	3,081.9 (4,029.5)	1,745.3 (1,746.4)

avoiding the extra additions that are required to make the result of subtractions positive when using an unsigned representation.
Our approach reduces the cost of the portable C polynomial multiplication from 76,300 to 18,400 cycles for $n = 1024$, and from 174,800 to 43,900 cycles for $n = 2048$.

Sampling of y is sped up by using the AVX2 implementation of SHAKE by Bertoni, Daemen, Hoffert, Peeters, Van Assche, and Van Keer [15], 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 it degrades the performance on smaller platforms with no vector instruction support.

6.3 Performance on x64

We evaluated the performance of our implementations on an 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 portable C and AVX2 implementations are summarized in Table 5.

qTESLA computes the combined (median) time of signing and verification on the Skylake platform in approximately 0.92 and 2.15 milliseconds with qTESLA-p-I and qTESLA-p-III, respectively. This demonstrates that the speed of qTESLA, although slower than other lattice-based signature schemes, can still be considered practical for most applications.

The AVX2 optimizations improve the performance by a factor 1.5x, approximately. The speedup is mainly due to the AVX2 implementation of the polynomial multiplication, which is responsible for $\sim$ 70% of the total speedup. qTESLA computes the combined (median) time of signing and verification on the

Skylake platform in approximately 0.60 and 1.42 milliseconds with `qTESLA-p-I` and `qTESLA-p-III`, respectively.

We note that the overhead of including g, i.e., the hash of part of the public key, in the signature computation of c' is between 3–8% of the combined cost of signing and verification.

6.4 Comparison

Table 6 compares `qTESLA` to representative state-of-the-art post-quantum signature schemes in terms of bit security, signature and public key sizes, and performance of portable C 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`, 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 schemes that use other cost models, we write in brackets the bit security stated in the respective papers).

FALCON-512, the only other scheme proposing parameters according to their (tight) security reduction, features the smallest (pk + sig) size among all the post-quantum signature schemes shown in the table. However, Falcon has some shortcomings due to its high complexity. This scheme relies on very complex Fourier sampling methods and requires floating-point arithmetic, which is not supported by many devices. This makes the scheme significantly harder to implement in general, and hard to protect against side-channel and fault attacks in particular. The recent efficient implementation by Pornin [47] makes use of complicated floating-point emulation code to deal with the portability issues, and contains several thousands of lines of C code. Still, the software cannot be labeled as a strictly *constant-time* implementation because some portions of it allow a limited amount of leakage to happen. This should be contrasted against the simple and compact implementation of `qTESLA`.

Schemes based on other underlying problems, such as SPHINCS$^+$ and MQDSS, offer compact public keys at the expense of having very large signature sizes. In contrast, `qTESLA` has smaller signature sizes, and is significantly faster for signing and verification.

In summary, `qTESLA` offers a good balance between efficiency, accompanied by a simple, compact, and secure design.

References

[1] Akleylek, S., Bindel, N., Buchmann, J.A., Krämer, J., Marson, G.A.: An efficient lattice-based signature scheme with provably secure instantiation.

Table 6: Comparison of different post-quantum signature schemes

	Scheme	Security [bit]	const.-time	Sizes [B]		Cycle counts [k-cycles] Reference	AVX2	CPU
Selected lattice-based signatures	BLISS-BI [24, 25]	128	✗	pk: 896 sig: 717	sign: ≈435.2 verify: ≈102.0		-	U
	FALCON-512[a] [47]	158[b] (103)	✗	pk: 897 sig: 617	sign: 1,368.5 verify: 95.6		1,009.8 81.0	S
	Dilithium-I [41]	77[b] (53)	✓	pk: 896 sig: 1,387	sign: 785.2 verify: 172.7		265.2 78.2	S
	Dilithium-II [41]	122[b] (91)	✓	pk: 1,184 sig: 2,044	sign: 1,378.1 verify: 272.8		410.7 109.0	S
	Dilithium-III [41]	160[b] (125)	✓	pk: 1,472 sig: 2,701	sign: 2,035.9 verify: 375.7		547.2 155.8	S
	`qTESLA-p-I` [a] (this paper)	95[b]	✓	pk: 14,880 sig: 2,592	sign: 3,089.9 verify: 814.3		1,759.0 678.5	S
	`qTESLA-p-III` [a] (this paper)	160[b]	✓	pk: 38,432 sig: 5,664	sign: 7,122.6 verify: 2,102.3		4,029.5 1,746.4	S
Others	SPHINCS$^+$-128f-s[a] (SHAKE256) [32]	128[c]	✓	pk: 32 sig: 16,976	sign: 325,311 verify: 13,541		129,137 9,385	H
	MQDSS-31-64 [50]	128[c]	✓	pk: 64 sig: 43,728	sign: 85,268.7 verify: 62,306.1		9,047.1 6,133.0	H

[a] Parameters are chosen according to given security reduction in the ROM/QROM.

[b] Bit security against classical and quantum adversaries with BKZ cost model $0.265\beta + 16.4 + \log_2(8d)$ [4]; (originally stated bit security given in brackets).

[c] Bit security analyzed against classical and quantum adversaries.

U: Unknown 3.4GHz Intel Core for BLISS.

S: 3.3GHz Intel Core i7-6567U (Skylake) for FALCON-512 (TurboBoost enabled), 2.6GHz Intel Core i7-6600U (Skylake) for Dilithium, and 3.4GHz Intel Core i7-6700 (Skylake) for `qTESLA`.

H: 3.5GHz Intel Core i7-4770K (Haswell).

In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2016. LNCS, vol. 9646, pp. 44–60. Springer (2016)

[2] Albrecht, M.R.: On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In: Coron, J., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 103–129. Springer, Heidelberg (Apr / May 2017)

[3] Albrecht, M.R., Cid, C., Faugère, J., Fitzpatrick, R., Perret, L.: Algebraic algorithms for LWE problems. ACM Comm. Computer Algebra 49(2), 62 (2015)

[4] Albrecht, M.R., Curtis, B.R., Deo, A., Davidson, A., Player, R., Postlethwaite, E.W., Virdia, F., Wunderer, T.: Estimate all the LWE, NTRU schemes! In: Catalano, D., De Prisco, R. (eds.) SCN 18. LNCS, vol. 11035, pp. 351–367. Springer, Heidelberg (Sep 2018)

[5] Albrecht, M.R., Fitzpatrick, R., Göpfert, F.: On the efficacy of solving LWE by reduction to unique-SVP. In: Lee, H.S., Han, D.G. (eds.) Information Security and Cryptology - ICISC 2013. LNCS, vol. 8565, pp. 293–310. Springer (2013)

[6] Albrecht, M.R., Göpfert, F., Virdia, F., Wunderer, T.: Revisiting the expected cost of solving uSVP and applications to LWE. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part I. LNCS, vol. 10624, pp. 297–322. Springer, Heidelberg (Dec 2017)

[7] Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology 9(3), 169–203 (2015)

[8] Alkim, E., Bindel, N., Buchmann, J.A., Dagdelen, Ö., Eaton, E., Gutoski, G., Krämer, J., Pawlega, F.: Revisiting TESLA in the quantum random oracle model. In: Lange, T., Takagi, T. (eds.) Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017. pp. 143–162. Springer, Heidelberg (2017)

[9] Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16. pp. 327–343. USENIX Association (2016)

[10] Arora, S., Ge, R.: New algorithms for learning in presence of errors. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 403–415. Springer, Heidelberg (Jul 2011)

[11] Bai, S., Galbraith, S.D.: An improved compression technique for signatures based on learning with errors. In: Benaloh, J. (ed.) CT-RSA 2014. LNCS, vol. 8366, pp. 28–47. Springer, Heidelberg (Feb 2014)

[12] Bai, S., Galbraith, S.D.: Lattice decoding attacks on binary LWE. In: Susilo, W., Mu, Y. (eds.) ACISP 14. LNCS, vol. 8544, pp. 322–337. Springer, Heidelberg (Jul 2014)

[13] Barreto, P.S.L.M., Longa, P., Naehrig, M., Ricardini, J.E., Zanon, G.: Sharper ring-LWE signatures. Cryptology ePrint Archive, Report 2016/1026 (2016), http://eprint.iacr.org/2016/1026

[14] Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In: Krauthgamer, R. (ed.) 27th SODA. pp. 10–24. ACM-SIAM (Jan 2016)

[15] Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., , Assche, G.V., Keer, R.V.: The eXtended Keccak Code Package (XKCP), https://github.com/XKCP/XKCP

[16] Blake-Wilson, S., Menezes, A.: Unknown key-share attacks on the station-to-station (STS) protocol. In: Imai, H., Zheng, Y. (eds.) Public Key Cryptography (PKC'99). Lecture Notes in Computer Science, vol. 1560, pp. 154–170. Springer (1999)

[17] Boneh, D., Dagdelen, Ö., Fischlin, M., Lehmann, A., Schaffner, C., Zhandry, M.: Random oracles in a quantum world. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 41–69. Springer, Heidelberg (Dec 2011)

[18] Bruinderink, L.G., Pessl, P.: Differential fault attacks on deterministic lattice signatures. IACR TCHES 2018(3), 21–43 (2018), https://tches.iacr.org/index.php/TCHES/article/view/7267

[19] Cantero, H., Peter, S., Bushing, Segher: Console hacking 2010 – PS3 epic fail. 27th Chaos Communication Congress (2010), https://www.cs.cmu.edu/~dst/GeoHot/1780_27c3_console_hacking_2010.pdf

[20] Chen, Y.: Réduction de réseau et sécurité concrète du chiffrement complètement homomorphe. Ph.D. thesis, Paris, France (2013)

[21] Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better lattice security estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Heidelberg (Dec 2011)

[22] Cramer, R., Ducas, L., Peikert, C., Regev, O.: Recovering short generators of principal ideals in cyclotomic rings. In: Fischlin, M., Coron, J. (eds.) Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques. LNCS, vol. 9666, pp. 559–585. Springer (2016)

[23] Dagdelen, Ö., Bansarkhani, R.E., Göpfert, F., Güneysu, T., Oder, T., Pöppelmann, T., Sánchez, A.H., Schwabe, P.: High-speed signatures from standard lattices. In: Aranha, D.F., Menezes, A. (eds.) Progress in Cryptology – LATINCRYPT 2014. LNCS, vol. 8895, pp. 84–103. Springer (2015)

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

[25] Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal Gaussians. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 40–56. Springer, Heidelberg (Aug 2013)

[26] Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: 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

[27] Dworkin, M.J.: 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

[28] Elias, Y., Lauter, K.E., Ozman, E., Stange, K.E.: Provably weak instances of ring-LWE. In: Gennaro, R., Robshaw, M. (eds.) Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference. LNCS, vol. 9215, pp. 63–92. Springer (2015)

[29] Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on Computing 17(2), 281–308 (Apr 1988)

[30] Güneysu, T., Lyubashevsky, V., Pöppelmann, T.: Practical lattice-based cryptography: A signature scheme for embedded systems. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 530–547. Springer, Heidelberg (Sep 2012)

[31] Guo, Q., Johansson, T., Stankovski, P.: Coded-BKW: Solving LWE using lattice codes. In: Gennaro, R., Robshaw, M. (eds.) Advances in Cryptology – CRYPTO 2015, LNCS, vol. 9215, pp. 23–42. Springer (2015)

[32] Hulsing, A., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Aumasson, J.P.: SPHINCS+. Tech. rep., National Institute of Standards and Technology (2019), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions

[33] Jackson, D., Cremers, C., Cohn-Gordon, K., Sasse, R.: Seems legit: Automated analysis of subtle attacks on protocols that use signatures. Cryptology ePrint Archive, Report 2019/779 (to appear at ACM CCS'19) (2019), https://eprint.iacr.org/2019/779

[34] Kelsey, J.: SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. NIST Special Publication 800, 185 (2016), avail-

able at http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf

[35] Kiltz, E., Lyubashevsky, V., Schaffner, C.: A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 552–586. Springer, Heidelberg (Apr / May 2018)

[36] Laarhoven, T.: Search problems in cryptography. Ph.D. thesis, Eindhoven University of Technology (2016)

[37] Laarhoven, T., Mosca, M., Pol, J.: Solving the Shortest Vector Problem in Lattices Faster Using Quantum Search. In: Gaborit, P. (ed.) Post-Quantum Cryptography, Lecture Notes in Computer Science, vol. 7932, pp. 83–101. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-38616-9_6

[38] Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 319–339. Springer, Heidelberg (Feb 2011)

[39] Lyubashevsky, V.: Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 598–616. Springer, Heidelberg (Dec 2009)

[40] Lyubashevsky, V.: Lattice signatures without trapdoors. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 738–755. Springer, Heidelberg (Apr 2012)

[41] Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2019), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions

[42] Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 1–23. Springer, Heidelberg (May / Jun 2010)

[43] Menezes, A., Smart, N.P.: Security of signature schemes in a multi-user setting. Des. Codes Cryptogr. 33(3), 261–274 (2004)

[44] 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

[45] Poddebniak, D., Somorovsky, J., Schinzel, S., Lochter, M., Rösler, P.: Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014 (2017), http://eprint.iacr.org/2017/1014

[46] Pöppelmann, T., Güneysu, T.: Towards practical lattice-based public-key encryption on reconfigurable hardware. In: Lange, T., Lauter, K.E., Lisonek, P. (eds.) Selected Areas in Cryptography - SAC 2013. Lecture Notes in Computer Science, vol. 8282, pp. 68–85. Springer (2014)

[47] Pornin, T.: New efficient, constant-time implementations of Falcon. `https://falcon-sign.info/falcon-impl-20190918.pdf` (2019), accessed: 2019-10-11

[48] Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Tech. rep., National Institute of Standards and Technology (2019), available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`

[49] Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact Ring-LWE cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2014. Lecture Notes in Computer Science, vol. 8731, pp. 371–391. Springer (2014)

[50] Samardjiska, S., Chen, M.S., Hulsing, A., Rijneveld, J., Schwabe, P.: MQDSS. Tech. rep., National Institute of Standards and Technology (2019), available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`

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

[52] Zhandry, M.: Secure identity-based encryption in the quantum random oracle model. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 758–775. Springer, Heidelberg (Aug 2012)

[53] Zhang, Z., Chen, C., Hoffstein, J., Whyte, W.: pqNTRUSign – Submission to the NIST's post-quantum cryptography standardization process (round 1) (2017), available at `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/pqNTRUsign.zip`

A Correctness of qTESLA

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. 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) $z \in \mathcal{R}_{[B-S]}$ 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 $[a_i y]_M = [a_i z - t_i c]_M = [a_i(y + sc) - (a_i s + e_i)c]_M = [a_i y + a_i sc - a_i sc - e_i c]_M = [a_i y - e_i c]_M$. From the definition of $[\cdot]_M$, this means proving that $(a_i y \bmod^{\pm} q - [a_i y]_L)/2^d = (a_i y - e_i c \bmod^{\pm} q - [a_i y - e_i c]_L)/2^d$, or simply $[a_i y]_L = e_i c + [a_i y - e_i c]_L$.

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

$$[\alpha]_L = \varepsilon + [\alpha - \varepsilon]_L. \tag{6}$$

Proof. To prove Equation (6), start by noticing that $|\varepsilon| \leq E < 2^{d-1}$ implies $[\varepsilon]_L = \varepsilon$. Thus, from $-2^{d-1} + E < [\alpha - \varepsilon]_L < 2^{d-1} - E$ and $-E \leq [\varepsilon]_L \leq E$ it follows that

$$-2^{d-1} = -2^{d-1} + E - E < [\varepsilon]_L + [\alpha - \varepsilon]_L < 2^{d-1} - E + E = 2^{d-1},$$

and therefore

$$[[\varepsilon]_L + [\alpha - \varepsilon]_L]_L = [\varepsilon]_L + [\alpha - \varepsilon]_L = \varepsilon + [\alpha - \varepsilon]_L. \tag{7}$$

Next we prove that

$$[[\varepsilon]_L + [\alpha - \varepsilon]_L]_L = [\alpha]_L. \tag{8}$$

Since $|\varepsilon| \leq E < \lfloor q/2 \rfloor$ and $|\alpha - \varepsilon \bmod^{\pm} q| < \lfloor q/2 \rfloor$, it holds further that

$$\begin{aligned} & [[\varepsilon]_L + [\alpha - \varepsilon]_L]_L && (9) \\ = \;& ((\varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d + (\alpha - \varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d) \bmod^{\pm} q \bmod^{\pm} 2^d && (10) \\ = \;& (\varepsilon \bmod^{\pm} q + (\alpha - \varepsilon \bmod^{\pm} q)) \bmod^{\pm} 2^d. && (11) \end{aligned}$$

Since $|\varepsilon| \leq E$ and $|\alpha - \varepsilon \bmod^{\pm} q| < \lfloor q/2 \rfloor - E$, it holds that $|\alpha - \varepsilon| + |\varepsilon| < (\lfloor q/2 \rfloor - E) + E = \lfloor q/2 \rfloor$. Hence, Equation (11) is the same as

$$= \quad (\varepsilon + \alpha - \varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d = (\alpha \bmod^{\pm} q) \bmod^{\pm} 2^d = [\alpha]_L.$$

By combining Equation (7) and Equation (8) we deduce that $[\alpha]_L = \varepsilon + [\alpha - \varepsilon]_L$, which is the equation we needed to prove. □

Now define $\alpha := (a_i y)_j$ and $\varepsilon := (e_i c)_j$ with $i \in \{1, ..., k\}$ and $j \in \{0, ..., n-1\}$. From line 18 of Algorithm 4, we know that for $i = 1, ..., k$, $\|[a_i y - e_i c]_L\|_\infty < 2^{d-1} - E$ and $\|a_i y - e_i c\|_\infty < \lfloor q/2 \rfloor - E$ for an honestly generated signature, and that Algorithm 3 (line 13) guarantees $\|e_i c\|_\infty \leq E$. Likewise, by definition it holds that $E < \lfloor q/2 \rfloor$; see §5. 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 $i = 1, ..., k$.